Two Self-Adaptive Crossover Operations for Genetic Programming

Peter J. Angeline

Loral Federal Systems - Owego

pja@lfs.loral.com

To Appear in

Advances in Genetic Programming: Volume 2

Peter J. Angeline

Two self-adaptive crossover operations are studied in a nonstandard genetic program. It is shown that for three distinct problems the results obtained when using either of the self-adaptive crossover operations are equivalent or better than the results when using standard GP crossover. A postmortem analysis of the evolved values for the self-adaptive parameters suggests that certain heuristics commonly used in genetic programming may not be optimal.

5.1 Introduction: Adaptive and Self-Adaptive Evolutionary Computations

Evolutionary computations have proven to be powerful yet general methods for search and optimization. As with most search and optimization techniques, evolutionary computations include a number of operational parameters whose values significantly alter the behavior of the algorithm on a given problem, and usually in unpredictable ways. Typically, the optimal parameter values are not only problem dependent but population dependent as well.

Adaptive evolutionary computations are evolutionary computations that modify the values for certain operational parameters while solving a problem. The ultimate efficiency of an adaptive evolutionary computation follows from the method used to update the adaptive parameters. [Angeline 1995] identifies two distinct types of adaptive parameter update rules. *Absolute update rules* compute a predetermined function over a set of generations or populations and use the changes in this heuristic to determine when and how to modify the algorithm's adaptive parameters. *Empirical update rules* use the same variation and selection process evolving the problem solutions to also modify the adaptive parameters. When an adaptive evolutionary computation evolves the values for its adaptive parameters (i.e. uses an empirical update rule), then it is termed *self-adaptive*. Self-adaptive methods are more flexible and can more readily adjust to a broader range of adaptive situations than absolute update rules [Angeline 1995].

Adaptive evolutionary computations can be separated into three basic classes depending on the association between the adaptive parameters and the evolutionary process [Angeline 1995]. *Population-level* adaptations modify population-wide parameters, often with heuristics computed over the current or past populations. Such methods include updating the global frequency of operator application [Rechenberg 1973; Davis 1989], and dynamically adjusting the interpretation of the representation [Shaefer 1987; Schraudolph and Belew 1992; Whitley, Mathias and Fitzhorn 1991]. *Individual-level* adaptations associate parameters with each individual that determine how the algorithm manipulates the individual. Examples include evolving crossover positions in

genetic algorithms [Rosenberg 1967; Schaffer and Morishima 1987] and adapting the relative probability of mutating components of finite state machines in an evolutionary program [Fogel, Fogel and Angeline 1994]. *Component-level* adaptations associate adaptive parameters with each component of an evolving individual that determine how each component is modified during reproduction. These methods include evolving the variance of the mutational noise applied to each component of real-valued fixed-length vector representations [Schwefel 1981; Bäch and Schwefel 1993; Fogel et al. 1991; Fogel et al. 1992] and evolving the absolute probability of mutating particular components in finite state machines [Fogel, Fogel and Angeline 1994]. Typically, individual-level and component-level adaptive parameters are self-adaptive.

The current study is concerned with self-adaptive crossover operators for genetic programming. Some previous studies have investigated adaptive and self-adaptive representations in genetic programs, although not under this guise. The *adaptive representation genetic program* [Rosca and Ballard 1994] is a population-level adaptive genetic program that uses statistics gathered over all subtrees occurring in the population to determine more advantageous crossover points and preserve high-fitness subtrees. The *genetic library builder* (GLiB) [Angeline and Pollack 1992; Angeline and Pollack 1994] is an individual-level self-adaptive genetic program that co-evolves a hierarchical representation for each individual in the population. GLiB adapts the content, interface, and number of "modules" used in an individual. Valid and invalid positions for crossover, distinct for each individual, emerge naturally as a by-product of this process. *Automatically defined functions* (ADFs) [Koza 1994] is an individual-level self-adaptive genetic program where each individual adapts its definitions for a predetermined set of subroutines. [Koza and Andre 1996] extend this method to also allow the number and interface of an individual's subroutines to adapt as well. Elsewhere in this volume, [Teller 1996] investigates a self-adaptive crossover scheme for a variant of genetic programming and [Iba and de Garis 1996] describes an adaptive crossover operation that uses a variety of absolute update rules.

This chapter investigates two self-adaptive crossover operators for genetic programming inspired by the self-adaptive mutation operators for finite state machines recently described in [Fogel, Fogel and Angeline 1994]. The first, termed *selective self-adaptive crossover* (SSAC), adapts values that determine where crossover will occur in an individual. *Self-adaptive multi-crossover* (SAMC), the second operator investigated in the study, adapts both where and how many crossovers are performed to a tree. Experimental results demonstrate that both of these self-adaptive operators perform as well or better than standard genetic programming crossover for several problems. Before describing the self-adaptive crossover operations and experimental results, the next section provides detailed description of the nonstandard genetic program used in the study.

5.2 A Nonstandard Genetic Program

The genetic program used in this study has a number of deviations from the genetic program introduced in [Koza 1992]. Most of the modifications are standard in evolutionary programming [Fogel, Owens and Walsh 1966; Fogel 1995]. While the philosophical underpinnings of evolutionary programming discourages recombination of population members, relying instead on representation specific mutation operations, there is nothing that prevents using recombination as a form of mutation. The genetic program studied here is essentially a standard evolutionary program as described in [Fogel 1995] that uses tree-based crossover as one form of mutation. The remainder of this section provides details of the algorithm.

5.2.1 Parent Selection

The type of selection performed in this study is a form of tournament selection modified to minimize over-selection (i.e. multiple reproductions given to a single high-fitness parent) [Fogel 1995]. First, a tournament score is associated with each member of the population based on its fitness relative to other population members. The algorithm assigns an individual's tournament score by randomly selecting k additional population members and summing the number with worse fitness. The population is then sorted by the tournament score and the best 50% is reserved as parents for the next generation. As a result, any given population member can be selected only once as a parent for the next generation.

Unlike the typical tournament selection algorithm used in genetic algorithms, this scheme forces both an individual and its offspring to have above average fitness in order to increase the representation of the lineage in the population. It thus places additional selection pressure on *mutable* fit population members rather than just those with high fitness. While this difference is subtle, it may hold advantages in complex environments.

5.2.2 Offspring Reproduction

Creation of offspring occurs in three stages. First, each parent selected from the last population is copied to create its offspring. Each parent is copied exactly once for the next generation. Next, crossover is applied to all children in the population. When a crossover operation is executed, a second "child" tree is selected from the population that has yet to undergo crossover. Once crossover is complete, both of the resulting children are placed in the population. Finally, each child is mutated with a user defined probability. The number of mutations performed on the child is given by a Poisson random variable with a user specified rate resampled for each child. Given that k mutations are chosen to be executed on a child, k mutation operations are selected uniformly with replacement from the five

mutation operators described below. The chosen set of mutations are applied to the child in the order selected. Any mutation or crossover operation that results in a child with a number of nodes outside a user-defined range is returned to its original form. For the experiments below, a viable tree had more than three nodes and less than 50.

