NASA-CR-194277

NAG9-635

Adaptively Resizing Populations: Algorithm, Analysis, and First Results

Robert E. Smith
Department of Engineering Science
and Mechanics
The University of Alabama
Box 870278
Tuscaloosa, Alabama 35487
phone: (205) 348-1618
fax: (205) 348-7240
email: rob@comec4.mh.ua.edu

Ellen Smuda
Department of Aerospace Engineering
The University of Alabama
Tuscaloosa, Alabama 35487

IN-61-CR
185348
57P

September 13, 1993

Abstract

Deciding on an appropriate population size for a given GA application can often be critical to the algorithm's success. Too small, and the GA can fall victim to sampling error, affecting the efficacy of its search. Too large, and the GA wastes computational resources. Although advice exists for sizing GA populations, much of this advice involves theoretical aspects that are not accessible to the novice user. This paper suggests an algorithm for adaptively resizing GA populations. This algorithm is based on recent theoretical developments that relate population size to schema fitness variance. The suggested algorithm is developed theoretically, and simulated with expected value equations. The algorithm is then tested on a problem where population sizing can mislead the GA. The work presented suggests that the population sizing algorithm may be a viable way to eliminate the population sizing decision from the application of GAs.

(NASA-CR-194277) ADAPTIVELY RESIZING POPULATIONS: ALGORITHM, ANALYSIS, AND FIRST RESULTS (Alabama Univ.) 57 p

N94-12808

Unclas

G3/61 0185348

1 Introduction

A novice GA user who sets out to apply a GA to an optimization problem faces a series of decisions. First, the user discovers that the problem must be encoded into a GA amenable representation. Next, the user must choose selection, recombination and mutation operators. Finally, the user must choose a set of GA parameters, typically a crossover probability (p_c) , mutation probability (p_m), and population size (N).

For the purpose of this discussion, one can make the assumption that, for a given set of parameters, a GA with a given set of operators is sufficiently robust to cope with a variety of problem encodings. This, of course, ignores the possible dilemma of deception in certain encodings, the relative merits of various encoding alphabets, and the different effects of various operators. However, GAs are generally robust, and this assumption allows us to focus on tunable aspects of a given GA, rather than the wide variations of GA implimentations.

Given these assumptions, the user's key decisions are those parameter settings. For setting p_c and p_m, some heuristic guidance exists. Crossover probabilities between 0.6 and 0.9 are recommended. Mutation rates should be between $1/N$, and $1/\ell$ (where ℓ is the encoding length). In practice, GAs are typically robust enough to cope with some variation in the settings of these two parameters. That is, experience shows that, within the bounds of these heuristics, variations in these parameters result in relatively small changes in GA performance. However, variations in population size can have substantial effects on GA performance on many problems. Too small a population, and the GA can fall prey to sampling error. Too large a population, and one wastes computational effort on extra fitness evaluations.

Some advice exists for sizing populations. In early papers on this topic, Goldberg develops formulae and indicative plots for population sizing (1985; 1989b). These results are based on some limiting assumptions about the necessary levels of schemata sampling in the GA. A more recent paper (Goldberg, Deb, & Clark, 1991), presents a formula for sizing populations that is directly based on schemata fitness variance. This formula directly indicates the necessary schema sampling for given schema statistics.

These formulae are useful, but they do present a difficulty for the novice. Without an understanding of the GA theoretic concepts of schemata and their statistics, the developments are inaccessible. Therefore, from the novice user's point-of-view, the GA may not seem robust at all. Novice users may comment, "I want to use the GA, but I don't want to have to understand it." Thus, what appears to be a robust algorithm to the experienced GA user may not be robust at all to the world at large.

One might comment that all optimization schemes suffer from this difficulty. To use them adequately in a broad range of problems, the user must understand some of the interactions of the algorithm's parameters, and thus, must understand the algorithm to some extent. However, parameters used in traditional search algorithms are often easy for a novice user to access and understand. These parameters are typically a desired accuracy for the search process or its end result, and a constraint on the computational resources (time and memory) that the user is willing to expend. In other words, the user need only answer, "How good do I want my answer to be, how long do I want to spend, and how much memory do I have in which to work?" The GA suffers from the difficulty of having parameters (like population size) that are much more difficult to relate to the user's needs.

This paper is the first outcome of a project to develop a GA whose parameters are more user accessible. Currently, the project's efforts are concentrated on the population sizing decision. This seems a logical first step, since population sizing is a decision the user must make in any GA, and it is a decision on which the novice has little accessible advice. The desired end result of this project is the development of a GA where the user need only specify a desired accuracy parameter, a computational time constraint, and a memory constraint. When the desired GA is developed, it will adaptively resize the population in an effort to meet the user's requirements. The remainder of this paper will present a suggested technique for dynamically resizing populations given these parameters. The technique is first developed theoretically, then simulated in expectation, then tested on a problem where population sizing is critical to GA performance.

2 Algorithm Development

To design an algorithm for adaptively resizing populations, one must form a basis for resizing, and design a procedure that exploits that basis. The following sections undertake that task.

2.1 Population Sizing Theory

The technique presented here is based on recent suggestions on population sizing from the literature (Goldberg et al., 1991). Thus, a brief summary of these developments is provided.

Theory suggests that GAs search by implicitly evaluating the mean fitness of various schemata based on a series of population samples, and then recombining highly fit schemata. Since the schemata average fitness values are based on samples, they typically have a non-zero variance. Consider the competing schemata:

$$H_1 = * * * * 1 1 0 * * 0$$
$$H_2 = * * * * 0 1 0 * * 0$$

Assuming a deterministic fitness function, variance of average fitness values of these schemata exists due to the various combinations of bits that can be placed in the don't-care (*) positions. This variance has been called *collateral noise* (Goldberg & Rudnick, 1991). Let $f(H_1)$ and $f(H_2)$ represent the average fitness values for schemata H_1 and H_2, respectively, taken over all possible strings in each schema. Also let σ_1^2 and σ_2^2 represent the variances taken over all corresponding schemata members.

The GA does not make its selection decisions based on $f(H_1)$ and $f(H_2)$. Instead, it makes these decisions based on a sample of a given size for each schema. Let's call these observed fitness values $f_o(H_1)$ and $f_o(H_2)$. Observed fitness values are a function of $n(H_1)$ and $n(H_2)$, the numbers of copies of schemata H_1 and H_2 in the population, respectively. Given moderate sample sizes, the central limit theorem tells us that $f_o$values will be distributed normally, with mean $f(H)$ and variance $\sigma^2/n(H)$.

Due to the sampling process and the related variance, it is possible for the GA to err in its selection decisions on schema H_1 versus H_2. In other words, if one assumes $f(H_1) > f(H_2)$, there is a probability that $f_o(H_1) < f_o(H_2)$. If such mean fitness values are observed the GA will incorrectly select H_2 over H_1. Given the $f(H)$ and σ^2 values, one can calculate the probability of such an error based on the convolution of the two normals. This convolution is itself normal with mean $f(H_1) - f(H_2)$ and variance $(\sigma_1^2/n(H_1)) + (\sigma_2^2/n(H_2))$. Thus, the probability that $f_o(H_1) < f_o(H_2)$ is α, where

$$z^2(\alpha) = \frac{(f(H_1) - f(H_2))^2}{(\sigma_1^2/n(H_1)) + (\sigma_2^2/n(H_2))},$$

and $z(\alpha)$ is the ordinate of the unit, one-sided, normal deviate. Note that $z(\alpha)$ is, in effect, a signal-to-noise ratio, where the signal in question is a selective advantage, and the noise is the collateral noise for the given schemata competition.

For a given z, α can be found in standard tables, or approximated. For values of $|z| > 2$ (two standard deviations from the mean), one can use the Gaussian tail approximation:

$$\alpha = \frac{\exp(-z^2/2)}{\left(z\sqrt{2\pi}\right)}.$$

For values of $|z| \leq 2$, one can use the sigmoidal approximation suggested by Valenzuela (1989):

$$\alpha = \frac{1}{1 + \exp(-1.6z)}.$$

Given this calculation, one can match a desired maximum level of error in selection to a desired population size. This is accomplished by setting $n(H_1)$ and $n(H_2)$ such that the error probability is lowered below the desired level. In effect, raising either of the $n(H)$ values "sharpens" (lowers the variance of) the associated normal distribution, thus reducing the convolution of the two distributions.

Goldberg et al. (1991) suggest that if the largest value of $2^{o(H)}\sigma^2/|f(H_1) - f(H_2)|$ is known for competitive schemata of order $o(H)$, one can conservatively size the population by assuming the $n(H)$ values are the expected values for a random population of size N. This gives the sizing formula:

$$N = z^2(\alpha)2^{o(H)}\frac{\sigma_1^2 + \sigma_2^2}{(f(H_1) - f(H_2))^2}$$

Goldberg et al. (1991) goes on to consider particular versions of this formula for various problem configurations and levels of deception. These developments are not considered here, but they may have implications for later developments of the suggested adaptive resizing algorithm.

The formula presented above is a thorough compilation of the concepts of schemata variance and its relationship to population sizing. However, it does present some difficulties from the point of view of robustness for the novice. The values and ranges of $f(H)$ are not known beforehand for any schemata, although these values are implicitly estimated in the GA process. Moreover, the values of σ^2 are neither known nor estimated in the usual GA process.

In addition, the formula does not consider the relative importance of schemata competitions. If two competing schemata have fitness values that are nearly equal, the overlap in the distributions will be great, thus suggesting a large population. However, if the fitness values of these schemata are very nearly equal, their importance to the overall search may be minimal, thus precluding the need for a large population on their account. To compensate for this effect, one could consider the absolute expected selection loss, $L(H_1, H_2)$, due to an error in selection, as opposed to the probability of such an error:

$$L(H_1, H_2) = |f(H_1) - f(H_2)|\alpha(H_1, H_2).$$

The technique suggested here will attempt to dynamically size a population based on matching the selection loss L to a desired target for this loss L_t provided by the user. To do so, one must estimate schemata variance, a topic taken up in the next section.

2.2 Estimating Variance and Selection Loss

The standard GA estimates schema mean fitness values through repeated, probabilistic selection and recombination of strings, based on each single string's fitness value. The fitness of a single string that belongs to a schema is clearly the smallest sample possible for evaluating a meaningful average fitness value for that schema. Clearly, this minimal sample cannot be relied upon to deliver an adequate average fitness. However, the GA uses a population of strings to implicitly deliver the schema average fitness for a larger sample, while never explicitly evaluating this sample's average. Also, under many selection schemes, the GA makes selection decisions over competing schemata in small increments (small changes in population proportions), thus distributing the evaluation of schemata averages fitness values over time. This spatially and temporally distributed evaluation is key to a GA's *implicit parallelism* (Holland, 1975), as well as its easy *explicit* use on parallel computers. The distributed approach also maintains the GA's naturally-inspired character. In developing a method for estimating variance, it would be desirable to maintain a similar approach.

Clearly, one can't evaluate an estimate of schema variance based on a single member of that schema. To evaluate a variance, one must consider at least two points. Consider the fitness variance of a pair of strings, in particular, the variance of mates. What schemata variances does one learn about by observing the variance in fitness between two strings? In a deterministic problem, any variance is caused by the bits where the two strings differ. Thus, the variance in fitness of two strings gives an estimate of the fitness variance of the schemata that the strings have in common. For example, consider the following strings and fitness values:

	string	fitness
S_1	0 1 0 1 0 0 1 1 1 0 1 0	100
S_2	1 0 0 1 0 1 1 0 0 1 0 1	50

The strings S_1 and S_2 share schema

$$H_{1,2} = **010*1*****,$$

and indicate that an estimate of its average fitness is $f(H_{1,2}) = (f(S_1) + f(S_2))/2 = 75$. The indicated variance is given by

$$\sigma^2_{1,2} = \frac{(f(S_1) - f(H_{1,2}))^2 + (f(S_2) - f(H_{1,2}))^2}{2} = \frac{(f(S_1) - f(S_2))^2}{4} = 625$$

Although this is a very crude estimate of schema fitness variance, it is no more crude than the estimate of schema average fitness implicit in the usual GA.

Note at this point that almost all children of a pair of strings will share the same common schemata as their parents. Only children who mutate in the defined bits of the shared schema will fail to share this schema with their parents. This fact will prove useful in the algorithm suggested here.

To size populations based on the developments of Goldberg et al. (1991), one must consider competing schemata as a basis for calculating α. To do so, let's consider a competition of pairs of schemata. Recall that each pair can be used to gain variance information on their common schemata. When one compares two such pairs, one obtains information about fitness differences that result from any bits that are common within each pair, and are in common bit positions in the competing pairs. These are the competitive schemata indicated by a competition of pairs. For instance, consider the strings S_1 and S_2 listed above, competing as a mating pair against the following strings, arranged as another mating pair:

	string	fitness
S_3	1 1 1 0 0 0 1 1 1 1 1 0	152
S_4	1 0 0 0 0 1 1 0 0 0 0 1	68

Strings S_3 and S_4 share the following schema:

$$H_{3,4} = 1**00*1*****,$$

and indicate that $f(H_{3,4}) = 110$ and $\sigma^2_{3,4} = 7056$. The schemata $H_{1,2}$ and $H_{3,4}$ share defined positions 4, 5, and 7. Thus, the competition of string pair S_1 and S_2 against string pair S_3 and S_4 gives information about the following two competitive schemata

$$H'_{1,2} = ***10*1*****$$

$$H'_{3,4} = * * *00 * 1 * * * **$$

both of which are in partition

$$J_{1,2,3,4} = * * *\texttt{ff} * \texttt{f} * * * **.$$

Thus, from this information, one can begin to obtain an estimate of α, and thus L, for the schemata indicated by this competition of pairs. To complete this calculation, one must determine the numbers of copies in the current populations for each of the two competing schemata under consideration, $n(H'_{1,2})$ and $n(H'_{3,4})$. Discussion of methods for obtaining these counts and their computational expense will be deferred to the conclusions of this paper. For the time being, assume that such a count can be obtained at reasonable computational expense.

