## 15.7 Classification of Genetic Algorithm

There exist wide variety of GAs including simple and general GAs discussed in Sections 15.4 and 15.5, respectively. Some other variants of GA are discussed below.

### 15.7.1 Messy Genetic Algorithms

In a "classical" GA, the genes are encoded in a fixed order. The meaning of a single gene is determined by its position inside the string. We have seen in the previous chapter that a GA is likely to converge well if the optimization task can be divided into several short building blocks. What, however, happens if the coding is chosen such that couplings occur between distant genes? Of course, one-point crossover tends to disadvantage long schemata (even if they have low order) over short ones.

Messy GAs try to overcome this difficulty by using a variable-length, position-independent coding. The key idea is to append an index to each gene which allows identifying its position. A gene, therefore, is no longer represented as a single allele value and a fixed position, but as a pair of an index and an allele. Figure 15-8 (A) shows how this "messy" coding works for a string of length 6.

Since with the help of the index we can identify the genes uniquely, genes may be swapped arbitrarily without changing the meaning of the string. With appropriate genetic operations, which also change the order of the pairs, the GA could possibly group coupled genes together automatically.

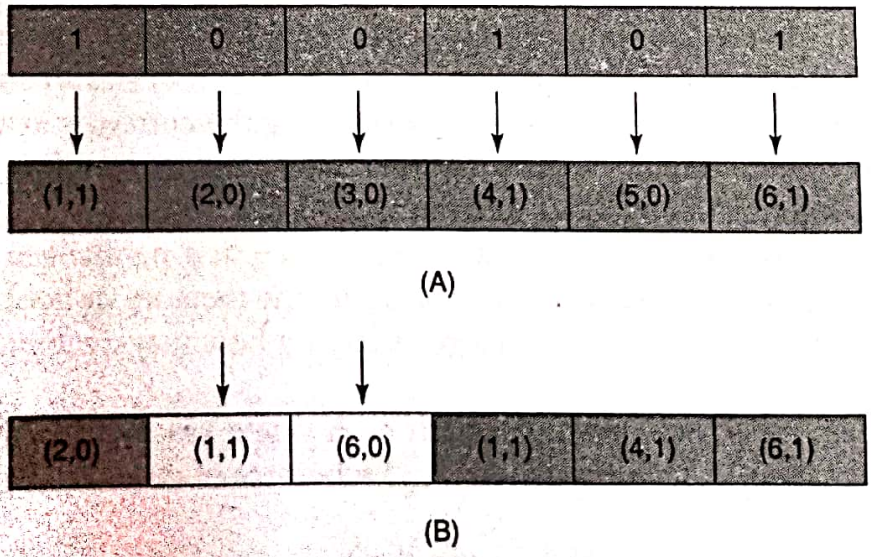



**Figure 15-8** (A) Messy coding and (B) positional preference: Genes with indices 1 and 6 occur twice, the first occurrences are used.