This algorithm employs five separate mutations for evolving trees: *grow*, *shrink*, *cycle*, *switch*, and a *numerical terminal mutation*. *Grow* randomly selects a leaf node of the tree and replaces it with a randomly generated tree. The depth for the generated tree is limited by a user-defined constant set to seven in the experiments below. The *shrink* mutation selects a random subtree in the program and replaces it with a randomly selected terminal. *Cycle* chooses a function symbol in the program and replaces it with another function symbol that requires the same number of parameters. Only the name of the function at a particular node in the tree is changed in this mutation; the children (i.e., parameters) of the node are unmodified. A *switch* mutation selects two sibling subtrees of the same parent node and switches their positions in the parent's function call. Finally, the *numerical terminal mutation* is defined for what [Koza 1992] called *ephemeral random constants* and here are called *numerical terminals*. This mutation operation selects a single real-valued numerical terminal in the tree and adds to it Gaussian noise with a particular variance. In this study, the variance of the Gaussian noise for this mutation was always set to 0.1. This mutation is especially important since small changes in a numerical constant within an evolving program will typically correspond to small changes in the behavior of the program. Therefore, even after crossover becomes ineffective due to the emergence of introns [Angeline 1994], manipulation of numerical constants may improve offspring.

5.2.3 Comparison with a Standard Genetic Program

While there are a number of operational differences between the algorithm used in this study and more traditional genetic programs, these differences should not affect the relative performance of the crossover operators investigated below. It is important however to verify that this algorithm is at least competitive with more traditional genetic programs otherwise what appears to be a distinction in the results may be an artifact should the performance of this algorithm be significantly worse.

In order to gauge the performance of the genetic program used in this study, a preliminary experiment was run for a single test function. The 6-multiplexor problem, as defined in [Koza 1992], is a suitable comparison problem. This experiment used a population size of 250 with 50% population replacement resulting in an effective population size of 125. Standard GP crossover was applied to each child with leaves selected for 50 percent of the recombinations. Each child then had a 0.1 probability of being mutated with the number of mutations performed given by a Poisson random

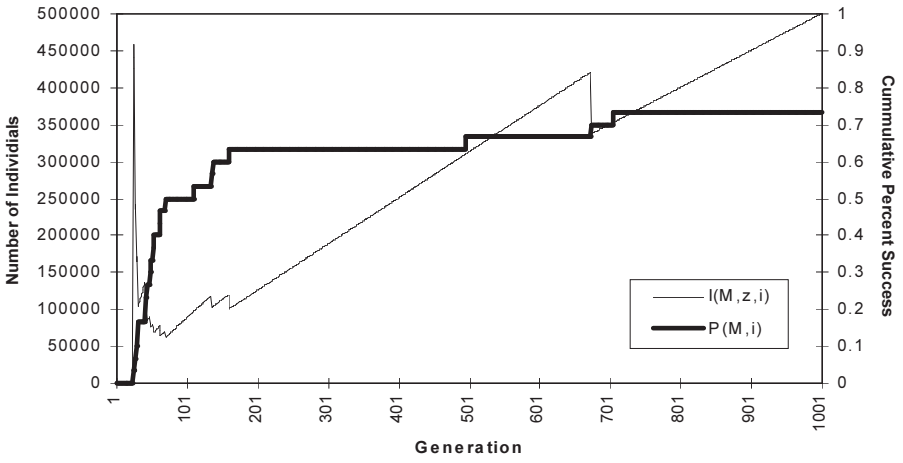


Figure 5.1
Graph showing the computational effort (*I(M, z, i)*, left axis) and cumulative probability of success (*P(M, i)*, right axis) for the described genetic program on the 6-multiplexor problem.

variable with a rate of four. Each trial was allowed to run for at most 1000 generations before being halted. A total of 30 separate trials were conducted.

[Koza 1992] defines the *computational effort* (*I*) required to evolve a genetic program as a function of population size (*M*), the generation (*i*) and a desired probability of success (*z*), typically set to 99%. Figure 5.1 shows the results of the experiment in terms of the cumulative probability of success of the algorithm (denoted *P(M,i)* in the graph) over the 30 runs and the calculated computational effort, both plotted against generation. As evidenced by the graph, for this experiment the minimum computational effort for $z = 0.99$ was 62,125 and occurred at generation 73. [Koza 1992] reports results for the 6-multiplexor problem for a number of population sizes. Table 5.1 shows the corresponding

Table 5.1
Standard GP Results for 6-Multiplexor Problem

Population Size	Minimum Computational Effort	Generation
500	245,000	69
1000	343,000	48
2000	200,000	49
4000	160,000	39

population size and computational effort reported in [Koza 1992] for the 6-multiplexor problem. The smallest computational effort reported in [Koza 1992] used a population

size of 4000 and required a minimum computational effort of 160,000, which is roughly three times the computational effort required by the genetic program used in this study. However, the statistics here are drawn from only 30 trials which is statistically insufficient to draw and conclusions with confidence. But the comparison does provide evidence that the described system is at least comparable in performance to a standard genetic program.

5.3 Two Self-Adaptive Crossover Operators

The two self-adaptive crossover operations investigated in this study are derived from the self-adaptive mutations defined in [Fogel, Fogel and Angeline 1994] for mutating finite state machines in an evolutionary program. These methods are similar to the self-adaptive mutations employed in the standard forms of evolution strategies and evolutionary programming used for optimizing fixed-length real-valued vectors.[1]

Selective self-adaptive crossover (SSAC) is an individual-level self-adaptation that adapts the relative probabilities of crossing a tree at particular subtrees. *Self-adaptive multi-crossover* (SAMC) is a component-level self-adaptation that adapts the absolute probability of crossover occurring at a each subtree in an individual. The following subsections describe these operators in detail.

5.3.1 Selective Self-Adaptive Crossover (SSAC)

Conceptually, selective self-adaptive crossover provides an adaptive mechanism for the selection of *where* a program should be crossed with another parent to produce a child. To implement SSAC, a second tree, called the *parameter tree*, is associated with each individual. An individual's parameter tree has the identical size and shape to its program tree but with real-valued numbers rather than functions or terminals at each position. Figure 5.2 shows such a composite individual.

The numbers in an individual's parameter tree determine the probability of crossing the tree at the corresponding position. The probability of performing a crossover at the *i*th subtree, p_i, when using SSAC is given by:

$$p_i = \frac{\sigma_i}{\sum_j \sigma_j} \tag{5.1}$$

1. The forms of self-adaptive crossover described here are similar to mechanisms studied in genetic algorithms by [Rosenberg 1967] and [Shafer and Morishima 1987] but are conceptual descendents of EP and ES.