Given these factors, one can use a competition of two pairs to calculate an estimated selection loss:

$$\hat{L}(H_{1,2}, H_{3,4}) = |f(H_{1,2}) - f(H_{3,4})|\alpha(H_{1,2}, H_{3,4}),$$

where $\alpha(H_{1,2}, H_{3,4})$ is calculated using either the sigmoidal or Gaussian tail approximations, and an estimated z value, given by

$$z = \frac{(f(H_{1,2}) - f(H_{3,4}))}{\sqrt{(\sigma^2_{1,2}/n(H'_{1,2})) + (\sigma^2_{3,4}/n(H'_{3,4}))}}.$$

The following section will show how this estimated selection loss can be used to adaptively resize populations.

2.3 An Adaptive Population Sizing Algorithm

Consider the following steps to adaptively resize populations. Assume that the user has supplied a desired target value for acceptable selection loss, L_t. Start with a small, randomly initialized population, and repeat the following steps until population size and distribution stabilizes:

1. Randomly put population members together in mating pairs.
2. Determine the shared schemata in each mating pair.
3. Put together competitions of mating pairs.
4. Determine the competitive schemata for each mating-pair competition.
5. Obtain a count of each competitive schema.
6. Calculate $\hat{L}$ for each mating-pair competition.
7. Use $\hat{L}/L_t$ as a basis for assigning more (or fewer) population members (see below for details).
8. Construct new population members by crossover and mutation of selected mating pairs.

As was previously noted, almost all of the children of a given mating pair will have the common schema from that pair. Thus, one can expect the schema survival relationships of the typical GA will extend to the common schemata in the suggested algorithm.

It is desired that the suggested process converge to a population where expected selection losses match the target value L_t. Consider the following use of $\hat{L}/L_t$ values. Each $\hat{L}/L_t$ value is mapped to an expected one-step growth rate for the associated mating-pair competition, via a sigmoid function:

$$G = (1-\gamma) + \frac{2\gamma}{1+\exp\left(-\beta\left(\frac{\hat{L}}{L_t}-1\right)\right)}$$

where γ is a desired, maximum, expected percentage increase (or decrease) per generation, and β is a parameter. This growth factor can be mapped to any one of a variety of selection schemes (Goldberg & Deb, 1990).

Of course, this represents the introduction of two new parameters, which may defeat the original purpose of this work: to increase ease of GA use. However, if robust heuristics can be determined for these parameters, ease of use will not be affected. In the simulations presented below, $\beta = 1$ and $\gamma = 0.9$ work well.

The G factor listed above is associated with all four strings in the mating pair competition. One should ask how this factor should be divided between the two competing mating pairs. $\hat{L}$ gives us no information on which schemata's numbers should be increased in the given competitive partition. For lack of a better strategy, one can inject a degree of selection by dividing the change in number of the schemata under consideration based on relative fitness. For instance, if $G_{1,2,3,4}$ reflects a competition of mating pair S_1, S_2 against S_3, S_4, then

$$G_{1,2} = \left(1 - \frac{f(H_{1,2})}{f(H_{1,2}) + f(H_{3,4})}\right) + \left(G_{1,2,3,4}\frac{f(H_{1,2})}{f(H_{1,2}) + f(H_{3,4})}\right)$$

and

$$G_{3,4} = \left(1 - \frac{f(H_{3,4})}{f(H_{1,2}) + f(H_{3,4})}\right) + \left(G_{1,2,3,4}\frac{f(H_{3,4})}{f(H_{1,2}) + f(H_{3,4})}\right)$$

where $G_{1,2}$ and $G_{3,4}$ are respective growth factors for the two mating pairs.

Given a one-step growth rate for each mating pair, one can use a selection scheme to assign new copies. The $\hat{L}/L_t$ values for each mating pair competition will effect population growth through this selective process. When $\hat{L}/L_t$ is greater than one (for a given mating pair competition), population size will increase (with respect to that competition). When $\hat{L}/L_t$ is less than one, population size will decrease. When $\hat{L}/L_t$ values are equal to one, population growth will cease. When population growth ceases, or population size becomes relatively stable, the regular genetic algorithm can begin to select purely based on fitness. This divides the GA selective process into two distinct phases. First is a selection based on variance, with variable population size. Next is selection based on fitness, with a fixed population size. This is a clear division of exploration and exploitation phases. Later revisions of the algorithm will require a gradual transition between these phases. Viewed in this light, the suggested algorithm bears some resemblance to the exploration-exploitation strategies for reinforcement learning problems suggested by Thrun (1992).

When population size adjustment ceases in the suggested scheme, the proportions of various schemata will not be evenly distributed in any given partition. However, the fitness biasing used to divide growth factors should help to insure that high fitness schemata in any given partition will be likely to have higher numbers of copies.

3 Analytical Simulation

One can simulate the suggested algorithm's behavior by considering a single partition of competitive schemata. Before doing so, it is useful to illustrate the sampling error effects that can occur due to finite population size. This can be accomplished in simulation by considering eight competing schemata in an order three partition. Consider $f(H)$ and σ values for the following eight schemata:

schema	$f(H)$	σ
1	1.00	6.4
2	0.90	13.2
3	0.80	3.2
4	0.70	10.0
5	0.65	16.4
6	0.95	2.0
7	0.80	20.0
8	0.72	0.4

These values were selected to show a variety of combinations of fitness and variance for simulating the population resizing scheme.

Iterating proportion equations (Goldberg, 1987) will show the expected performance of fitness proportionate selection on these schemata:

$$P_i^{t+1} = \frac{f_i P_i^t}{\sum_j P_j^t f_j}$$

for all i and j in the competitive partition. However, this expected value model does not consider variance effects. As an abstraction of actual fitness proportionate selection under sampling noise, consider iteration of a noisy proportion equation:

$$P_i^{t+1} = \frac{f_i' P_i^t}{\sum_j P_j^t f_j'}$$

where the f' values are given by the f values with zero-mean Gaussian noise added. The variance of this noise is given by the previously listed σ values, divided by appropriate counts of the corresponding schema. To simulate a count of each schema, assume a population size N, and let $n(H_i) = P_i N$. The proportion values will also be rounded to reflect the simulated finite population size.

Figure 1 shows typical results of iterating the noisy proportion equations with $N = 1024$. The highest fitness schema quickly takes over the population. This result occurs consistently with this population size. However, with $N = 64$, the results are quite different. Figure 2 shows a typical run. As shown in this example, the population often diverges from the highest fitness schema. The propensity of the proportion equations to diverge increases with decreasing population size. Figure 3 shows a convergence histogram for 25 runs for $N = 64$. This histogram shows the proportion of each schema at the end of 100 generations, averaged over the 25 runs. In all 25 runs, some schema overtakes nearly the entire population (well over 98%). The histogram indicates the schema that overtakes the population is often not the most fit schema. Figures 4 and 5 show similar histograms for $N = 128$ and $N = 32$, respectively. Clearly, sampling error can cause selection error, and associated divergence from the highest fitness schemata in a partition. Since selection error is a function of population size, the population resizing algorithm should reduce selection error to a point where correct convergence occurs.

The f and σ values listed above are used in an expected value model to simulate the action of the suggested population resizing algorithm. Note that this simulation considers the probability of any two schemata i and j coming into competition as $P_i P_j$.

Given a starting population size of $N = 64$, with equal initial numbers of each schema in the competitive partition, Figure 6 shows the results of simulating the population resizing scheme. The final population size from this simulation is 2048.

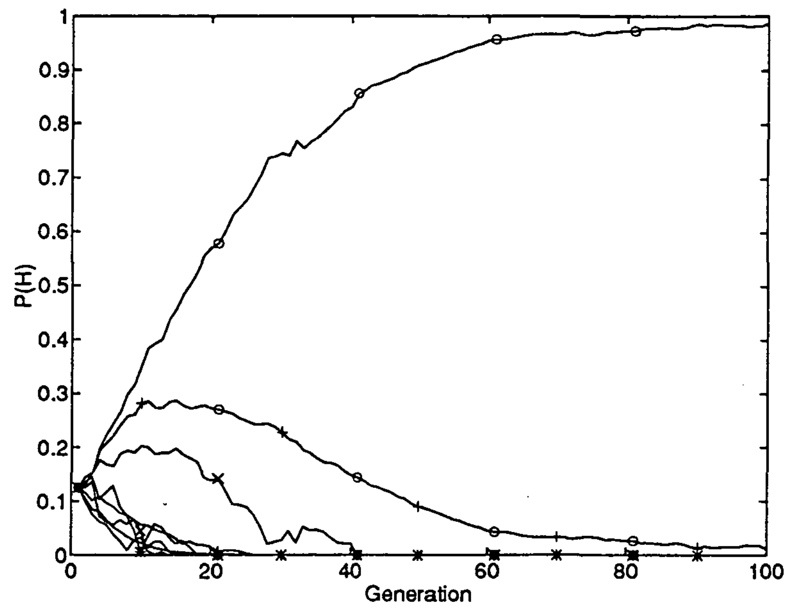

Figure 1: Simulated evolution of schema proportions under selection with noisy fitness values ($N = 1024$). Symbols are as follows: -o=H_1, -x=H_2, - +=H_3, -*=H_4, o-x=H_5, o-+=H_6, o-x=H_7, —=H_8.

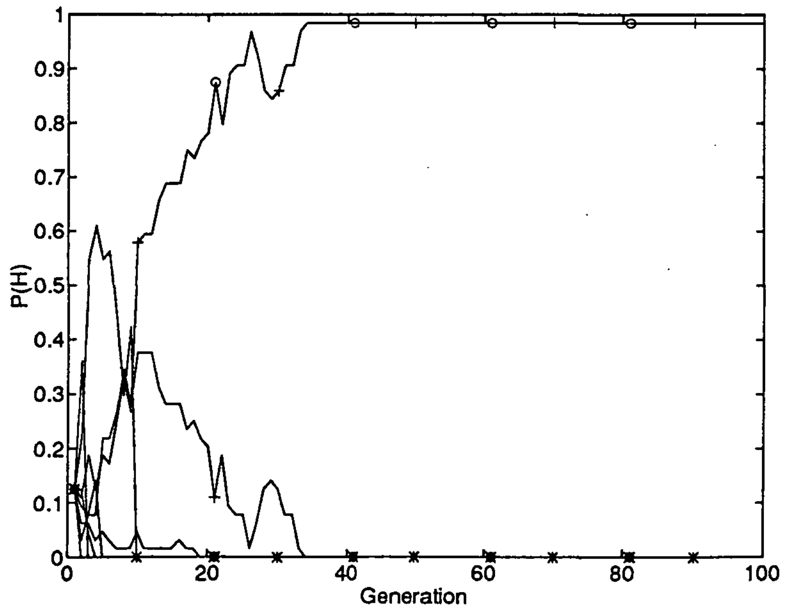

Figure 2: Simulated evolution of schema proportions under selection with noisy fitness values ($N = 64$). Symbols are as in the previous figure.

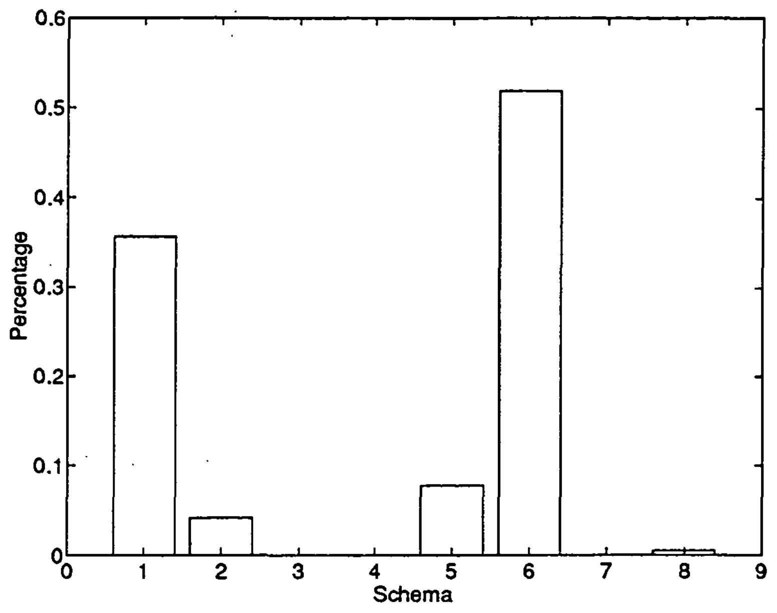

Figure 3: Histogram indicating the final schema converged to in 25 simulations under selection with noisy fitness values ($N = 64$).