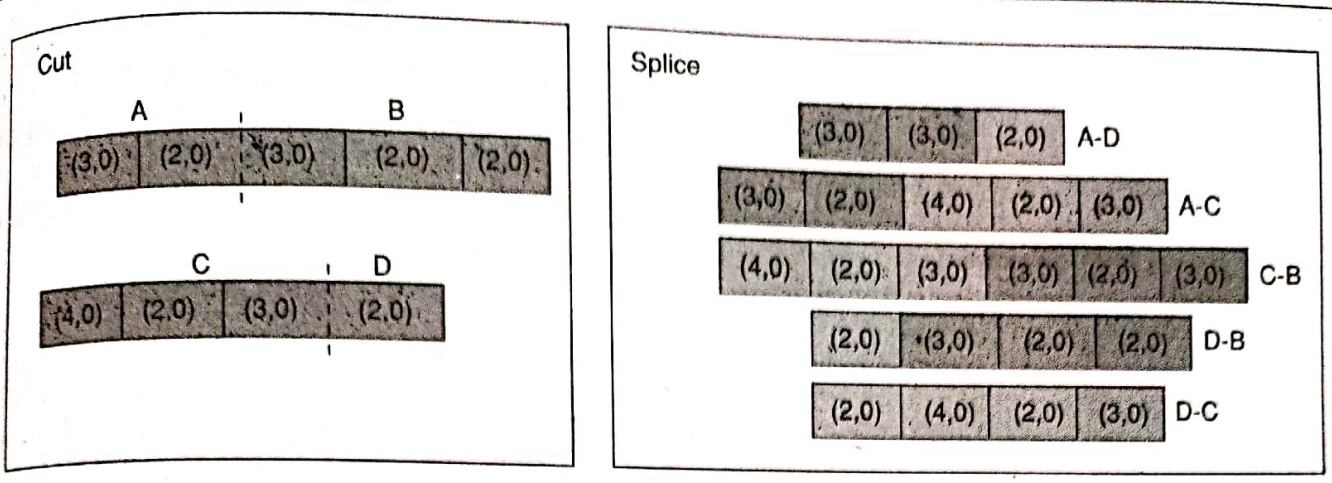



**Figure 15-9** The cut and splice operation.

Owing to the free arrangement of genes and the variable length of the encoding, we can, however, run into problems, which do not occur, in a simple GA. First of all, it can happen that there are two entries in a string, which correspond to the same index but have conflicting alleles. The most obvious way to overcome this "over-specification" is positional preference – the first entry, which refers to a gene, is taken. Figure 15-8(B) shows an example. The reader may have observed that the genes with indices 3 and 5 do not occur at all in the example in Figure 15-8(B). This problem of "under specification" is more complicated and its solution is not as obvious as for over-specification. Of course, a lot of variants are reasonable.

One approach could be to check all possible combinations and to take the best one (for $k$ missing genes, there are $2^k$ combinations). With the objective to reduce this effort, Goldberg et al. have suggested to use so-called competitive templates for finding specifications for k missing genes. It is nothing else than applying a local hill climbing method with random initial value to the k missing genes.

While messy GAs usually work with the same mutation operator as simple GAs (every allele is altered with a low probability pM), the crossover operator is replaced by a more general cut and splice operator which also allows to mate parents with different lengths. The basic idea is to choose cut sites for both parents independently and to splice the four fragments. Figure 15-9 shows an example.

## 15.7.2 Adaptive Genetic Algorithms

Adaptive GAs are those whose parameters, such as the population size, the crossing over probability, or the mutation probability, are varied while the GA is running. A simple variant could be the following: The mutation rate is changed according to changes in the population – the longer the population does not improve, the higher the mutation rate is chosen. Vice versa, it is decreased again as soon as an improvement of the population occurs.

### 15.7.2.1 Adaptive Probabilities of Crossover and Mutation

It is essential to have two characteristics in GAs for optimizing multimodal functions. The first characteristic is the capacity to converge to an optimum (local or global) after locating the region containing the optimum. The second characteristic is the capacity to explore new regions of the solution space in search of the global optimum. The balance between these characteristics of the GA is dictated by the values of $p_m$ and $p_c$, and the type of crossover employed. Increasing values of $p_m$ and $p_c$ promote exploration at the expense of

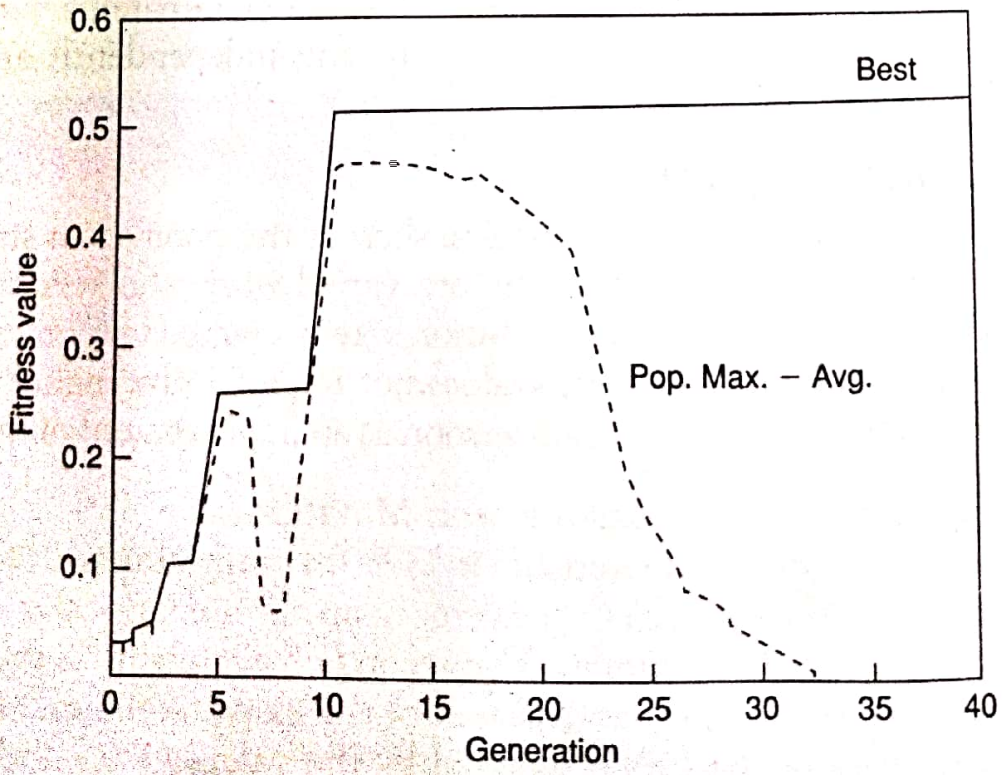
exploitation. Moderately large values of $p_c$ (in the range 0.5–1.0) and small values of $p_m$ (in the range 0.001–0.05) are commonly employed in GA practice. In this approach, we aim at achieving this tradeoff between exploration and exploitation in a different manner, by varying $p_c$ and $p_m$ adaptively in response to the fitness values of the solutions; $p_c$ and $p_m$ are increased when the population tends to get stuck at a local optimum and are decreased when the population is scattered in the solution space.