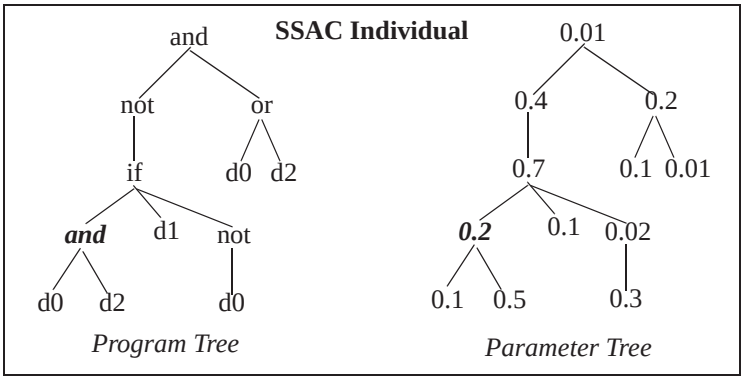


Figure 5.2
A self-adaptive individual composed of a program tree and a parameter tree. Roulette wheel selection over the individual's normalized parameter values is used to choose a crossover point. The emphasized parameter designates the position selected in this individual.

where σ_i is the parameter in the *i*th position of the individual's parameter tree and the sum is taken over all parameters in the individuals parameter tree. To select a crossover position in the individual, a roulette wheel selection [Holland 1992; Goldberg 1989] is performed over the components using the value given by Equation 5.1 as the component's relative selection probability. The equation gives a higher probability to those components with larger relative parameter values.

After the crossover points in both parents are selected, both the program trees and the parameter trees are crossed at the selected position. This ensures that both trees in the children maintain the same shape.

Once crossover has been performed on an individual, random noise is added to every parameter in the individual's parameter tree. The form of random noise added to a parameter σ_i is given by:

$$\sigma_i' = \sigma_i + \alpha \cdot \sigma_i \cdot N(0, 1) \tag{5.2}$$

where σ_i' is the new value of the parameter, N(0, 1) is a Gaussian random variable with mean 0 and variance 1 and α is a scaling constant set equal to 0.1 in the experiments described below. Parameter values that wander outside a user-defined range are reset to the closest value within the range. Initial parameter values for all randomly generated trees and subtrees (e.g., an initial population or a subtree created by a grow mutation) are set to the minimum parameter value. In the experiments below, the minimal value for SSAC was 0.0001 and the maximum value was 0.999.

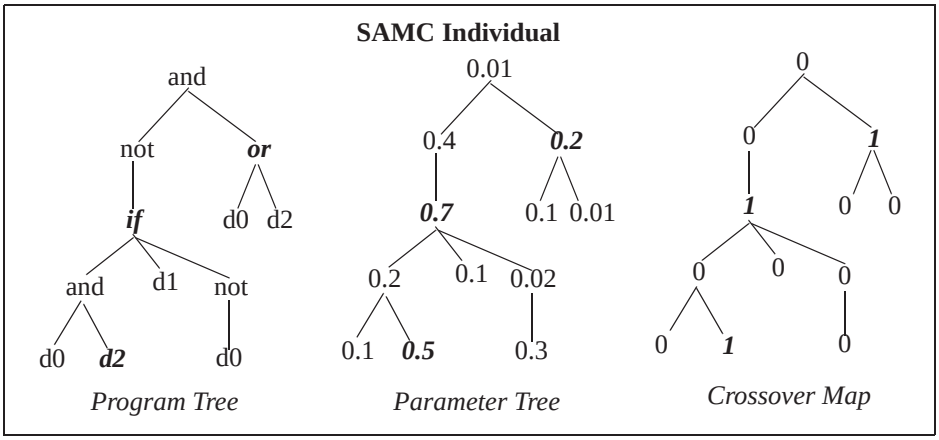


Figure 5.3
An SAMC individual composed of a program tree, a parameter tree and a crossover map. Parameter values are interpreted as the absolute probability of a crossover occurring at that position in the individual. A "*1*" in the crossover map designates where the probabilities passed the test in the individual. The above individual has three positions where crossover will be performed.

5.3.2 Self-Adaptive Multi-Crossover (SAMC)

The second self-adaptive crossover operator investigated in this study is similar to the first. It also associates a parameter tree with each individual and updates the values in the parameter tree using random noise as in Equation 5.2. Self-adaptive multi-crossover (SMAC) differs from SSAC in how it interprets the parameters. In SSAC, the probability of crossing over a particular position is a relative function of an individual's parameters. In SAMC, the parameter value in the parameter tree is the *absolute* probability of crossover being performed at that position in this individual during this reproduction. Consequently, SAMC permits the individual to determine both the position and number of crossovers applied to generate its offspring.

Self-adaptive multi-crossover proceeds as follows. First, every subtree in the individual is tested to determine if crossover should be performed at that position. For each subtree, a new uniform random number is rolled and checked against the subtree's parameter value. If the parameter value is less than the random number then the subtree is scheduled for crossover. Through this process, a third tree is created that records each position in the individual where crossover is to be performed. This is the individual's *crossover map*, shown in Figure 5.3. Like the parameter tree, the crossover map has the same size and shape as the individual's program tree.

Given two parents and their respective crossover maps, the actual crossover of subtrees proceeds iteratively. Crossover positions for both parents are selected at random from

those that are indicated by a "1" in their respective crossover maps. Selection of a position toggles the 1 to a 0 in both individual's crossover maps. Crossover is then performed between the parents at the selected respective positions in all three trees. That is, each actual crossover switches subtrees between the respective program trees, the parameter trees, and the crossover maps in the two individuals. The crossover map must also be crossed to ensure that the positions that were previously scheduled for crossing are maintained. Crossovers continue until one individual's crossover map indicates that no crossovers are left to be executed (i.e., it has no 1s at any position in its crossover map).

The dynamics of this operator are anything but simple. Its complexity arises from the fact that while genetic material mixes between the two individuals, the scheduled crossover maps also mix. Consequently, it is possible for two individuals that originally have several positions each scheduled for crossover actually execute only a single crossover. This occurs when after the first crossover all remaining scheduled crossover points are collected within a single individual. On the other hand, all points within an individual originally scheduled for crossover may not be paired with points in the other individual but may instead get crossed with other points within itself. Such a situation occurs when an original crossover map is separated between the two individuals and random selection of scheduled positions identifies positions that were originally within the same individual. At this point, it is unclear if the complex dynamics of this operator are a benefit or a liability, but they do pose some interesting potential crossover scenarios.

As in SSAC, an individual's parameter tree in SAMC is mutated by adding gaussian noise to each of the parameters. Parameter values are also restricted to a user-defined range. The minimum value for a parameter in the experiments below was 0.01 and the maximum was 0.999. The initial value for an entry in the parameter tree was 0.1.