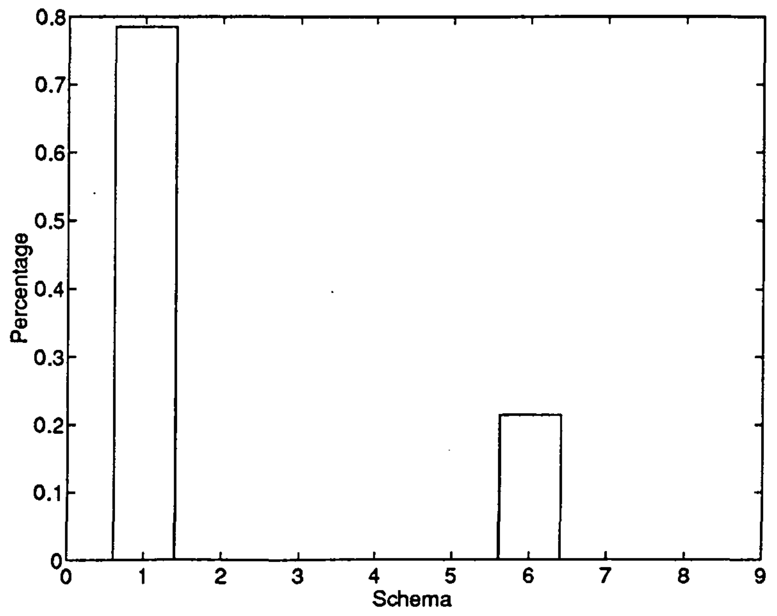

Figure 4: Histogram indicating the final schema converged to in 25 simulations under selection with noisy fitness values ($N = 128$).

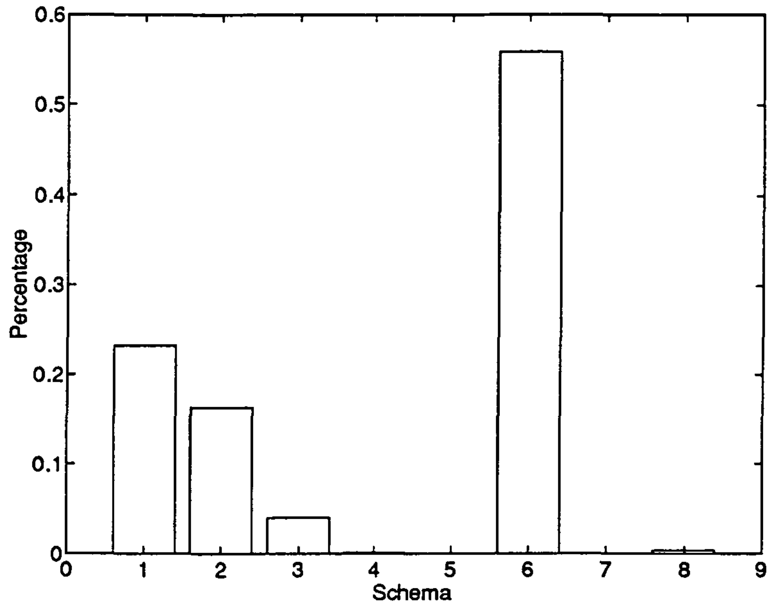

Figure 5: Histogram indicating the final schema converged to in 25 simulations under selection with noisy fitness values ($N = 32$).

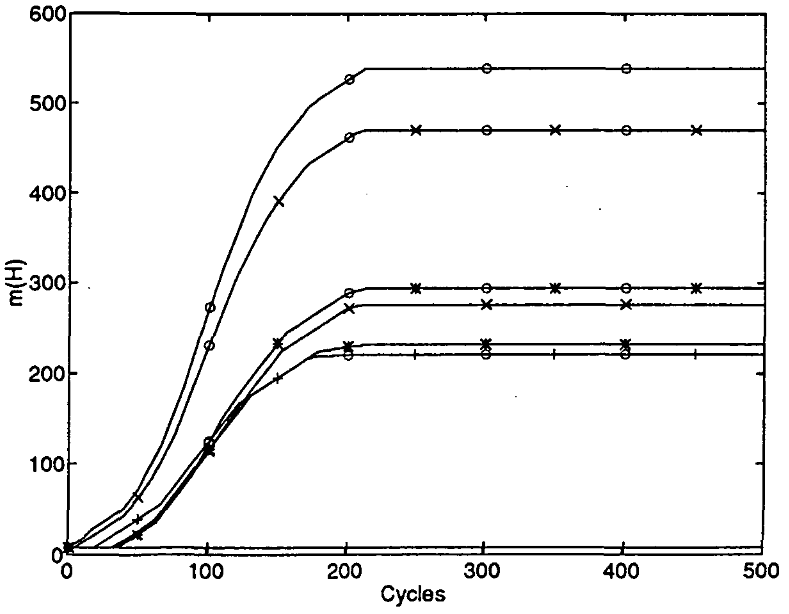

Figure 6: Evolution of numbers of schemata in simulation of the population resizing scheme. Symbols are as in previous figures.

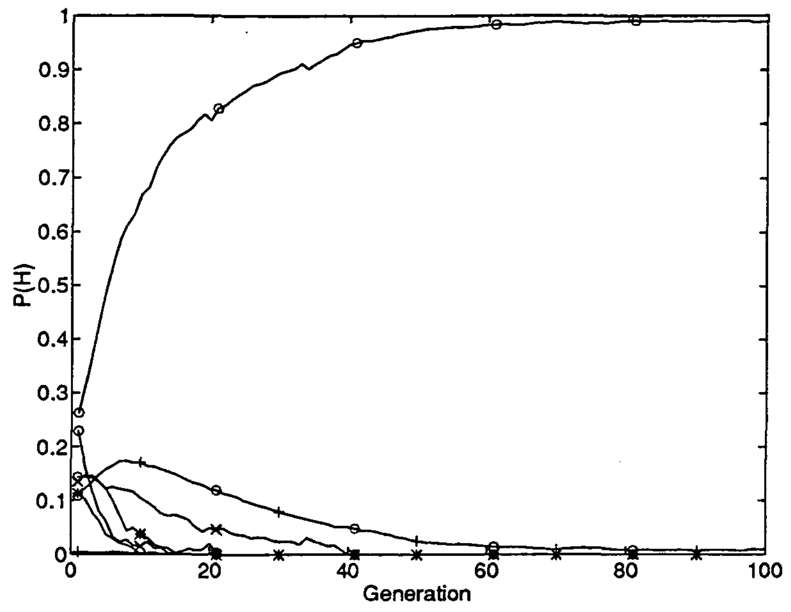

Figure 7: Evolution of population proportions starting from the resulting population size and proportions from the population resizing scheme, under selection with noisy fitness values. Symbols are as in previous figures.

Given the resulting proportions and population size at cycle 500 in Figure 6, one can iterate the noisy proportion equations to simulate selection after population resizing. Figure 7 shows the results of these iterations. Note the simulation quickly converges to the highest fitness schemata. Figure 8 shows a histogram of the convergence of 25 similar runs. In all of these runs, the most fit schema overtakes the population.

4 Implementation

The results presented above are only those of simulations. They certainly do not reflect the complexities of a live GA run with the suggested population sizing scheme. In particular, the schema fitness and variance values used in the simulation are assumed to be explicitly given, as opposed to estimated from the stochastic process inherent to a real GA. However, such expected value simulations have been useful in past GA studies (Goldberg, 1987; Smith & Goldberg, 1992). The simulations only consider the competitive schemata in a single partition. In a real GA, there will be an interplay of many partitions of schemata, some of which indicate that the population should be larger, some of which indicate that the population should be smaller. The results of the simulations used here indicate that the suggested scheme may be an effective method for automatically resizing populations. This section presents the next phase of testing the method: its implementation in a real GA.

The interplay of multiple schemata competitions in the real GA led to somewhat different dynamics than those of the simulations. In particular, the variations in population size were broader than desired. Often the population would expand to a much larger value than was necessary for a given test problem

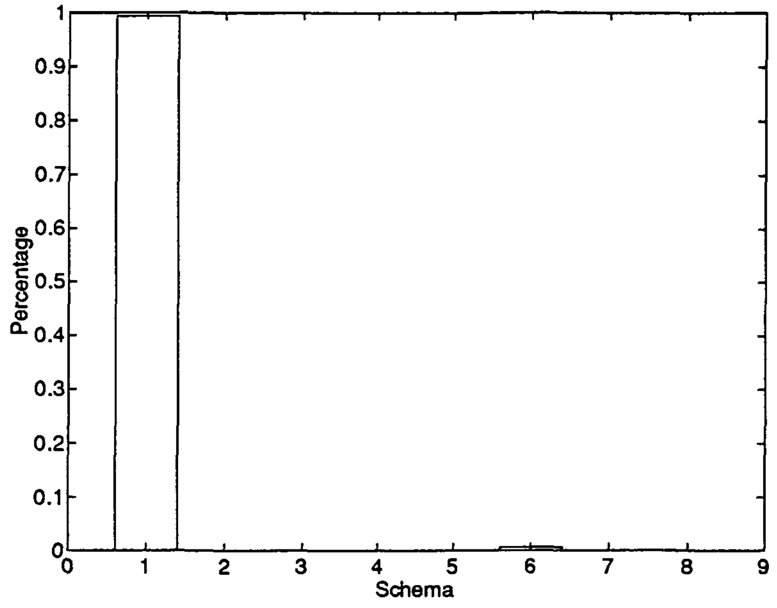

Figure 8: Histogram of final schema converged to in 25 simulations starting from the resulting population size and proportions from the population resizing scheme, under selection with noisy fitness values.

(see below). This expansion seems to be due to the conservative nature of the loss measure. Recall that expansion is based on the estimated expect loss,

$$\hat{L}(H_{1,2}, H_{3,4}) = |f(H_{1,2}) - f(H_{3,4})|\alpha(H_{1,2}, H_{3,4}).$$

The system must obtain the given level of target loss for *every* schemata competition, which is difficult in the real GA, and introduces broad, sustained oscillations in the population size. To reduce these oscillations, the criteria is relaxed:

$$\hat{L}(H_{1,2}, H_{3,4}) = \frac{|f(H_{1,2}) - f(H_{3,4})|}{n(H'_{1,2}) + n(H'_{3,4})}\alpha(H_{1,2}, H_{3,4}).$$

In effect, this weights the importance on any one selection loss less severely for larger population sizes. This seems logical in connection with the real GA, since is is less critical for the GA to select correctly every time in a given competition if that competition occurs several times in the population.

In implementing the algorithm simulated in previous sections, one must more seriously consider the transition from population sizing to actual selection. In the simulations, the transition was made abruptly, after the population size had obviously stabilized. While this is easy in an expected value simulation, it is difficult in an actual GA, where the complex interplay of various schemata make the convergence of population size less obvious. Although the population size may converge in expectation, it will still vary around the mean in the real GA. This effect makes the decision to switch to pure selection difficult. Rather than attempting to use some statistical measure to make this decision (and possibly introducing more tunable parameters), the implementation presented here folds selection and population resizing into one continuous operation. Details are as follows.

In the implementation presented here, growth or decay is accomplished by a procedure that is similar to stochastic remainder selection (Goldberg, 1989a; Wetzel, 1983). First, the growth rate is multiplied by two. If the growth rate for a mating pair is $2 \cdot G_{1,2}$, the pair is given the integer part of $2 \cdot G_{1,2}$ children deterministically, and an extra copy with probability equal to the fractional part of $2 \cdot G_{1,2}$. If the growth rate for a particular competition $G_{1,2,3,4}$ is greater than or equal to one, the growth rate is divided between the two mating pairs based on relative fitness, as was suggested earlier. If $f(H_{1,2}) = f(H_{3,4})$, and $G_{1,2,3,4} < 1$, the growth rate is also divided as was indicated in previous sections. However, if the growth rate $G_{1,2,3,4}$ is less than one, *all of losses are assigned to the less fit mating pair.* In other words, if $f(H_{1,2}) > f(H_{3,4})$, and $G_{1,2,3,4} < 1$, the growth rate is divided as follows:

$$G_{1,2} = 1,$$

$$G_{3,4} = G_{1,2,3,4}.$$

The motivation for this method is based on the population sizing theory discussed previously. A growth rate of less than one indicates that the variance level of the schemata competition at hand is such that selection can be performed with some confidence in the results. If the algorithm is started with a small population size, this condition will not occur frequently until late in the run, when the population size is corrected downward. Thus, the algorithm should have two distinct phases, with population expansion and little selection early on, and population contraction with selection later.

In an actual implementation of the algorithm, another important decision is how to set the mutation rate. Since mutation rate is usually tied to population size in a GA, there is no clear way to set a fixed value for the mutation rate in a GA where population size varies. In this implementation, the mutation rate is held at $1/N$, where N is population size, throughout the run.

Since the algorithm suggested is based on competitions of four strings at a time, it is helpful for the population size to remain a multiple of four. In the implementation presented here, this is accomplished as follows. If the new population size is not a multiple of four, a random decision is made as to whether to add or subtract individuals. This prevents a bias towards population expansion or reduction. If subtraction is selected, the last few individuals are deleted from the population. If addition is selected, individuals are randomly created. In either case, the resulting population size is the nearest multiple of four.

In addition to the previously discussed details, one must also consider that small values of L_t are likely to dictate population sizes that are larger than are practical. Thus, user dictated memory and computational time limitations must be added to the algorithm. In the implementation given here, individuals are randomly deleted when a maximum population size is exceeded. There may be better strategies for such deletion, including deletion based on fitness. Such strategies are an avenue for future study. Another important extension of this implementation is a user-specified limitation based on computational time. Implementing such a limitation will require online estimates of GA convergence time for a given population size, and corresponding adjustments of the maximum population size. Previous studies of convergence times for GA selection schemes may be helpful in this extension (Goldberg & Deb, 1990).

An implementation of the algorithm is included in the appendix. The main body of the algorithm is included in the subroutine **adaptpop()**. The remainder of the code is a slight variation of the SGA-C package (Smith et al., 1991). Note that the schema comparison procedures are implemented in fast-executing binary operations. The expense of these operations, and alternatives to them, are discussed in the final comments section of this paper.