### 15.7.2.2 Design of Adaptive $p_c$ and $p_m$

To vary $p_c$ and $p_m$ adaptively for preventing premature convergence of the GA to a local optimum, it is essential to identify whether the GA is converging to an optimum. One possible way of detecting is to observe average fitness value $\bar{f}$ of the population in relation to the maximum fitness value $f_{max}$ of the population. The value $f_{max} - \bar{f}$ is likely to be less for a population that has converged to an optimum solution than that for a population scattered in the solution space. We have observed the above property in all our experiments with GAs, and Figure 15-10 illustrates the property for a typical case. In Figure 15-10 we notice that $f_{max} - \bar{f}$ decreases when the GA converges to a local optimum with a fitness value of 0.5. (The globally optimal solution has a fitness value of 1.0.) We use the difference in the average and maximum fitness value, $f_{max} - \bar{f}$, as a yardstick for detecting the convergence of the GA. The values of $p_c$ and $p_m$ are varied depending on the value of $f_{max} - \bar{f}$. Since $p_c$ and $p_m$ have to be increased when the GA converges to a local optimum, i.e. when $f_{max} - \bar{f}$ decreases, $p_c$ and $p_m$ will have to be varied inversely with $f_{max} - \bar{f}$. The expressions that we have chosen for $p_c$ and $p_m$ are of the form

$$p_c = k_1/(f_{max} - \bar{f})$$
$$p_m = k_2/(f_{max} - \bar{f})$$

It has to be observed in the above expressions that $p_c$ and $p_m$ do not depend on the fitness value of any particular solution, and have the same values for all the solution of the



population. Consequently, solutions with high fitness values as well as solutions with low fitness values are subjected to the same levels of mutation and crossover. When a population converges to a globally optimal solution (or even a locally optimal solution), $p_c$ and $p_m$ increase and may cause the disruption of the near-optimal solutions. The population may never converge to the global optimum. Though we may prevent the GA from getting stuck at a local optimum, the performance of the GA (in terms of the generations required for convergence) will certainly deteriorate.

To overcome the above-stated problem, we need to preserve "good" solutions of the population. This can be achieved by having lower values of $p_c$ and $p_m$ for high fitness solutions and higher values of $p_c$ and $p_m$ for low fitness solutions. While the high fitness solutions aid in the convergence of the GA, the low fitness solutions prevent the GA from getting stuck at a local optimum. The value of $p_m$ should depend not only on $f_{max} - \bar{f}$ but also on the fitness value $f$ of the solution. Similarly, $p_c$ should depend on the fitness values of both the parent solutions. The closer $f$ is to $f_{max}$ the smaller $p_m$ should be, i.e. $p_m$ should vary directly as $f_{max} - f$. Similarly, $p_c$ should vary directly as $f_{max} - f'$, where $f'$ is the larger of the fitness value of the solutions to be crossed. The expressions for $p_c$ and $p_m$ now take the forms

$$p_c = k_1[(f_{max} - f')/(f_{max} - \bar{f})], \; k_1 \leq 1.0$$
$$p_m = k_2[(f_{max} - f)/(f_{max} - \bar{f})], \; k_2 \leq 1.0$$

(Here $k_1$ and $k_2$ have to be less than 1.0 to constrain $p_c$ and $p_m$ to the range 0.0–1.0.)