5.3.3 Empirical Credit Assignment Applied to Adaptive Parameters

It is not entirely obvious how the empirical update rules used in the above self-adaptive operators will adapt suitable parameter values. Consider that the natural dynamics of an evolutionary computation encourages the promotion of those structures that more quickly lead to more fit individuals. If a modification is made to an adaptive parameter in a particular individual, then if that modification is comparatively beneficial (or neutral) to offspring development, it will persistent in the population along with the lineage of the individual. If it is detrimental then the individual's offspring will eventually be supplanted by those with adaptive parameters more beneficial to creating increasingly fit offspring [Angeline 1995]. Thus, when using such an update rule, no explicit form of credit assignment or evaluation is required since the dynamics of the evolutionary computation automatically performs an empirical form of credit assignment [Angeline 1993].

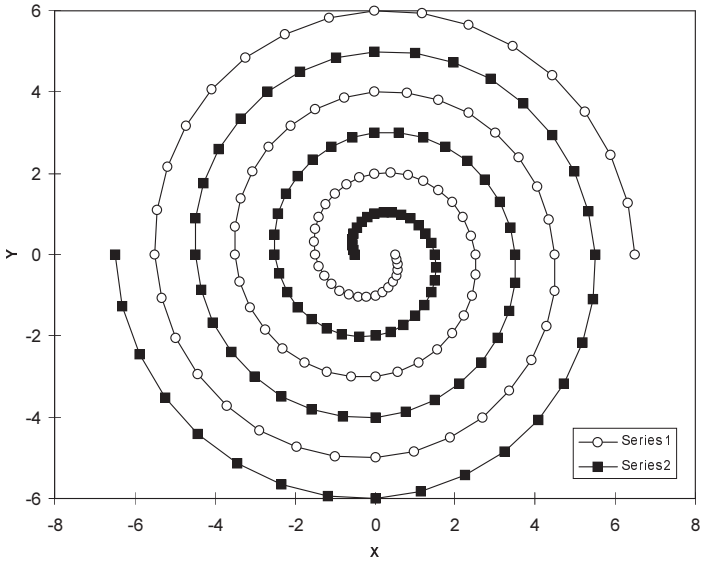


Figure 5.4
Standard data set for the interlocking spiral problem used in experiment.

5.4 Experiments

Three test problems were chosen in order to determine the relative performance of each of the crossover operations. Each problem had distinct characteristics with the set comprising a minimal but broad subset of the problem types investigated in genetic programming.

The first problem investigated was the boolean 6-multiplexer function (6-Mux) as originally defined in [Koza 1992]. The function set for this problem is {*not, and, or, if*} and the terminal set is {*a0, a1, d0, d1, d2, d3*} where *d0, d1, d2* and *d3* are the data inputs and *a0* and *a1* are the selector bits. The fitness of an individual was the number of correct responses for the 64 possible inputs.

The interlocking spirals problem was the second problem investigated. In this problem, a function must be learned that separates two classes of data that form interlocking spirals in the coordinate plane. Figure 5.4 shows the two data sets for the problem. [Koza 1992] also investigated this problem and the form of this experiment follows the experiment described there. The function set is defined to be {+, -, *, /, *iflte*, *sin*, *cos*} and the terminal set is defined as {*x, y, RNT*} where *x* and *y* are the coordinates of the point to be classified and RNT is a *random numerical terminal*, which is identical to an ephemeral random constant [Koza 1992] except it may be mutated with the numerical terminal mutation

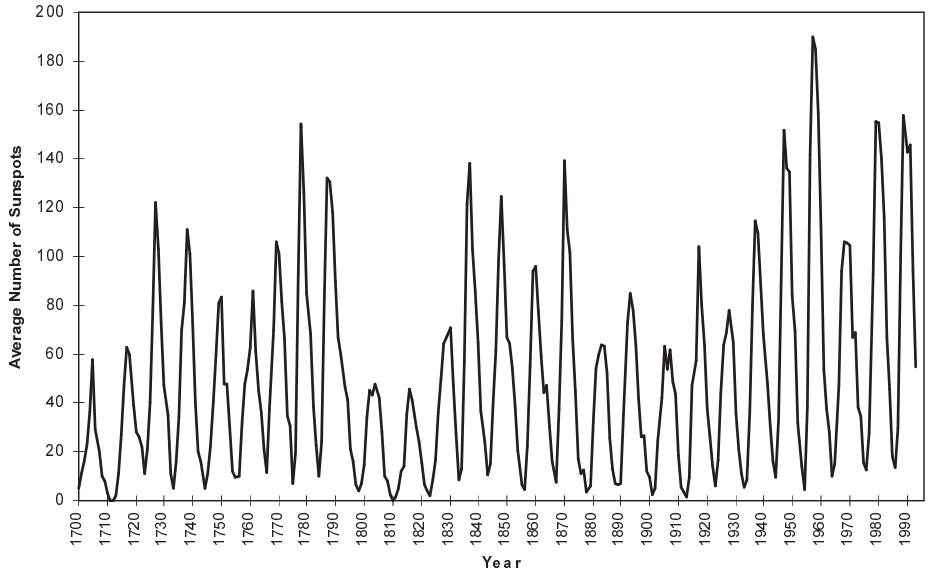


Figure 5.5
Graph showing the average number of sunspots per year (the Wolfe Sunspot data) from 1700 to 1993.

described above. The fitness of an individual for this problem is the number of points from the 194 points in the two data sets correctly classified.

The third problem used in the experiments is a time series modeling problem. Since the year 1700, the average number of sunspots observed each month have been recorded. The collection of these values are referred to as the Wolfe Sunspot Data and are available online.[1] Figure 5.5 shows the average number of sunspots per year for the time period from 1700 to 1993. This data is a naturally occurring chaotic time series which makes it more appealing than artificial chaotic time series, such as the Mackey-Glass equation for experiments, since the underlying generating model of the series is unknown. The object of this problem is to predict the number of sunspots that will be observed in the following year given a minimal number of previous data points. Following [Oakley 1994], a function set and terminal set were selected that fit a time series prediction problem. The functions used were {+, -, *, %, *sin*, *cos*}[2] and the terminal set was {*x1*, *x2*, *x4*, *x8*, *RNT*}

1. The World Wide Web address is http://www.ngdc.noaa.gov/stp/SOLAR/SSN/ssn.html.

2. The exclusion of a conditional function for this problem is intentional. As observed in [Angeline 1994], conditionals allow the development of diploidic structures in genetic programs that can also serve an intron-like function and potentially reduce the probability that certain mutations will alter the performance of the evolving tree. Not allowing a conditional forces every mutation to the evolving tree to have a resulting behavioral effect.