5 Testing

This section formulates a test problem for the resizing technique, and demonstrates the technique's effects on this problem. It will be useful to employ a test problem where one can demonstrate that the results

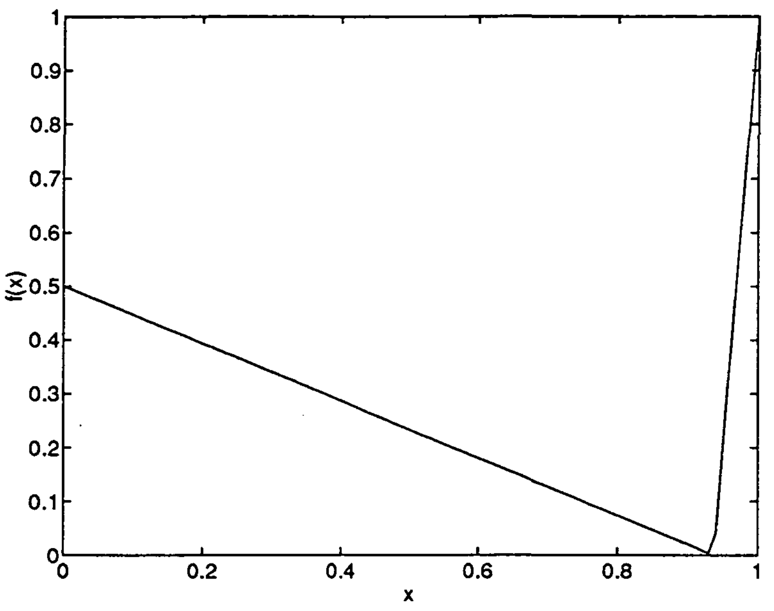

Figure 9: A partially deceptive function that the GA can solve.

of a fixed-population size GA vary with the selection of the population size. To do this, it is necessary to insure that mutation alone is not sufficient to (eventually) locate the problems optima. If mutation is sufficient, then the eventual convergence of the GA to the optima is assured. Population size may effect transient response of the GA, but this will be insufficient for the desired illustration of the population sizing scheme's effects. Because of this, it is necessary to have at least a partial degree of *deception* in the problem (Goldberg, 1989a), so that order one building blocks alone cannot lead to the optima.

Consider the following problem:

$$\text{fitness} \;=\; f(x) = \begin{cases} m_1 x + b_1 & \text{if } x_1 < F_2 \\ m_2 x + b_2 & \text{otherwise} \end{cases}$$

where x is the interpretation of the GA bit string as an integer, scaled to a real number between zero and one, and $m_1 = -0.533$, $b_1 = 0.5$, $m_2 = 16$, $b_2 = -15$, and $F_2 = 15/16$. This function is plotted in Figure 9. The function is partially deceptive, which can be shown by considering the four order-one schemata competitions represented by the four most-significant bits in x, shown in Table 1. The function is partially deceptive, since the first two schemata competitions have selective pressure towards the false optima ($x = 0$), while the last two have selective pressure towards the true optima ($x = 1$). Moreover, the order four competitive partition in the most-significant bits is not deceptive, as is shown in Table 1. If disruption of the schema 1111 ...*** is low, one would expect that the GA would select this schema over others in its partition, and the process would converge to the correct optima.

However, it is easy to imagine the GA being misled by the problem shown in Figure 9. If high order bits are set to zeros (through disruption of the high-order schema or selection error), the remainder of the string will improve overall performance by moving away from the true optima at $x = 1$. However, the linkage in the high order building blocks is tight, and the GA is not usually not misled by this function,

schema H	approximate average fitness $f(H)$
0*** ...***	0.3672
1*** ...***	0.1679
*0** ...***	0.2987
*1** ...***	0.2359
0* ...*	0.2668
1* ...*	0.2683
0 ...	0.2501
1 ...	0.2683

Table 1: Approximate order-1 schema average fitness values for the partially deceptive function.

schema H	approximate average fitness $f(H)$
0000 ...***	0.4843
0001 ...***	0.4509
0010 ...***	0.4174
0011 ...***	0.3840
0100 ...***	0.3506
0101 ...***	0.3171
0110 ...***	0.2837
0111 ...***	0.2502
1000 ...***	0.2168
1001 ...***	0.1833
1010 ...***	0.1499
1011 ...***	0.1164
1100 ...***	0.0830
1101 ...***	0.0496
1110 ...***	0.0161
1111 ...***	0.5241

Table 2: Approximate order-4 schema average fitness values for the partially deceptive function.

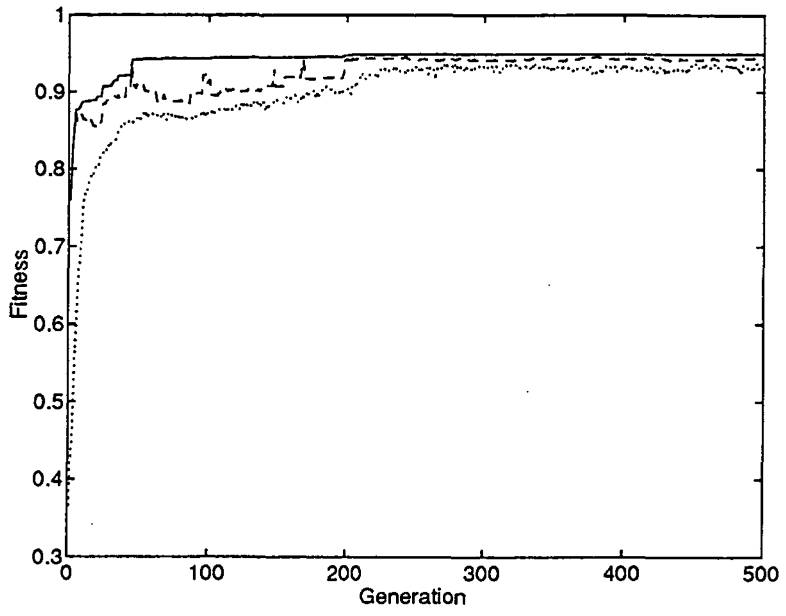

Figure 10: Average results for ten simple GA runs with population size 50 on the partially deceptive problem. The best fitness individual located thus far is shown with a solid line, maximum fitness in the current generation is shown with a dashed line, and average fitness in the current generation is shown with a dotted line.

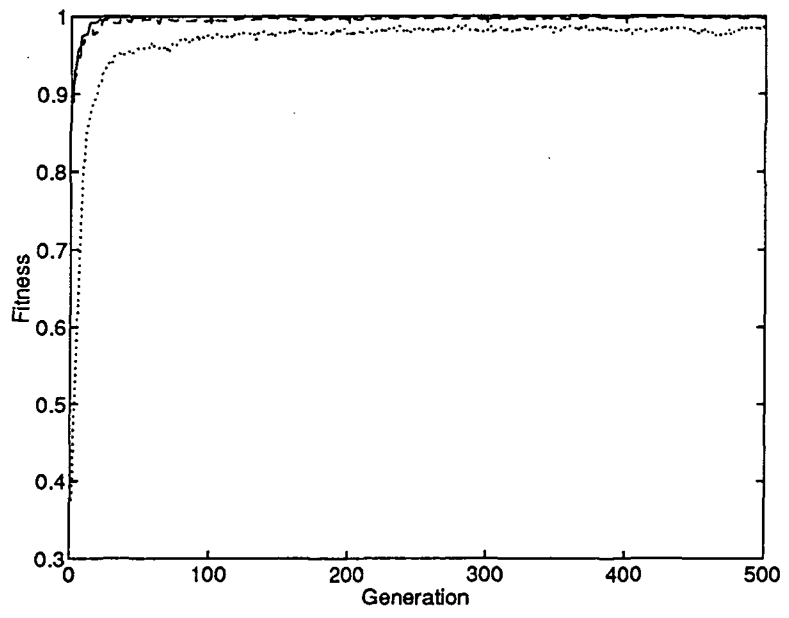

Figure 11: Average results for ten simple GA runs with population size 100 on the partially deceptive problem. The best fitness individual located thus far is shown with a solid line, maximum fitness in the current generation is shown with a dashed line, and average fitness in the current generation is shown with a dotted line.

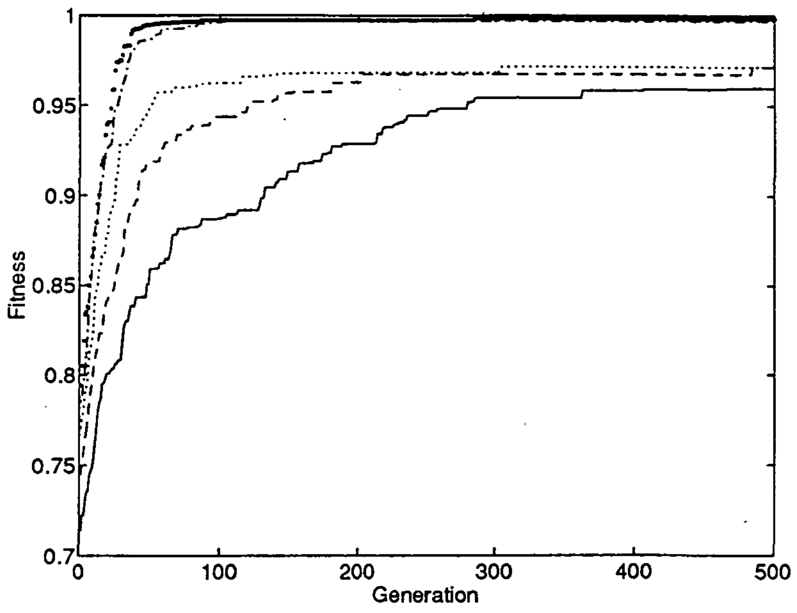

Figure 12: Plots show the best individual found up to the current generation for various population sizes on the modified partially deceptive problem. Plots represent an average of ten runs. Population size 50 is shown with a solid line, 100 with a dashed line, 200 with a dotted line, 500 with a dashed-dotted line, and 1000 with a large-dotted line.

even with relatively small population sizes, as Figures 10 and 11 illustrate. These figures show GA results for populations of size 50 and 100 (respectively), with bit strings of length 21, crossover rate $p_c = 0.6$, mutation rate $p_m = 0.001$, and tournament selection with tournaments of size two (Goldberg & Deb, 1990). The results shown are averaged over ten independent runs. For a population size of 50, the GA converges to the true optima nine out of ten times. For the population size of 100, the GA consistently converges to the true optima. Qualitatively similar results were obtained with other parameter settings.

Given a partially deceptive problem that the GA can solve (with appropriately linked building blocks), consider decreasing the selective signal-to-noise ratio. Consider the previously demonstrated problem, concatenated with a 21-bit, bitwise linear problem. In other words, consider interpreting the first 21 bits of the string as x in the previous problem, and each of the remaining bits as a separate variable x_i. The suggested fitness function is

$$\text{fitness} = 0.1 \cdot f(x) + \frac{0.9}{21} \cdot \sum_{i=1}^{21} x_i.$$

In effect, the previously discussed partially deceptive function is added to the simplest of all problems for the GA. However, the results are not what one might expect. Figures 12 and 13 illustrate the effects of GAs with various population sizes on this problem. Figure 12 shows the best individual found up to the current generation, and Figure 13 shows the maximum individual in the current population. Parameters are otherwise as in the previous example. Each line on these plots represents an average of ten GA runs. Note that the function $f(x)$ represents the final 10% of fitness in these experiments, and that if the GA is misled to $x = 0$, the maximum fitness is 0.95. With small population sizes, this happens often. As

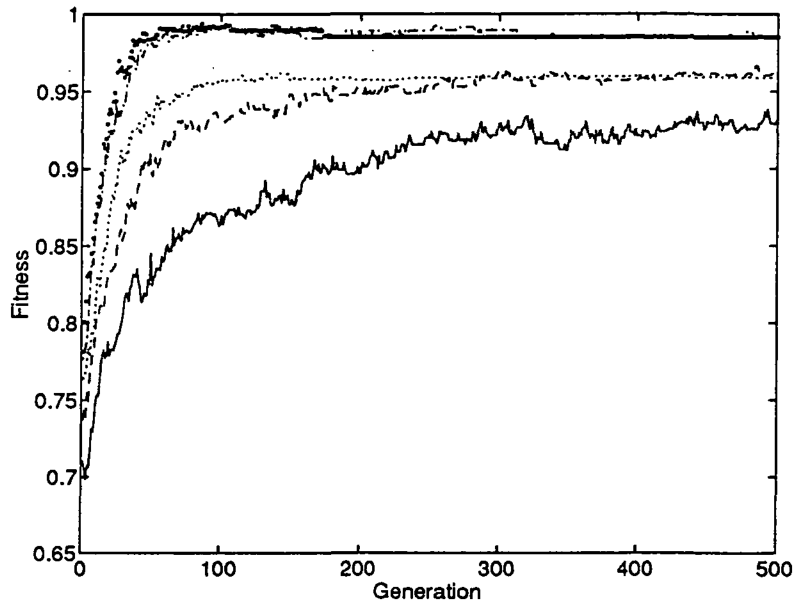

Figure 13: Plots show the maximum individual in the current generation for various population sizes on the modified partially deceptive problem. Plots represent an average of ten runs. Population size 50 is shown with a solid line, 100 with a dashed line, 200 with a dotted line, 500 with a dashed-dotted line, and 1000 with a large-dotted line.

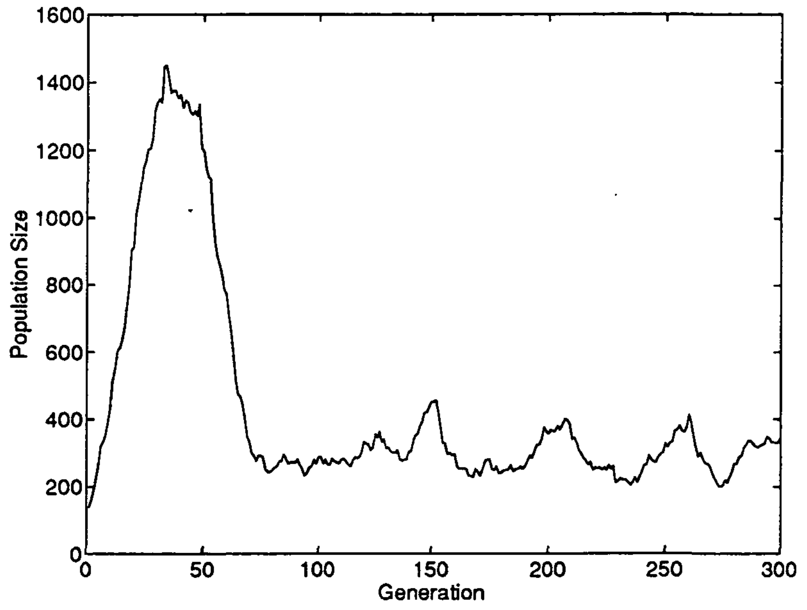

Figure 14: Population size versus generation for the adaptive population sizing GA applied to the modified partially deceptive function.

the population size is increased, the GA more frequently finds the correct solution. Qualitatively similar results were obtained for other parameter settings.

The failures in these runs are caused by the low signal-to-noise ratios for the critical, higher order bits of parameter x. In effect, for small populations the GA initially concentrates its selective decisions on the most important bits. These bits are the 21 x_is. The higher order bits of x are carried along in these selective decisions, often with incorrect values. These bits are what have been called *hitchhikers* in the GA theoretic literature (Forrest & Mitchell, 1993). In effect, hitchhikers are the result of insufficient population size.

Given a problem where population sizing can be seen as critical to overall performance, one can examine the effects of adaptive population sizing, with a foundation for interpreting effects. Figures 14 and 15 show the results of a representative run of the adaptive population sizing code. This run used the parameters shown in Table 3. Results of similar, independent runs were qualitatively similar. The GA consistently sized the population in a similar range, and found the true optima. Note that the population size first increases, lowering variances and spreading a population over the search space. Then, as the population size drops, and selection is applied to the individuals deleted, recombination assembles the correct solution. This can be seen in the jump from the false optima to the true optima when overall population size decreases. Near the end of the run, the population size remains steady at a value lower than its maximum. This is because selection has lowered the diversity of the population, thus lowering the variance of population members, and lowering the adequate population size needed to match the target loss.

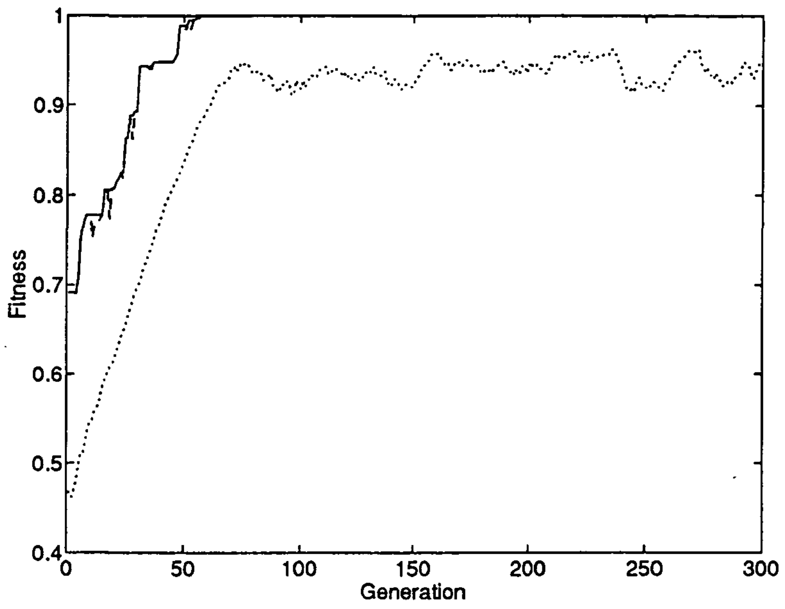

Figure 15: Fitness versus generation for the adaptive population sizing GA applied to the modified partially deceptive function. Best individual found up to the current generation is shown with a solid line, maximum individual in the current population is shown with a dashed line, and the current population average is shown with a dotted line.

Initial Population Size	100
Maximum Population Size	5000
Crossover Rate	0.6
Mutation Rate	0.001
Target Loss L_t	0.0001
Sigmoid Slope β	10
Maximum Growth Rate γ	1.0

Table 3: Parameters used in the representative run of the adaptive population sizing GA.

6 Final Comments

The theoretical developments, simulations, and experiments presented in this paper indicate that the suggested technique for adaptive population size adjustment is viable. Further experimentation will be necessary to fully confirm the effectiveness of the algorithm. The fundamental framework of the algorithm is firmly founded. That is, that schemata variances effect the accuracy of selection, and that population sizes effect variances. Therefore, population size adjustment should be based on variances. The suggested procedure uses mating pairs to estimate schemata fitness variances, in much the same way that the traditional GA uses individuals to estimate schemata average fitness values. This keeps with the notion of implicit parallelism in the GA, and allows for explicit parallelism as well.

There are aspects of the suggested algorithm that could be altered while staying within the same conceptual framework. In particular, it may be useful to explore alternatives to the use of the sigmoidal mapping to determine the growth rate $G_{1,2,3,4}$, and the division of $\hat{L}$ by $n(H'_{1,2}) + n(H'_{3,4})$ to control population over-expansion. In effect, these features are a control strategy over population growth. Other stochastic control strategies may be effective. Exploration of a variety of such methods will be an important area for future research. However, the direct or indirect use of estimated variance and loss as a basis seems appropriate for any adaptive population sizing scheme.

Parts of the suggested algorithm have much in common with fitness sharing (Deb, 1989). Fitness sharing spreads the population over high-fitness niches in the search space. The population resizing algorithm spreads the population across niches of variance. Like sharing, the population resizing algorithm uses a count of similar strings to control reproduction. A word must be said about the expense of counting the number of competitive schema in the suggested algorithm. Like the comparisons required in fitness sharing, this is an $O(N^2)$ operation. However, in considering this expense, one must compare it to the $O(N)$ operation of calculating string fitness values. In many GA applications, the expense of calculating fitness values far exceeds the cost of $O(N^2)$ masked, binary comparisons. Therefore, this expense may be negligible. Also, it may be possible to estimate the schema counts. Oei, Goldberg, and Chang (1991) have suggested using samples to evaluate approximate counts in fitness sharing, and a similar technique may be useful in the population resizing scheme. It may also be possible to abstract a method of estimation from recent work on sharing-like behavior in GA immune system simulations (Smith et al., 1993). Another method for estimating counts may be to periodically gather strings into *families* that share common schema. The resulting family sizes could be used as estimated counts. Since the results of a mating usually preserve common schema between parents, adding children strings to a family would simply result in an increment of the estimated count. As mutations occurs, so would new families. Periodic creation of hybrid families could help to insure accurate count estimates.

The algorithm presented here is a logical, practical, extension of the traditional GA, but one with extensive potential. If GAs can control their search by automatically adjusting population sizes, they can effectively expand and contract to accommodate variations in problem complexity, and variations in available computational resources. A class of adaptive algorithms of this sort could extend the GAs applicability, and extend its usability beyond the GA expert, to the novice user.

7 Acknowledgments

The author gratefully acknowledges support for this work provided by NASA under grant number NAG 9-625. The author also acknowledges support provided by the National Science Foundation under grant number ECS-9212066.

8 Appendix

This appendix includes the complete code of the population resizing code used in this paper.

References

Ackley, D. H. (1987). *A connectionist machine for genetic hillclimbing.* Boston: Kluwer Academic.

Deb, K. (1989). *Genetic algorithms in multimodal function optimization* (TCGA Report No. 89002). Tuscaloosa: The University of Alabama, The Clearinghouse for Genetic Algorithms.

Forrest, S., & Mitchell, M. (1993). Relative building-block fitness and the building-block hypothesis. In D. Whitley (Ed.), *Foundations of genetic algorithms 2* (109–126). San Mateo, CA: Morgan-Kaufmann.

Goldberg, D. E. (1985). *Optimal initial population size for binary-coded genetic algorithms* (TCGA Report No. 85001). Tuscaloosa: University of Alabama, The Clearinghouse for Genetic Algorithms.

Goldberg, D. E. (1987). Simple genetic algorithms and the minimal, deceptive problem. In L. Davis (Ed.), *Genetic algorithms and simulated annealing* (74–88). Los Altos, CA: Morgan Kaufmann.

Goldberg, D. E. (1989a). *Genetic algorithms in search, optimization, and machine learning.* Reading, MA: Addison–Wesley.

Goldberg, D. E. (1989b). Sizing populations for serial and parallel genetic algorithms. *Proceedings of the Third International Conference on Genetic Algorithms*, 70–79.

Goldberg, D. E., & Deb, K. (1990). *A comparative analysis of selection schemes used in genetic algorithms* (TCGA Report No. 90007). Tuscaloosa: The University of Alabama, The Clearinghouse for Genetic Algorithms.

Goldberg, D. E., Deb, K., & Clark, J. H. (1991). *Genetic algorithms, noise, and the sizing of populations* (IlliGAL Technical Report No. 91010). Urbana, Illinois: University of Illinois at Urbana-Champaign.

Goldberg, D. E., & Rudnick, M. (1991). Genetic algorithms and the variance of fitness. *Complex Systems*, *5*, 265–278.

Holland, J. H. (1975). *Adaptation in natural and artificial systems.* Ann Arbor: The University of Michigan Press.

Oei, C. K., Goldberg, D. E., & Chang, S. (1991). *Tournament selection, niching, and the preservation of diversity* (IlliGal Tech. Rep. 91001). Urbana, IL: University of Illinois, Department of General Engineering.

Smith, R. E., Forrest, S., & Perelson, A. S. (1993). Searching for diverse, cooperative populations with genetic algorithms. *Evolutionary Computation*, *1*(2), 127–149.

Smith, R. E., & Goldberg, D. E. (1992). Diploidy and dominance in artificial genetic search. *Complex Systems*, *6*(3), 251–285.

Smith, R. E., Goldberg, D. E., & Earickson, J. (1991). *SGA-C: A C-language implementation of a simple genetic algorithm* (TCGA Report No. 91002). Tuscaloosa: University of Alabama, The Clearinghouse for Genetic Algorithms.

Thrun, S. B. (1992). The role of exploration in learning control. In *Handbook of intelligent control: Neural, fuzzy, and adaptive approaches* (527–560). New York: Van Nostrand Reinhold.

Valenzuela-Rendon, M. (1989). *Two analysis tools to describe the operation of classifier systems* (TCGA Report No. 89005). Tuscaloosa: The University of Alabama, The Clearinghouse for Genetic Algorithms.

Wetzel, A. (1983). *Evaluation of the effectiveness of genetic algorithms in combinatorial optimization.* Unpublished manuscript, University of Pittsburgh, Pittsburgh.

```
/*-----------------------------------------------------------*/
/* adaptpop.c - adaptively resizes GA population           */
/*-----------------------------------------------------------*/

#include "external.h"

static int     *matelist, matepos;
/* tournament list, position in list */

mate_memory()
{
  unsigned nbytes;
  int j;

  nbytes = maxpop*sizeof(int);
  if((matelist = (int *) malloc(nbytes)) == NULL)
    nomemory(stderr,"matelist");
}

makematelist()
{
  reset();
  matepos = 0;
}

reset()
     /* Shuffles the matelist at random */
{
  int i, rand1, rand2, temp;

  for(i=0; i<popsize; i++) matelist[i] = i;

  for(i=0; i < popsize; i++)
    {
      rand1=rnd(i,popsize-1);
      rand2=rnd(i,popsize-1);
      temp = matelist[rand1];
      matelist[rand1]=matelist[rand2];
      matelist[rand2]=temp;
    }
}

getmate()
     /* sets up random list of pop members */
{
  int index;

  index = matelist[matepos];
  matepos++;

  return(index);
}

mate_free()
{
  free(matelist);
};


adaptpop()
{

  for(gen=0;gen<maxgen;gen++)
    {

      fprintf(outfp,"\nRUN %d of %d: GENERATION %d->%d\n"
...

      /*application dependent routines*/
      application();
```