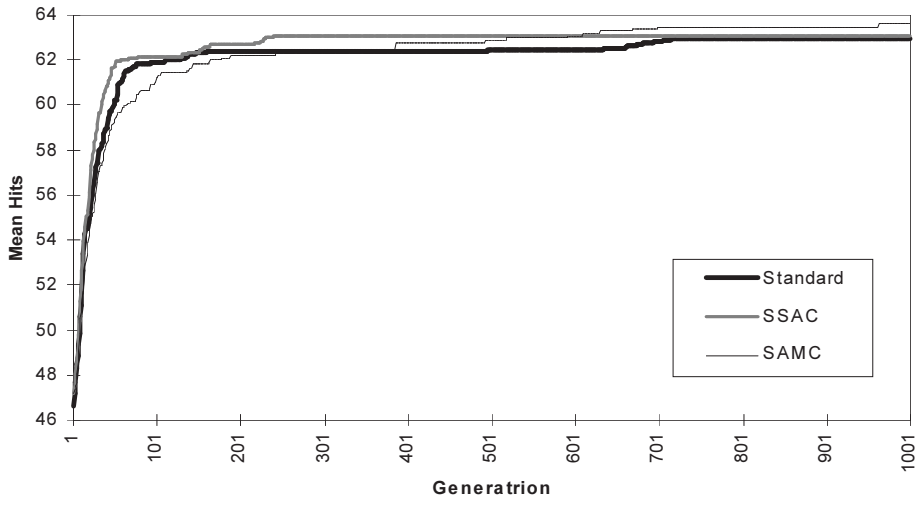
where *x1* is the number of sunspots observed in the year before the predicted year, *x2* is the number of sunspots observed two years before the predicted year and so on. The available terminals were selected to be powers of two for no particular reason other than it gave a good spread over the previous data. Other terminal sets may produce better results than those reported below. It should be noted that the data was not separated into training and test sets making this a time series modeling task rather than a time series prediction task. Given that the object is to minimize the prediction error of the evolved function, the fitness function is the sum of the squared prediction error for each of the data points (years 1708 to 1993). The actual fitness function used negates the error to make this a maximization problem.

Three experiments were run for each problem, one using standard GP crossover, one using SSAC and one using SAMC, all other parameters being the same. All experiments included 30 trials each using the parameters and methodology described above. In each of the experiments, the population size was 250 with 50 percent of the population replaced on each generation giving an effective population size of 125. When used, random numerical terminals were generated within the range [-1.0, 1.0] and were mutated with a variance of 0.1. The length of each trial in each experiment was limited to 1000 generations. The minimum and maximum number of nodes for any created individual were 3 and 50 respectively. Other specific parameters may be found above in the section that describes the component of the algorithm that relies on that parameter.

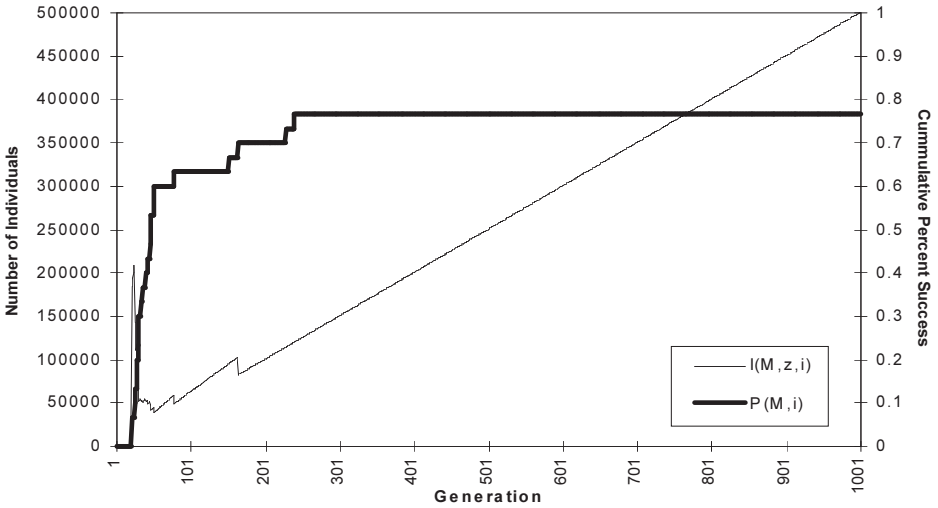
In order to permit a fair comparison between standard subtree crossover and the self-adaptive crossovers being investigated in this study, the probability of selecting a leaf as the crossover point in a parent was set to 0.5. The typical value for this parameter in a genetic program is 0.1, selecting leaves as crossover points 10 percent of the time, as defined in [Koza 1992]. Future work will access if this change is beneficial or detrimental.

5.5 Results

Figure 5.6a shows the average fitness results for each of the three different crossover operations on the 6-multiplexor problem. Standard crossover rises very quickly to roughly 62 hits on the function and then rises much more slowly for the rest of the run. SSAC rises just as quickly also topping out close to 62 before leveling off and staying consistently above standard crossover. SAMC rises much more slowly than both standard crossover and SSAC on average but also reaches a peak of close to 62 before leveling off around generation 200. After this point, it continually rises at a slower pace and eventually exceeds the performance of both standard crossover and SSAC.



(a)



(b)

Figure 5.6
Results for 6-mux problem. (a) The average score over the 30 trials for each of the three crossover operations plotted against the number of generations. (b) The cumulative probability of success and computational effort for SSAC plotted against the number of generations.

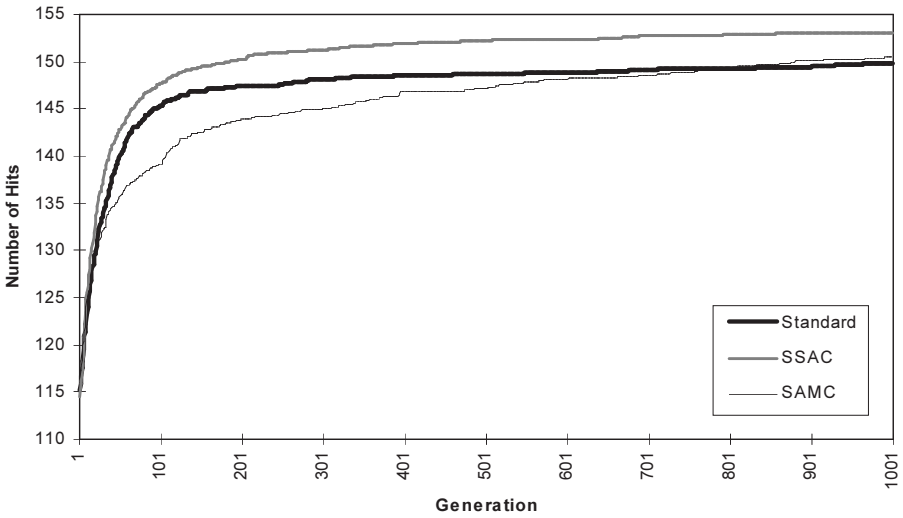


Figure 5.7
Results for Intertwined Spiral Problem showing the relative performance of the three crossover operators investigated in the study averaged over 30 trials.