mate_memory makematelist reset getmate mate_free adaptpop

```
    /* shuffle the indicies of population to random order */
    /* so that mating pairs can be assigned randomly      */

    makematelist();

    Gtotal = 0;
    j = 0;

    while(j<popsize)
      {
        /* select the first mating pair */
        mate1 = getmate();
        mate2 = getmate();

        /* determine the shared schemata in the mating pair */
        for(i=0;i<=chromsize;i++)
          {
            J12[i] = (oldpop[mate1].chrom[i] & oldpop[mate2].chrom[i])|
              ((~oldpop[mate1].chrom[i]) & (~oldpop[mate2].chrom[i]));
          }


        /* determine the fitnesses of the two strings and use */
        /* them to evaluate the fitness and sigma of H12      */
        fmate1 = oldpop[mate1].fitness;
        fmate2 = oldpop[mate2].fitness;
        fH12 = (fmate1+fmate2)/2;
        temp = fmate1-fmate2;
        sigH12 = (temp*temp)/4;

        /* select the second mating pair */
        mate3 = getmate();
        mate4 = getmate();

        /* determine the shared schemata in the mating pair */
        for(i=0;i<=chromsize;i++)
          {
            J34[i] = (oldpop[mate3].chrom[i] & oldpop[mate4].chrom[i])|
              ((~oldpop[mate3].chrom[i]) & (~oldpop[mate4].chrom[i]));
          }


        /* determine the fitnesses of the two strings and use */
        /* them to evaluate the fitness of H34               */
        fmate3 = oldpop[mate3].fitness;
        fmate4 = oldpop[mate4].fitness;
        fH34 = (fmate3+fmate4)/2;
        temp = fmate3-fmate4;
        sigH34 = (temp*temp)/4;

        /* determine the competitive schemata for each */
        /* mating pair competition                     */
        for(i=0;i<=chromsize;i++)
          {
            J1234[i] = J12[i] & J34[i];
          }

        /* check if H12 = H34 */
        flag=0;
        for(i=0;i<chromsize;i++)
          {
            if
(0 == ((oldpop[mate1].chrom[i]^oldpop[mate3].chrom[i])&J1234[i]))
               flag++;
          }


        if(flag == chromsize)
```

```
            {
              G12=1;
              G34=1;
            }
          else
            {
              /* otherwise */
              /* obtain a count of each competitive schemata */
              n12 = 0;
              for(k=0;k<popsize;k++)
                {
                  flag = 0;
                  for(i=0;i<chromsize;i++)
                    {
                      if                                                          ...
(0 != ((oldpop[k].chrom[i] ^ oldpop[mate1].chrom[i]) & J1234[i]))
                        {
                          flag = 1;
                          continue;
                        }
                    }
                  if(flag == 0)
                    {
                      n12++;
                    }
                }

              n34 = 0;
              for(k=0;k<popsize;k++)
                {
                  flag = 0;
                  for(i=0;i<chromsize;i++)
                    {
                      if                                                          ...
(0 != ((oldpop[k].chrom[i] ^ oldpop[mate3].chrom[i]) & J1234[i]))
                        {
                          flag = 1;
                          continue;
                        }
                    }
                  if(flag == 0)
                    {
                      n34++;
                    }
                }

              /* calculate Lhat for each mating pair competition */
              temp = sigH12/n12 + sigH34/n34;

              if(temp == 0)
                {
                  Lhat = 0;
                  zabs = 0;
                  alpha = 0;
                }
              else
                {
                  z = (fH12 - fH34)/pow(temp,.5);
                  zabs = fabs(z);
                  if(zabs > 2)
                    {
                      temp = zabs*pow(2*pi,.5);
                      alpha = (exp(-(zabs*zabs)/2))/temp;
                    }
                  else
                    {
                      alpha = 1/(1 + exp(-1.6*zabs));
                    }

                  temp = fH12 - fH34;
            Lhat = (fabs(temp))*alpha;
          }


        /* use Lhat/Lt as a basis for assigning more population members */
        Lhat = Lhat/(n12 + n34);
        G1234 = (1-gmma) + ((2*gmma)/(1 + exp(-beta*(Lhat/Lt - 1))));


        if(G1234 > 1)
          {
            p12 = fH12/(fH12+fH34);
            p34 = fH34/(fH12+fH34);
          }
        else
          if(fH12 > fH34)
            {
              p12 = 0;
              p34 = 1;
            }
          else
            if(fH12 < fH34)
              {
                p12 = 1;
                p34 = 0;
              }
            else
              {
                p12 = 0.5;
                p34 = 0.5;
              };



        G12 = (G1234-1)*p12 + 1;
        G34 = (G1234-1)*p34 + 1;
      };


    Grate12 = 0;
    G12 = 2*G12;

    while(G12>1)
      {
        Grate12++;
        G12 = G12 - 1;
      }
    if(flip(G12))
      {
        Grate12++;
      }

    Grate34 = 0;
    G34 = 2*G34;

    while(G34>1)
      {
        Grate34++;
        G34 = G34 - 1;
      }
    if(flip(G34))
      {
        Grate34++;
      }

    /* confirm that popsize < maxpop */
    if((Gtotal + Grate12 + Grate34) >= maxpop)
      break;
```

```
        /* update population */
        for(k=1;k<=Grate12;k++)
          {
            i = Gtotal - 1 + k;

            jcross = crossover(oldpop[mate1].chrom,oldpop[mate2].chrom,
                                 newpop[i].chrom,newpop[i+1].chrom);
            mutation(newpop[i].chrom);

            objfunc(&(newpop[i]));
            newpop[i].parent[0] = mate1 + 1;
            newpop[i].xsite = jcross;
            newpop[i].parent[1] = mate2 + 1;

          }

        for(k=1;k<=Grate34;k++)
          {
            i =  Gtotal - 1 + Grate12 + k;

            jcross = crossover(oldpop[mate3].chrom,oldpop[mate4].chrom,
                                 newpop[i].chrom,newpop[i+1].chrom);
            mutation(newpop[i].chrom);

            objfunc(&(newpop[i]));
            newpop[i].parent[0] = mate3 + 1;
            newpop[i].xsite = jcross;
            newpop[i].parent[1] = mate4 + 1;

          }

        /* sum the growth rates */
        Gtotal = Gtotal + Grate12 + Grate34;

        /* increment population index */
        j = j + 4;

      }


    if(Gtotal < maxpop)
      popsize = Gtotal;
    else
      popsize = maxpop;


    /* increment the population size */
    /* check popsize */
    if(popsize < 4)
      {
        tempm = pmutation;
        pmutation = 0.5;
        for(i=0;i<=4;i++)
          {
            for(cnt=0;cnt<chromsize;cnt++)
              newpop[i].chrom[cnt] = newpop[0].chrom[cnt];
            mutation(newpop[i].chrom);
          }
        popsize = 4;
        pmutation = tempm;
      }
    else
      {
        if(popsize%4 != 0)
          {
            if((popsize - (popsize%4)) < 4)
              {
                tempm = pmutation;
                pmutation = 0.5;
                for(i=1;i<=(4-popsize%4);i++)
                {
                  for(cnt=0;cnt<chromsize;cnt++)
                    newpop[popsize-1+i].chrom[cnt] = newpop[0].chrom[cnt];
                  mutation(newpop[popsize-1+i]);
                }
              popsize = popsize + (4 - popsize%4);
              pmutation = tempm;
            }
          else
            if((popsize + (4 - popsize%4)) > maxpop)
              {
                popsize = popsize - (popsize%4);
              }
            else
              if(flip(0.5))
                {
                  popsize = popsize - (popsize%4);
                }
              else
                {
                  tempm = pmutation;
                  pmutation = 0.5;
                  for(i=1;i<=(4-popsize%4);i++)
                    {
                      for(cnt=0;cnt<chromsize;cnt++)
...
newpop[popsize-1+i].chrom[cnt] = newpop[0].chrom[cnt];
                      mutation(newpop[popsize-1+i]);
                    }
                  popsize = popsize + (4 - popsize%4);
                  pmutation = tempm;
                };
        };
    };


    /* report on results */
    report();

    /* compute fitness statistics on new population */
    statistics(newpop);

    /* advance the generation */
    temp2 = oldpop;
    oldpop = newpop;
    newpop = temp2;


    /* change mutation rate */
    pmutation=(1.0)/(float)popsize;

  }  /* end generation */

}
```

```
/*------------------------------------------------------------------------*/
/* sga.h - global declarations for main(), all variable declared herein must */
/*      also be defined as extern variables in external.h !!!           */
/*------------------------------------------------------------------------*/

#define LINELENGTH 80                                      /* width of printout */
#define BITS_PER_BYTE 8                /* number of bits per byte on this machine */
#define UINTSIZE (BITS_PER_BYTE*sizeof(unsigned))      /* # of bits in unsigned */
#include <stdio.h>

/* file pointers */
FILE *outfp, *infp;

/* Global structures and variables */
struct individual
{
    unsigned *chrom;                  /* chromosome string for the individual */
    double   fitness;                              /* fitness of the individual */
    int      xsite;                                /* crossover site at mating */
    int      parent[2];                   /* who the parents of offspring were */
    int      *utility;           /* utility field can be used as pointer to a */
             /* dynamically allocated, application-specific data structure */
};
struct bestever
{
    unsigned *chrom;        /* chromosome string for the best-ever individual */
    double   fitness;                  /* fitness of the best-ever individual */
    int      node;                           /* node from whence came best-ever */
    int      generation;                         /* generation which produced it */
};

struct individual *oldpop;                       /* last generation of individuals */
struct individual *newpop;                       /* next generation of individuals */
struct bestever bestfit;                              /* fittest individual so far */
double sumfitness;                         /* summed fitness for entire population */
double max;                                      /* maximum fitness of population */
double avg;                                      /* average fitness of population */
double min;                                      /* minumum fitness of population */
float  pcross;                                         /* probability of crossover */
float  pmutation;                                       /* probability of mutation */
float  Lt;                                       /* desired target selection loss */
int    maxpop;                                          /* maximum population size */
int    tourneysize;                                   /* tournament selection size */
int    numfiles;                                          /* number of open files */
int    popsize;                                                  /* population size */
int    lchrom;                             /* length of the chromosome per individual */
int    chromsize;                   /* number of bytes needed to store lchrom string */
int    gen;                                            /* current generation number */
int    maxgen;                                         /* maximum generation number */
int    run;                                                  /* current run number */
int    maxruns;                                    /* maximum number of runs to make */
int    printstrings = 1;        /* flag to print chromosome strings (default on) */
int    nmutation;
int    ncross;                                     /* number of crossovers per node */
float beta,gmma;

/* Variables added for adaptive population sizing */
struct individual *temp2;
float fmate1,fmate2,fmate3,fmate4,fH12,fH34,temp,sigH12,sigH34;
float z,zabs,alpha,Lhat,pi=3.14159,p12,p34;
float G1234,G12,G34;
int cnt,i,j=0,k,n12,n34,flag,mate1,mate2,mate3,mate4;
int Gtotal,Grate12,Grate34,jcross;
unsigned *J12,*J34,*J1234;
int tempm;


/* Application-dependent declarations go after here... */
```

A

3

```
/*---------------------------------------------------------------------*/
/* external.h - external global declarations from sga.h.              */
/*---------------------------------------------------------------------*/

#define LINELENGTH 80                                  /* width of printout */
#define BITS_PER_BYTE 8                 /* number of bits per byte on this machine */
#define UINTSIZE (BITS_PER_BYTE*sizeof(unsigned))        /* # of bits in unsigned */
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
extern float randomperc();

/* file pointers */
extern FILE *outfp, *infp;

/* Global structures and variables */
struct individual
{
    unsigned *chrom;                  /* chromosome string for the individual */
    double   fitness;                            /* fitness of the individual */
    int      xsite;                              /* crossover site at mating */
    int      parent[2];                  /* who the parents of offspring were */
    int      *utility;           /* utility field can be used as pointer to a */
                 /* dynamically allocated, application-specific data structure */
};
struct bestever
{
    unsigned *chrom;         /* chromosome string for the best-ever individual */
    double   fitness;                   /* fitness of the best-ever individual */
    int ,    node;                        /* node from whence came best-ever */
    int      generation;                       /* generation which produced it */
};

extern struct individual *oldpop;             /* last generation of individuals */
extern struct individual *newpop;             /* next generation of individuals */
extern struct bestever bestfit;                      /* fittest individual so far */
extern double sumfitness;                /* summed fitness for entire population */
extern double max;                             /* maximum fitness of population */
extern double avg;                             /* average fitness of population */
extern double min;                             /* minumum fitness of population */
extern float  pcross;                                /* probability of crossover */
extern float  pmutation;                              /* probability of mutation */
extern float  Lt;                              /* desired target selection loss */
extern int    maxpop;                                /* maximum popualtion size */
extern int    tourneysize;                           /* tournament selection size */
extern int    numfiles;                                  /* number of open files */
extern int    popsize;                                         /* population size */
extern int    lchrom;                /* length of the chromosome per individual */
extern int    chromsize;       /* number of bytes needed to store lchrom string */
extern int    gen;                                   /* current generation number */
extern int    maxgen;                                /* maximum generation number */
extern int    run;                                          /* current run number */
extern int    maxruns;                          /* maximum number of runs to make */
extern int    printstrings;  /* flag to print chromosome strings (default on) */
extern int    nmutation;                        /* number of mutations per node */
extern int    ncross;                           /* number of crossovers per node */
extern float beta,gmma;

/* Variables added for adaptive population sizing */
extern struct individual *temp2;
extern float fmate1,fmate2,fmate3,fmate4,fH12,fH34,temp,sigH12,sigH34;
extern float z,zabs,alpha,Lhat,pi=3.14159,p12,p34;
extern float G1234,G12,G34;
extern int cnt,i,j=0,k,n12,n34,flag,mate1,mate2,mate3,mate4;
extern int Gtotal,Grate12,Grate34,jcross;
extern unsigned *J12,*J34,*J1234;
extern int tempm;
```

```
/* Application-dependent external declarations go after here... */
```

```
/*********************************************/
/*    Simple Genetic Algorithm - SGA    */
/*        Haploid Version          */
/* (c)  David Edward Goldberg 1986     */
/*        All Rights Reserved        */
/*    C translation by R.E. Smith      */
/*  v1.1 modifications by Jeff Earickson  */
/*********************************************/

#include "sga.h"

main(argc,argv)
     int argc;
     char *argv[];
{
  struct individual *temp;
  FILE    *fopen();
  void    copyright();
  char    *malloc();


  /* determine input and output from program args */
  numfiles = argc - 1;
  switch(numfiles)
    {
    case 0:
      infp = stdin;
      outfp = stdout;
      break;
    case 1:
      if((infp = fopen(argv[1],"r")) == NULL)
        {
          fprintf(stderr,"Input file %s not found\n",argv[1]);
          exit(-1);
        }
      outfp = stdout;
      break;
    case 2:
      if((infp = fopen(argv[1],"r")) == NULL)
        {
          fprintf(stderr,"Cannot open input file %s\n",argv[1]);
          exit(-1);
        }
      if((outfp = fopen(argv[2],"w")) == NULL)
        {
          fprintf(stderr,"Cannot open output file %s\n",argv[2]);
          exit(-1);
        }
      break;
    default:
      fprintf(stderr,"Usage is: sga [input file] [output file]\n");
      exit(-1);
    }


  /* print the author/copyright notice */
  copyright();

  if(numfiles == 0)
    fprintf(outfp," Number of GA runs to be performed-> ");
  fscanf(infp,"%d",&maxruns);

  for(run=1; run<=maxruns; run++)
    {
      /* Set things up */
      initialize();

      /* adaptively resize population, perform main GA loop */
      adaptpop();
```

```
        freeall();
    }

}
```

```
/*-----------------------------------------------------------------------*/
/* app.c - application dependent routines, change these for different problem */
/*-----------------------------------------------------------------------*/

#include <math.h>
#include "external.h"

FILE *appfp;
char appname[20];



application()
      /* this routine should contain any application-dependent computations */
      /* that should be performed before each GA cycle. called by main()    */
{
}


app_data()
      /* application dependent data input, called by init_data() */
      /* ask your input questions here, and put output in global variables */
{
  fprintf(outfp,"Input application output filename ----->");
  fscanf(infp,"%s",appname);
  if((appfp = fopen(appname,"w")) == NULL)
    {
      fprintf(stderr,"Application output file %s not found\n",appname);
      exit(-1);
    }
}


app_free()
      /* application dependent free() calls, called by freeall() */
{
}


app_init()
      /* application dependent initialization routine called by intialize() */
{
}


app_initreport()
      /* Application-dependent initial report called by initialize() */
{
  fprintf(appfp,"m=[");
}


app_malloc()
      /* application dependent malloc() calls, called by initmalloc() */
{
  /*   char *malloc(); */
}


app_report()
      /* Application-dependent report, called by report() */
{
  fprintf(appfp,"%d %g %g %g %g\n",popsize,bestfit.fitness,max,min,avg);
}


app_stats(pop)
      /* Application-dependent statistics calculations called by statistics() */
      struct individual *pop;
{
}


objfunc(critter)
      /* objective function used in Goldberg's book */
      /* fitness function is f(x) = x**n,
          normalized to range between 0 and 1,
          where x is the chromosome interpreted as an
          integer, and n = 10 */

      struct individual *critter;
{
   int i;
   float lx2,x1,x2;
   float range, maxf, D=9;
   float m1,b1,m2,b2;
   float F1,F2,F3;

   lx2=(float)lchrom/2.0;

   x2=0;

   for(i=lx2+1;i<=lchrom;i++)
     x2 += (float)ithruj2int(i,i,critter->chrom);

   x1 = (float)ithruj2int(1,(int)lx2,critter->chrom);


   range = pow(2.0,lx2)-1;


   /* x1 = (float)ithruj2int(1,lchrom,critter->chrom);
    range = pow(2.0,lchrom)-1;
    */
   x1 = x1/range;
   x2 = x2/lx2;

   F1 = 0.5;
   F2 = 15.0/16.0;
   F3 = 1.0;


   m1 = -(F1/F2);
   b1 = F1;
   m2 = (F3/(1.0-F2));
   b2 = -F2*m2;


   if(x1 < F2)
     critter->fitness = ((m1*x1)+b1) + (D*x2);
   else
     critter->fitness = ((m2*x1)+b2) + (D*x2);

   maxf = ((m2*1.0)+b2) + (D*1.0);



   /* if(x1 < F2)
    critter->fitness = ((m1*x1)+b1);
    else
    critter->fitness = ((m2*x1)+b2);

    maxf = ((m2*1.0)+b2);
    */


   critter->fitness = critter->fitness/maxf;

}
```

```
/*---------------------------------------------------------------------*/
/* initial.c - functions to get things set up and initialized          */
/*---------------------------------------------------------------------*/

#include "external.h"


initialize()
     /* Initialization Coordinator */
{

  int i,j;

  /* get basic problem values from input file */
  initdata();

  /* define chromosome size in terms of machine bytes, ie  */
  /* length of chromosome in bits (lchrom)/(bits-per-byte) */
  /* chromsize must be known for malloc() of chrom pointer */
  chromsize = (lchrom/UINTSIZE);
  if(lchrom%UINTSIZE) chromsize++;

  /* malloc space for global data structures */
  initmalloc();

  for(i=0;i<popsize;i++)
    for(j=0;j<chromsize;j++)
      {
        oldpop[i].chrom[j]=0;
        newpop[i].chrom[j]=0;
      };


  /* initialize application dependent variables*/
  app_init();

  /* initialize random number generator */
  randomize();

  /* initialize global counters/values */
  nmutation = 0;
  ncross = 0;
  bestfit.fitness = 0.01;
  bestfit.generation = 0;

  /* initialize the populations and report statistics */
  initpop();
  statistics(oldpop);
  initreport();
}


initdata()
     /* data inquiry and setup */
{
  char  answer[2];

  if(numfiles == 0)
    {
      fprintf(outfp,
...);
      fprintf(outfp," Enter the initial population size ------------> ");
    }
  fscanf(infp,"%d", &popsize);

  if((popsize%4) != 0)
    popsize = popsize+(popsize%4);


  if(numfiles == 0)
```

```
    fprintf(outfp," Enter maximum population size --------------> ");
  fscanf(infp,"%d", &maxpop);



  if(numfiles == 0)
    fprintf(outfp," Enter chromosome length --------------> ");
  fscanf(infp,"%d", &lchrom);

  if(numfiles == 0)
    fprintf(outfp," Print chromosome strings? (y/n) ------> ");
  fscanf(infp,"%s",answer);
  if(strncmp(answer,"n",1) == 0) printstrings = 0;

  if(numfiles == 0)
    fprintf(outfp," Enter maximum number of generations --> ");
  fscanf(infp,"%d", &maxgen);

  if(numfiles == 0)
    fprintf(outfp," Enter crossover probability ----------> ");
  fscanf(infp,"%f", &pcross);

  if(numfiles == 0)
    fprintf(outfp," Enter mutation probability -----------> ");
  fscanf(infp,"%f", &pmutation);

  if(numfiles == 0)
    fprintf(outfp," Enter target loss -----------> ");
  fscanf(infp,"%f", &Lt);

  if(numfiles == 0)
    fprintf(outfp," Enter beta (sigmoid slope) -----------> ");
  fscanf(infp,"%f", &beta);

  if(numfiles == 0)
    fprintf(outfp," Enter maximum increase/decrease factor -----------> ");
  fscanf(infp,"%f", &gmma);


  /* any application-dependent global input */
  app_data();
}


initpop()
     /* Initialize a population at random */
{
  int j, j1, k, stop;
  unsigned mask = 1;

  for(j = 0; j < popsize; j++)
    {
      for(k = 0; k < chromsize; k++)
        {
          oldpop[j].chrom[k] = 0;
          if(k == (chromsize-1))
            stop = lchrom - (k*UINTSIZE);
          else
            stop = UINTSIZE;

          /* A fair coin toss */
          for(j1 = 1; j1 <= stop; j1++)
            {
              if(flip(0.5))
                oldpop[j].chrom[k] = oldpop[j].chrom[k]|mask;
              oldpop[j].chrom[k] = oldpop[j].chrom[k]<<1;
            }
        }
      oldpop[j].parent[0] = 0; /* Initialize parent info. */
      oldpop[j].parent[1] = 0;
```

```
        oldpop[j].xsite = 0;
        objfunc(&(oldpop[j]));   /* Evaluate initial fitness */
    }
}


initreport()
      /* Initial report */
{
  void   skip();

  skip(outfp,1);
  /*fprintf(outfp," SGA Parameters\n");
  /*fprintf(outfp," -------------------------------------------------\n");*/
  /*fprintf(outfp," Total Population size            =  %d\n",popsize);*/
  /*fprintf(outfp," Chromosome length (lchrom)        =  %d\n",lchrom);*/
  /*fprintf(outfp," Maximum # of generations (maxgen)  =  %d\n",maxgen);*/
  /*fprintf(outfp," Crossover probability (pcross)     = %f\n", pcross);*/
  /*fprintf(outfp," Mutation  probability (pmutation)  = %f\n", pmutation);*/
  skip(outfp,1);


  /* application dependant report */
  app_initreport();

  fflush(outfp);
}
```

```
/*----------------------------------------------------------------------*/
/* generate.c - create a new generation of individuals                 */
/*----------------------------------------------------------------------*/

#include "external.h"

generation()
{

  int mate1, mate2, jcross, j = 0;

  /* perform any preselection actions necessary before generation */
  preselect();

  /* select, crossover, and mutation */
  do
    {
      /* pick a pair of mates */
      mate1 = select();
      mate2 = select();

      /* Crossover and mutation */
      jcross = crossover(oldpop[mate1].chrom, oldpop[mate2].chrom,
                         newpop[j].chrom, newpop[j+1].chrom);
      mutation(newpop[j].chrom);
      mutation(newpop[j+1].chrom);

      /* Decode string, evaluate fitness, & record */
      /* parentage date on both children */
      objfunc(&(newpop[j]));
      newpop[j].parent[0] = mate1+1;
      newpop[j].xsite = jcross;
      newpop[j].parent[1] = mate2+1;
      objfunc(&(newpop[j+1]));
      newpop[j+1].parent[0] = mate1+1;
      newpop[j+1].xsite = jcross;
      newpop[j+1].parent[1] = mate2+1;

      /* Increment population index */
      j = j + 2;
    }
  while(j < (popsize-1));

}
```

```
/*----------------------------------------------------------------------*/
/* operators.c - Genetic Operators: Crossover and Mutation              */
/*----------------------------------------------------------------------*/

#include "external.h"

mutation(child)
unsigned *child;
/* Mutate an allele w/ pmutation, count # of mutations */
{
    int j, k, stop;
    unsigned mask, temp = 1;

    for(k = 0; k < chromsize; k++)
    {
        mask = 0;
        if(k == (chromsize-1))
            stop = lchrom - ((k-1)*UINTSIZE);
        else
            stop = UINTSIZE;
        for(j = 0; j < stop; j++)
        {
            if(flip(pmutation))
            {
                mask = mask|(temp<<j);
                nmutation++;
            }
        }
        child[k] = child[k]^mask;
    }
}


int crossover (parent1, parent2, child1, child2)
unsigned *parent1, *parent2, *child1, *child2;
/* Cross 2 parent strings, place in 2 child strings */
{
    int j, jcross, k;
    unsigned mask, temp;

    /* Do crossover with probability pcross */
    if(flip(pcross))
    {
        jcross = rnd(1 ,(lchrom - 1));/* Cross between 1 and l-1 */
        ncross++;
        for(k = 1; k <= chromsize; k++)
        {
            if(jcross >= (k*UINTSIZE))
            {
                child1[k-1] = parent1[k-1];
                child2[k-1] = parent2[k-1];
            }
            else if((jcross < (k*UINTSIZE)) && (jcross > ((k-1)*UINTSIZE)))
            {
                mask = 1;
                for(j = 1; j <= (jcross-1-((k-1)*UINTSIZE)); j++)
                {
                    temp = 1;
                    mask = mask<<1;
                    mask = mask|temp;
                }
                child1[k-1] = (parent1[k-1]&mask)|(parent2[k-1]&(~mask));
                child2[k-1] = (parent1[k-1]&(~mask))|(parent2[k-1]&mask);
            }
            else
            {
                child1[k-1] = parent2[k-1];
                child2[k-1] = parent1[k-1];
            }
        }
    }
    else
    {
        for(k = 0; k < chromsize; k++)
        {
            child1[k] = parent1[k];
            child2[k] = parent2[k];
        }
        jcross = 0;
    }
    return(jcross);
}
```

```
/*----------------------------------------------------------------------*/
/* memory.c - memory management routines for sga code                   */
/*----------------------------------------------------------------------*/

#include "external.h"

initmalloc()
     /* memory allocation of space for global data structures */
{
  unsigned nbytes;
  int j;

  /* memory for old and new populations of individuals */
  nbytes = maxpop*sizeof(struct individual);
  if((oldpop = (struct individual *) malloc(nbytes)) == NULL)
    nomemory(stderr,"oldpop");

  if((newpop = (struct individual *) malloc(nbytes)) == NULL)
    nomemory(stderr,"newpop");

  /* memory for chromosome strings in populations */
  nbytes = chromsize*sizeof(unsigned);
  for(j = 0; j < maxpop; j++)
    {
      if((oldpop[j].chrom = (unsigned *) malloc(nbytes)) == NULL)
        nomemory(stderr,"oldpop chromosomes");

      if((newpop[j].chrom = (unsigned *) malloc(nbytes)) == NULL)
        nomemory(stderr,"newpop chromosomes");

      if((bestfit.chrom = (unsigned *) malloc(nbytes)) == NULL)
        nomemory(stderr,"bestfit chromosome");
    }

  if((J12 = (unsigned *) malloc(nbytes)) == NULL)
        nomemory(stderr,"J12");

  if((J34 = (unsigned *) malloc(nbytes)) == NULL)
        nomemory(stderr,"J34");

  if((J1234 = (unsigned *) malloc(nbytes)) == NULL)
        nomemory(stderr,"J1234");


  /* allocate any auxiliary memory for population resizing */
  mate_memory();

  /* call to application-specific malloc() routines  */
  /* can be used to malloc memory for utility pointer */
  app_malloc();
}

freeall()
     /* A routine to free all the space dynamically allocated in initspace() */
{
  int i;

  for(i = 0; i < popsize; i++)
    {
      free(oldpop[i].chrom);
      free(newpop[i].chrom);
    }
  free(oldpop);
  free(newpop);

  /* free any auxiliary memory needed for selection */
  mate_free();

  /* call to application-specific free() routines  */
  /* can be used to free memory for utility pointer */
  app_free();
}

popsize_free()
/* A routine to free memory associated with adaptively resizing the population */
{
int i;

for(i=0;i<maxpop;i++)
{
   free(oldpop[i].chrom);
   free(newpop[i].chrom);
}
free(oldpop);
free(newpop);

mate_free();
app_free();

}

nomemory(string)
     char *string;
{
  fprintf(outfp,"malloc: out of memory making %s!!\n",string);
  exit(-1);
}
```

```
/*---------------------------------------------------------------------*/
/* statistic.c - compute the fitness statistics                       */
/*---------------------------------------------------------------------*/

#include "external.h"

statistics(pop)
/* Calculate population statistics */
struct individual *pop;
{
    int i, j;

    sumfitness = 0.0;
    min = pop[0].fitness;
    max = pop[0].fitness;

    /* Loop for max, min, sumfitness */
    for(j = 0; j < popsize; j++)
    {
        sumfitness = sumfitness + pop[j].fitness;              /* Accumulate */
        if(pop[j].fitness > max) max = pop[j].fitness;        /* New maximum */
        if(pop[j].fitness < min) min = pop[j].fitness;        /* New minimum */

        /* new global best-fit individual */
        if(pop[j].fitness > bestfit.fitness)
          {
            for(i = 0; i < chromsize; i++)
              bestfit.chrom[i]      = pop[j].chrom[i];

            bestfit.fitness    = pop[j].fitness;
            bestfit.generation = gen;
          }
      }

    /* Calculate average */
    avg = sumfitness/popsize;

    /* get application dependent stats */
    app_stats(pop);
}
```

```
/*--------------------------------------------------------------------*/
/* report.c - generation report files                                  */
/*--------------------------------------------------------------------*/

#include "external.h"

report()
/* Write the population report */
{
    void    repchar(), skip();
    int     writepop(), writestats();

    repchar(outfp,"-",LINELENGTH);
    skip(outfp,1);

    if(printstrings == 1)
    {
        repchar(outfp," ",((LINELENGTH-17)/2));
        fprintf(outfp,"Population Report\n");
        fprintf(outfp, "Generation %3d", gen);
        repchar(outfp," ",(LINELENGTH-28));
        fprintf(outfp, "Generation %3d Popsize=%d \n", (gen+1), popsize);
        fprintf(outfp,"num   string ");
        repchar(outfp," ",lchrom-5);
        fprintf(outfp,"fitness    parents xsite  ");
        fprintf(outfp,"string ");
        repchar(outfp," ",lchrom-5);
        fprintf(outfp,"fitness\n");
        repchar(outfp,"-",LINELENGTH);
        skip(outfp,1);

        /* write out string info from all nodes        */
        /* in order from lowest to highest node number */
        writepop(outfp);
        repchar(outfp,"-",LINELENGTH);
        skip(outfp,1);

    }

    /* write the summary statistics in global mode */
    fprintf(outfp,"Generation %d Accumulated Statistics: \n",
            gen);

    fprintf(outfp,"Population size = %d\n", popsize);
    fprintf(outfp,"Total Crossovers = %d, Total Mutations = %d\n",
                  ncross,nmutation);
    fprintf(outfp,"min = %f   max = %f   avg = %f   sum = %f\n",
                min,max,avg,sumfitness);
    fprintf(outfp,"Global Best Individual so far, Generation %d:\n",
                bestfit.generation);
    fprintf(outfp,"Fitness = %f: ", bestfit.fitness);
    writechrom((&bestfit)->chrom);
    skip(outfp,1);
    repchar(outfp,"-",LINELENGTH);
    skip(outfp,1);

    /* application dependent report */
    app_report();
}


writepop()
{
    struct individual *pind;
    int j;

    for(j=0; j<popsize; j++)
    {
```