For comparison to the results provided in section 5.2.3, Figure 5.6b shows the cumulative probability of solution and the computational effort for SSAC on the 6-mux problem. The computational effort for SSAC on the 6-mux problem is 39,000 and occurred at generation 54. As already noted, using standard GP crossover and the non-standard genetic program described in this study an effective population size of 125 yielded a three fold decrease in computational effort for this problem over the standard GP method reported in [Koza 1992]. Figure 5.6b shows a four fold decrease in the minimum computational effort for SSAC on the 6-mux problem when compared to the results shown in Table 5.1 as reported in [Koza 1992]. As mentioned above, these statistics can not be taken as reliable since they are determined from on 30 trials.

Figure 5.7 shows the average fitness results for each of the three crossover operations on the intertwined spirals problem. In this experiment, standard crossover rises fairly quickly to a value of about 143 and then begins to taper off until flattening out at a fitness of 147 around generation 300. SSAC again rises more quickly than standard crossover and levels out at about fitness 151 around generation 250. From that point it rises in performance at a slightly faster rate than standard crossover maintaining a 3.5% advantage on average for the remainder of the run. SAMC again rises more slowly than standard crossover reaching an average fitness of 147 around generation 600. SAMC eventually

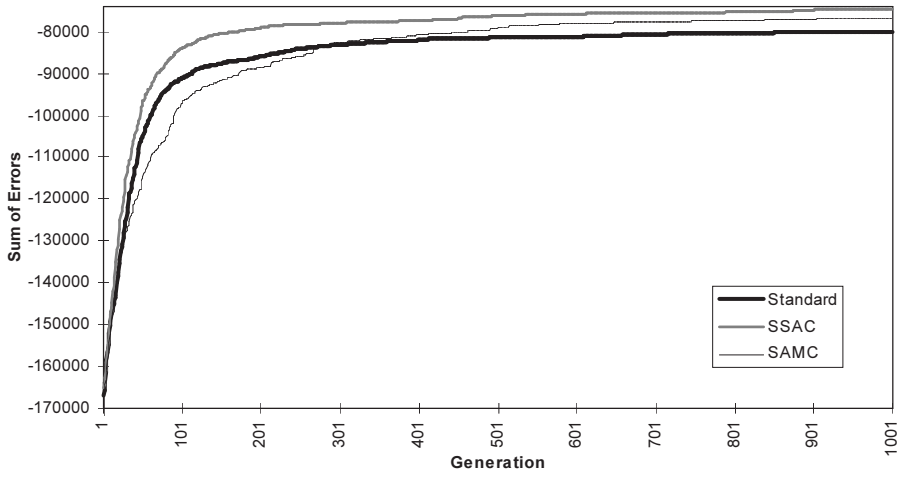


Figure 5.8
Results for Sunspot Problem showing the relative performance of the three crossover operators investigated in the study averaged over 30 trials.

equals the performance of standard crossover before the end of the run, although it clearly never attains the performance of SSAC on this problem.

Figure 5.8 shows the average fitness for the three crossover operations on the sunspot time series modeling problem. On this problem, standard crossover improves more slowly than the other problems and flattens out its improvement at a fitness of -88000 at around generation 200. The fitness of SSAC also rises more slowly for this problem, but is faster than standard crossover and levels out at a fitness of -80000 at generation 200. SAMC again rises more slowly than both standard crossover and SSAC but eventually surpasses the performance of standard crossover at generation 300 and approaches the performance of SSAC towards the end of the run.

Figure 5.9 shows the percent error between the actual data and the predictions of the evolved model for each data point in the sunspot series. Percent error was measured as the difference between the actual and predicted values divided by the maximum actual signal (190.2). The evolved function has captured the gross features of the sunspot data, rising and falling in the same period as the original data, but appears to have some difficulty modeling the extreme points of the quasi-periodic series. On average the prediction error was about 8% off from the actual predicted value. As mentioned above, a more carefully selected set of terminals may produce better results for this problem.

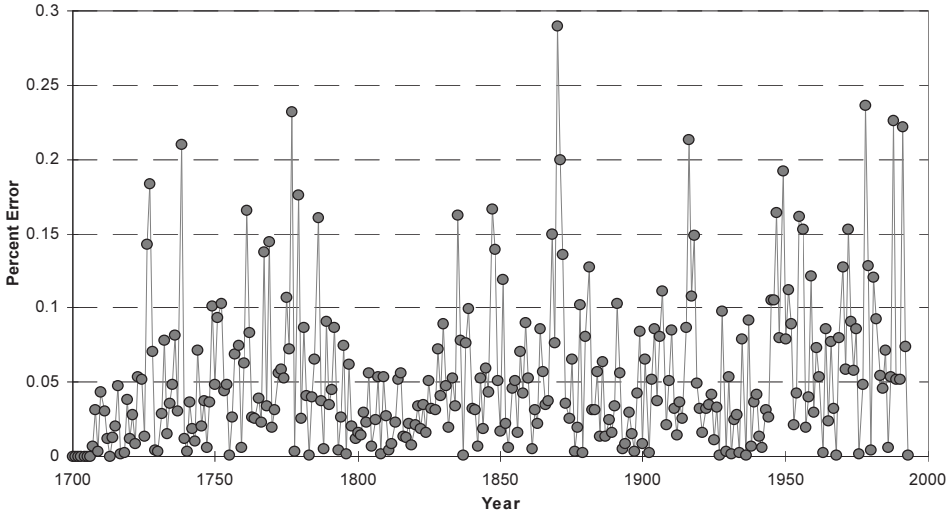


Figure 5.9
Graph showing the percent error of an evolved model for each data point in the sunspot data. Percent error is calculated as the difference between the predicted and actual signal divided by the maximum possible actual signal (190.2).

5.6 Postmortem Parameter Analysis

It is possible to use an adaptive parameter to validate heuristics for setting the same parameter in non-adaptive systems [Spears 1995]. Given the data for the various experiments described above, it is possible to ask if certain heuristics used in genetic programming are the natural values taken on during self-adaptation. For instance, if an analysis of the self-adapted parameters used in the SSAC experiments showed a tendency for trees to adapt parameters such that on average the probability of crossover at a leaf was close to 10% then this heuristic from [Koza 1992] for this parameter would be validated.