report

writepop

```
        fprintf(outfp,"%3d)   ",j+1);

        /* Old string */
        pind = &(oldpop[j]);
        writechrom(pind->chrom);
        fprintf(outfp,"  %8f | ", pind->fitness);

        /* New string */
        pind = &(newpop[j]);
        fprintf(outfp,"(%2d,%2d)   %2d   ",
        pind->parent[0], pind->parent[1], pind->xsite);
        writechrom(pind->chrom);
        fprintf(outfp,"  %8f\n", pind->fitness);
    }
}


writechrom(chrom)
/* Write a chromosome as a string of ones and zeroes          */
/* note that the most significant bit of the chromosome is the */
/* RIGHTMOST bit, not the leftmost bit, as would be expected... */
unsigned *chrom;
{
    int j, k, stop;
    unsigned mask = 1, tmp;

    for(k = 0; k < chromsize; k++)
    {
        tmp = chrom[k];
        if(k == (chromsize-1))
            stop = lchrom - (k*UINTSIZE);
        else
            stop = UINTSIZE;

        for(j = 0; j < stop; j++)
        {
            if(tmp&mask)
                fprintf(outfp,"1");
            else
                fprintf(outfp,"0");
            tmp = tmp>>1;
        }
    }
}
```

```
/*----------------------------------------------------------------------*/
/* random.c - contains random number generator and related utilities,     */
/* including advance_random, warmup_random, random, randomize, flip, and rnd  */
/*----------------------------------------------------------------------*/

#include <math.h>
#include "external.h"

/* variables are declared static so that they cannot conflict with names of   */
/* other global variables in other files.  See K&R, p 80, for scope of static */
static double oldrand[55];                      /* Array of 55 random numbers */
static int jrand;                                       /* current random number */
static double rndx2;                         /* used with random normal deviate */
static int rndcalcflag;                      /* used with random normal deviate */

advance_random()
/* Create next batch of 55 random numbers */
{
    int j1;
    double new_random;

    for(j1 = 0; j1 < 24; j1++)
    {
        new_random = oldrand[j1] - oldrand[j1+31];
        if(new_random < 0.0) new_random = new_random + 1.0;
        oldrand[j1] = new_random;
    }
    for(j1 = 24; j1 < 55; j1++)
    {
        new_random = oldrand [j1] - oldrand [j1-24];
        if(new_random < 0.0) new_random = new_random + 1.0;
        oldrand[j1] = new_random;
    }
}


int flip(prob)
/* Flip a biased coin - true if heads */
float prob;
{
    float randomperc();

    if(randomperc() <= prob)
        return(1);
    else
        return(0);
}


initrandomnormaldeviate()
/* initialization routine for randomnormaldeviate */
{
    rndcalcflag = 1;
}


double noise(mu ,sigma)
/* normal noise with specified mean & std dev: mu & sigma */
double mu, sigma;
{
    double randomnormaldeviate();

    return((randomnormaldeviate()*sigma) + mu);
}


randomize()
/* Get seed number for random and start it up */
{
    float randomseed;
```

advance_random

initrandomnormaldeviate

ndomize

```
    int j1;

    for(j1=0; j1<=54; j1++) oldrand[j1] = 0.0;
    jrand=0;

    if(numfiles == 0)
    {
        do
        {
            if(numfiles == 0)
              fprintf(outfp," Enter random number seed, 0.0 to 1.0 -> ");
            fscanf(infp,"%f", &randomseed);
        }
        while((randomseed < 0.0) || (randomseed > 1.0));
    }
    else
    {
        fscanf(infp,"%f", &randomseed);
    }

    warmup_random(randomseed);
}


double randomnormaldeviate()
/* random normal deviate after ACM algorithm 267 / Box-Muller Method */
{
    float randomperc();
    double t, rndx1;
    /*double sqrt(),sin(),cos(),log();*/

    if(rndcalcflag)
    {
        rndx1 = sqrt(- 2.0*log((double) randomperc()));
        t = 6.2831853072 * (double) randomperc();
        rndx2 = sin(t);
        rndcalcflag = 0;
        return(rndx1 * cos(t));
    }
    else
    {
        rndcalcflag = 1;
        return(rndx2);
    }
}


float randomperc()
/* Fetch a single random number between 0.0 and 1.0 - Subtractive Method */
/* See Knuth, D. (1969), v. 2 for details */
/* name changed from random() to avoid library conflicts on some machines*/
{
    jrand++;
    if(jrand >= 55)
    {
        jrand = 1;
        advance_random();
    }
    return((float) oldrand[jrand]);
}


int rnd(low, high)
/* Pick a random integer between low and high */
int low,high;
{
    int i;
    float randomperc();

    if(low >= high)
```

```
        i = low;
    else
    {
        i = (randomperc() * (high - low + 1)) + low;
        if(i > high) i = high;
    }
    return(i);
}


float rndreal(lo ,hi)
/* real random number between specified limits */
float lo, hi;
{
    return((randomperc() * (hi - lo)) + lo);
}


warmup_random(random_seed)
/* Get random off and running */
float random_seed;
{
    int j1, ii;
    double new_random, prev_random;

    oldrand[54] = random_seed;
    new_random = 0.000000001;
    prev_random = random_seed;
    for(j1 = 1 ; j1 <= 54; j1++)
    {
        ii = (21*j1)%54;
        oldrand[ii] = new_random;
        new_random = prev_random-new_random;
        if(new_random<0.0) new_random = new_random + 1.0;
        prev_random = oldrand[ii];
    }

    advance_random();
    advance_random();
    advance_random();

    jrand = 0;
}
```

```
/*---------------------------------------------------------------------*/
/* utility.c - utility routines, contains copyright,  repchar, skip          */
/*---------------------------------------------------------------------*/

#include "external.h"

void copyright()
{
     void repchar(), skip();
     int iskip;
     int ll = 59;

     iskip = (LINELENGTH - ll)/2;
     skip(outfp,1);
/*  repchar(outfp," ",iskip); repchar(outfp,"-",ll); skip(outfp,1); */
/*  repchar(outfp," ",iskip); */
/*  fprintf(outfp,"|       SGA-C (v1.1) - A Simple Genetic Algorithm        |\n"); */
/*  repchar(outfp," ",iskip); */
/*  fprintf(outfp,"|    (c) David E. Goldberg 1986, All Rights Reserved      |\n"); */
/*  repchar(outfp," ",iskip); */
/*  fprintf(outfp,"|       C version by Robert E. Smith, U. of Alabama        |\n"); */
/*  repchar(outfp," ",iskip); */
/*  fprintf(outfp,"|    v1.1 modifications by Jeff Earickson, Boeing Company  |\n"); */
/*  repchar(outfp," ",iskip); repchar(outfp,"-",ll); skip(outfp,2); */
}


void repchar (outfp,ch,repcount)
/* Repeatedly write a character to stdout */
FILE *outfp;
char *ch;
int repcount;
{
     int j;

     for (j = 1; j <= repcount; j++) fprintf(outfp,"%s", ch);
}


void skip(outfp,skipcount)
/* Skip skipcount lines */
FILE *outfp;
int skipcount;
{
     int j;

     for (j = 1; j <= skipcount; j++) fprintf(outfp,"\n");
}


int ithruj2int(i,j,from)
/* interpret bits i thru j of a individual as an integer      */
/* j MUST BE greater than or equal to i AND j-i < UINTSIZE-1  */
/* from is a chromosome, represented as an array of unsigneds */
int i,j;
int *from;
{
     unsigned mask, temp;
     int bound_flag;
     int iisin, jisin;
     int i1, j1, out;

     if(j < i)
     {
         fprintf(stderr,"Error in ithruj2int: j < i\n");
         exit(-1);
     }
     if(j-i+1 > UINTSIZE)
     {
         fprintf(stderr,"Error in ithruj2int: j-i+1 > UINTSIZE\n");
```

```
        exit(-1);
    }

    iisin = i/UINTSIZE;
    jisin = j/UINTSIZE;

    i1 = i - (iisin*UINTSIZE);
    j1 = j - (jisin*UINTSIZE);

        if(i1 == 0)
        {
                iisin = iisin-1;
                i1 = i - (iisin*UINTSIZE);
        };

        if(j1 == 0)
        {
                jisin = jisin-1;
                j1 = j - (jisin*UINTSIZE);
        };

    /* check if bits fall across a word boundary */
    if(iisin == jisin)
        bound_flag = 0;
    else
        bound_flag = 1;

    if(bound_flag == 0)
    {
        mask = 1;
        mask = (mask<<(j1-i1+1))-1;
        mask = mask<<(i1-1);
        out = (from[iisin]&mask)>>(i1-1);
        return(out);
    }
    else
    {
        mask = 1;
        mask = (mask<<j1+1)-1;
        temp = from[jisin]&mask;
        mask = 1;
        mask = (mask<<(UINTSIZE-i1+1))-1;
        mask = mask<<(i1-1);
        out = ((from[iisin]&mask)>>(i1-1)) | temp<<(UINTSIZE-i1);
        return(out);
    }
}
```

```
#

# Makefile for Simple Genetic Algorithm C code
# Adaptive population sizing version
#
#####################################
# Define implicit compilation rules #
#####################################

# uncomment following lines for UNIX computers
# nominally tested on Sun, Cray UNICOS, and Vax Ultrix systems
CC=cc
LDLIBS= -lm


SRCCODE=main.c adaptpop.c app.c generate.c initial.c memory.c\
        operators.c random.c report.c \
        statistic.c utility.c

OBJECTS=main.o adaptpop.o app.o generate.o initial.o\
        memory.o operators.o random.o report.o\
        statistic.o utility.o

sga:    $(OBJECTS)
        $(CC) $(CFLAGS) -o $@ $(OBJECTS) $(LDLIBS)
        chmod 755 $@

$(OBJECTS): sga.h external.h

listing:
        cat *.h > sga.list
        cat $(SRCCODE) >> sga.list

clean:
        rm -f sga
        rm -f *.on sga.sym
        rm -f *.o
```