Given the nature of the self-adaptive crossover operations described above, there are two parametric assumptions in standard genetic programming that can be tested. First, because SSAC self-adapts relative selection probabilities for the components of an individual, the relative probability of selecting a leaf compared to a non-leaf when choosing sites for crossover can be experimentally determined. This is accomplished here after the fact by averaging the probability of selection over all of the leaves of the best of generation individual for all runs. A second implicit parameter that can be investigated is the number of crossovers per individual when evolving solutions. Genetic programming

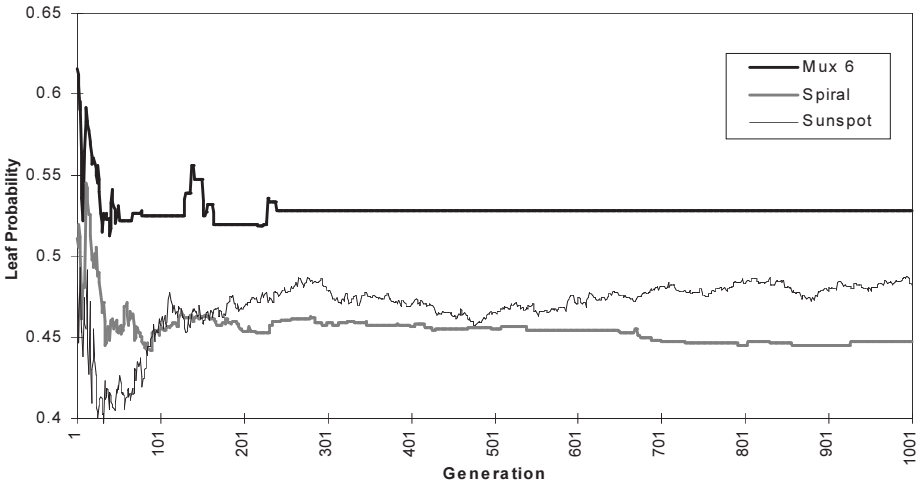


Figure 5.10
Average probability of selecting a leaf for the best of generation individuals when using Selective Self-Adaptive Crossover for the three problems studied.

has exclusively relied on a "single point" style of crossover even though traditional genetic algorithms now prefer more aggressive crossover operations. Given that multiple crossovers can occur between individuals when using SAMC, the average number of crossovers per individual can be tracked in these experiments and compared to the standard single subtree crossover assumption.

Figure 5.10 shows the average leaf frequency by generation for each of the three problems investigated. For the 6-mux problem, the probability of selecting a leaf when performing a crossover in the best of generation appears to fall on average around 53%. For both the interlocking spiral and sunspot problems the probability of selecting a leaf appears to gravitate more towards 45%.

The average number of crossovers executed per generation and per individual per generation when using SAMC is graphed in figure 5.11. For the 6-mux problem on average each individual received almost five crossovers per mating. For the spiral problem the number of crossovers reached almost eight crossings per mating. In the sunspot problem, the average number of crossovers per individual reached ten per mating.

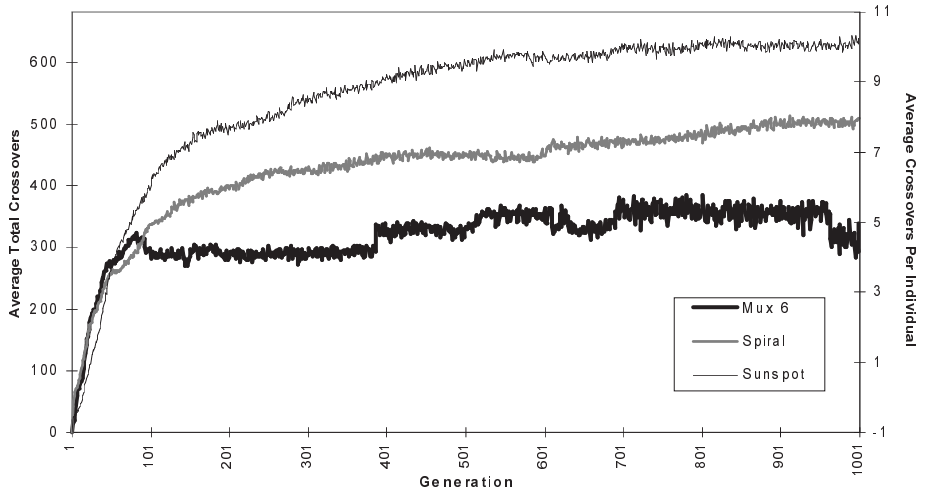


Figure 5.11
Average number of crossovers performed per generation (left axis) and per individual (right axis) when using Self-Adaptive Multi-Crossover for the three problems studied.

5.7 Discussion

From the experiments above it is clear that both of the self-adaptive methods are performing consistently as well or better than standard crossover for each of the three tested problems. On average, SAMC appears to take longer to level out than both standard crossover and SSAC. This is expected since poor coordination between the adaptive parameters in SAMC can lead to more catastrophic changes when used to guide crossover.

It is unclear from the experiments which of the two self-adaptive operators performs better. More often SSAC arrives at a level of performance first, although the 6-mux results show that this is not always the case. However, because SSAC is easier to implement and takes less time on average to execute than SAMC, SSAC seems the more prudent choice. These results agree with the comparative results for the non-recombinative versions of these operators investigated in [Fogel, Fogel and Angeline 1994] for finite state machines. This study clearly demonstrates, as does [Fogel, Fogel and Angeline 1994], that the update rules successfully applied to fixed-length real-valued feature vectors in EP and ES can also be successfully applied to variable length discrete representations.

The results of the postmortem analysis of values gravitated to by the self-adaptive processes suggest that the common values for both leaf selection probability during crossover and the number of crossovers per reproduction my not be optimal. First, the graphs of Figure 5.10 suggest raising the probability of selecting a leaf during crossover from the typical 10% may be beneficial. In addition, the consistently high values for the number of crossovers performed per individual appears to refute and notion that building blocks [Holland 1992] are preserved and propagated through the population in a genetic program. Such a large number of crossover operations is tantamount to macro-mutation and consequently would not respect building block boundaries. Runs that used a larger number of crossovers per individual consistently performed as well or better than those that used a single crossover per individual. This could be construed as evidence that preserving building blocks is either unimportant or possibly *counterproductive* to evolving genetic programs. Similar points has been made for crossover in genetic algorithms [Spears and De Jong 1991; Jones 1995]. However, it could also be the case that the values taken on by the self-adaptive parameters preserve building block boundaries in the evolving programs and effectively coevolve a macro-mutation that adoptively respects problem-specific building blocks. Resolving both this issue and the effect of larger leaf selection frequencies during crossover are questions for future work.

5.8 Conclusion

The ultimate effectiveness of any evolutionary computation is determined by the relationship between the shape of the landscape being searched and the operations that are used to generate new individuals. The rate of optimization or acquisition may be improved if the offspring distribution can be dynamically tuned to follow the contours of the landscape. This is the broad goal of all adaptive techniques, to dynamically acquire information about the response surface being traversed and adjust the parameters of the evolutionary computation accordingly. This study demonstrates that self-adaptation also improves the speed of traversing the fitness landscapes when evolving programs.

Acknowledgments

Thanks to Astro Teller, David Fogel, Bill Spears, and Byoung-Tak Zhang for reviewing intermediate versions of this chapter. Thanks also to the other members of the Advanced Technologies Department of Loral Federal Systems who continue to let me usurp all of the department's RS6000s during the off hours to run experiments.

References

Angeline, P. J. (1993). *Evolutionary Algorithms and Emergent Intelligence*. Doctoral Dissertation. The Ohio State University, Computer and Information Science Department.

Angeline, P. J. (1994). Genetic Programming and Emergent Intelligence. In *Advances in Genetic Programming*, K. Kinnear (ed.), Cambridge, MA: MIT Press, pp. 75-96.

Angeline, P. J. (1995). Adaptive and Self-Adaptive Evolutionary Computations. In *Computational Intelligence: A Dynamic System Perspective*, M. Palaniswami, Y. Attikiouzel, R. Marks, D. Fogel and T. Fukuda (eds.), Piscataway, NJ: IEEE Press, pp. 152-163.

Angeline, P. J. and Pollack, J. B. (1992). The Evolutionary Induction of Subroutines, In *Proceedings of the 14th Annual Conference of the Cognitive Science Society*, Hillsdale, NJ: Lawrence Erlbaum Associates, pp. 236-241.

Angeline, P. J. and Pollack, J. B. (1993). Evolutionary Module Acquisition. In *Proceedings of the Second Annual Conference on Evolutionary Programming*, D.B. Fogel and W. Atmar (eds.), La Jolla, CA: Evolutionary Programming Society, pp. 154-163.

Angeline, P. J. and Pollack, J. B. (1994). Coevolving High-Level Representations. In *Artificial Life III*, C.G. Langton (ed.), Addison-Wesley, Reading, MA, pp. 55-71.

Bäck, T. and Schwefel, H.-P. (1993). An Overview of Evolutionary Algorithms for Parameter Optimization, *Evolutionary Computation*, 1 (1), pp. 1-24.

Davis, L. (1989). Adapting Probabilities in Genetic Algorithms, *Proc. of the Second International Conference on Genetic Algorithms*, J.J. Grefenstette (ed.), Hillsdale, NJ: Lawrence Erlbaum, pp 61 - 69.

Fogel, D.B. (1995). *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, Piscataway, NJ: IEEE Press.

Fogel, D.B., Fogel, L.J. and Atmar, J.W. (1991). Meta-Evolutionary Programming. *Proc. of the 25th Asilomar Conference on Signals, Systems and Computers*, R.R. Chen (ed.), San Jose, CA: Maple Press, pp. 540-545.

Fogel, D.B., Fogel, L.J., Atmar, J.W. and Fogel, G.B. (1992). Hierarchic Methods of Evolutionary Programming, *Proceedings of the First Annual Conference on Evolutionary Programming*, D.B. Fogel and W. Atmar (eds.), La Jolla, CA: Evolutionary Programming Society, pp. 175-182.

Fogel, L.J., Owens, A.J. and Walsh, M.J. (1966). *Artificial Intelligence through Simulated Evolution*, New York: John Wiley.

Fogel, L.J., Fogel, D.B. and Angeline, P.J. l (1994). A Preliminary Investigation.on Extending Evolutionary Programming to Include Self-adaptation on Finite State Machines. *Informatica*, **18**, pp. 387-398.

Goldberg, D.E. (1989) Genetic Algorithms in Search, Optimization and Machine Learning, Reading,MA: Addison Wesley.

Holland, J.H. (1992). *Adaptation in Natural and Artificial Systems*, Ann Arbor, MI: University of Michigan Press.

Iba, H. and de Garis, H. (1996). Extending genetic programming with recombinative guidance. In *Advances in Genetic Programming: Volume 2*, P. Angeline and K. Kinnear (eds.), This volume.

Jones, T. (1995). Crossover, mutation, and population-based search. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, L. J. Eshelman (ed.), San Mateo, CA: Morgan Kaufmann, pp. 73-80.

Koza, J.R. (1992). *Genetic Programming: On the programming of computers by means of natural selection*. Cambridge, MA: MIT Press.

Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Subprograms*. Cambridge, MA: MIT Press.

Koza, J. R. and D. Andre (1996). Evolution of both the architecture and the sequence of work-performing steps of a computer program using genetic programming with architecture-altering operations. In *Advances in Genetic Programming: Volume 2*, P. Angeline and K. Kinnear (eds.), This volume.

Oakley, E. H. N., (1994). Two scientific applications of genetic programming: stack filters and non-linear equation fitting to chaotic data. In *Advances in Genetic Programming*, K. Kinnear (ed.), Cambridge, MA: MIT Press, pp. 369-390.

Rechenberg, I. (1973). *Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution*, Stuttgart: Frommann-Holzberg Verlag.

Rosenberg, R.S. (1967). *Simulation of genetic populations with biochemical properties*. Doctoral dissertation, University of Michigan, *Dissertation Abstracts International, 28 (7) 2732B*. (University Microfilms No. 67-17,836)

Schaffer, J.D. and Morishima, A. (1987). An Adaptive Crossover Distribution Mechanism for Genetic Algorithms. *Proc. of the Second International Conference on Genetic Algorithms*, J.J. Grefenstette (ed.), Hillsdale, NJ: Lawrence Erlbaum, pp. 36-40.

Schaudolph, N.N., and Belew, R.K. (1992). Dynamic parameter encoding for genetic algorithms. *Machine Learning*, **9** (1), pp 9-22.

Schwefel, H.-P. (1981). *Numerical Optimization of Computer Models*. Chichester, U.K: John Wiley.

Shaefer, C.G. (1987) The ARGOT System: Adaptive Representation Genetic Optimizing Technique. *Proc. of the Second International Conference on Genetic Algorithms*, J. Grefenstette (ed.), Hillsdale, NJ: Lawrence Erlbaum.

Spears, W. M. (1995). Adapting Crossover in Evolutionary Algorithms. In *Evolutionary Programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming*. J. McDonnell, R. Reynolds and D. Fogel (eds.), Cambridge, MA: MIT Press, pp. 367-384.

Spears, W. M. and De Jong, K. A. (1991). An analysis of multi-point crossover. In *Foundations of Genetic Algorithms*. G. J. Rawlins (ed), San Mateo, CA: Morgan Kaufmann Publishers, pp. 301-315.

Teller, E. (1996). Evolving programmers: The co-evolution of intelligent recombination operators. In *Advances in Genetic Programming: Volume 2*, P. Angeline and K. Kinnear (eds.), This volume.

Whitley, D., Mathias, K. and Fitzhorn, P. (1991) Delta coding: an iterative search strategy for genetic algorithms. *Proc. of the Fourth International Conference on Genetic Algorithms*, R. K. Belew and L. B. Booker (eds.), San Mateo, CA: Morgan Kaufmann, pp 77-84.