Note that $p_c$ and $p_m$ are zero for the solution with the maximum fitness. Also $p_c = k_1$ for a solution with $f' = \bar{f}$, and $p_m = k_2$ for a solution with $f = \bar{f}$. For solution with subaverage fitness values, i.e. $f < \bar{f}$, $p_c$ and $p_m$ might assume values larger than 1.0. To prevent the overshooting of $p_c$ and $p_m$ beyond 1.0, we also have the following constraints:

$$p_c = k_3, \; f' \leq \bar{f}$$
$$p_m = k_4, \; f \leq \bar{f}$$

where $k_3, k_4 \leq 1.0$.

### 15.7.2.3 Practical Considerations and Choice of Values for $k_1$, $k_2$, $k_3$ and $k_4$

In the previous subsection, we saw that for a solution with the maximum fitness value $p_c$ and $p_m$ are both zero. The best solution in a population is transferred undisrupted into the next generation. Together with the selection mechanism, this may lead to an exponential growth of the solution in the population and may cause premature convergence. To overcome the above-stated problem, we introduce a default mutation rate (of 0.005) for every solution in the Adaptive Genetic Algorithm (AGA).

We now discuss the choice of values for $k_1, k_2, k_3$ and $k_4$. For convenience, the expressions for $p_c$ and $p_m$ are given as

$$p_c = k_1(f_{max} - f')/(f_{max} - \bar{f}), \; f \geq \bar{f}$$
$$p_c = k_3, \; f < \bar{f}$$
$$p_m = k_2(f_{max} - f')/(f_{max} - \bar{f}), \; f \geq \bar{f}$$
$$p_m = k_4, \; f < \bar{f}$$

where $k_1, k_2, k_3, k_4 \leq 1.0$.

It has been well established in GA literature that moderately large values of $p_c$ ($0.5 < p_c < 1.0$) and small values of $p_m$ ($0.001 < p_m < 0.05$) are essential for the successful working of GAs. The moderately large values of $p_c$ promote the extensive recombination of schemata, while small values of $p_m$ are necessary to prevent the disruption of the solutions. These guidelines, however, are useful and relevant when the values of $p_c$ and $p_m$ do not vary.

One of the goals of the approach is to prevent the GA from getting stuck at a local optimum. To achieve this goal, we employ solutions with sub average fitnesses to search the search space for the region containing the global optimum. Such solutions need to be completely disrupted, and for this purpose we use a value of 0.5 for $k_4$. Since solutions with a fitness value of $f$ should also be disrupted completely, we assign a value of 0.5 to $k_2$ as well.

Based on similar reasoning, we assign $k_1$ and $k_3$ a value of 1.0. This ensures that all solutions with a fitness value less than or equal to $f$ compulsorily undergo crossover. The probability of crossover decreases as the fitness value (maximum of the fitness values of the parent solutions) tends to $f_{max}$ and is 0.0 for solutions with a fitness value equal to $f_{max}$.

### 15.7.3 Hybrid Genetic Algorithms

As they use the fitness function only in the selection step, GAs are blind optimizers which do not use any auxiliary information such as derivatives or other specific knowledge about the special structure of the objective function. If there is such knowledge, however, it is unwise and inefficient not to make use of it. Several investigations have shown that a lot of synergism lies in the combination of genetic algorithms and conventional methods.

The basic idea is to divide the optimization task into two complementary parts. The GA does the coarse, global optimization while local refinement is done by the conventional method (e.g. gradient-based, hill climbing, greedy algorithm, simulated annealing, etc.). A number of variants are reasonable:

1. The GA performs coarse search first. After the GA is completed, local refinement is done.
2. The local method is integrated in the GA. For instance, every $K$ generations, the population is doped with a locally optimal individual.
3. Both methods run in parallel: All individuals are continuously used as initial values for the local method. The locally optimized individuals are re-implanted into the current generation.

In this section a novel optimization approach is used that switches between global and local search methods based on the local topography of the design space. The global and local optimizers work in concert to efficiently locate quality design points better than either could alone. To determine when it is appropriate to execute a local search, some characteristics about the local area of the design space need to be determined. One good source of information is contained in the population of designs in the GA. By calculating the relative homogeneity of the population we can get a good idea of whether there are multiple local optima located within this local region of the design space.

To quantify the relative homogeneity of the population in each subspace, the coefficient of variance of the objective function and design variables is calculated. The coefficient of

variance is a normalized measure of variation, and unlike the actual variance, is independent of the magnitude of the mean of the population. A high coefficient of variance could be an indication that there are multiple local optima present. Very low values could indicate that the GA has converged to a small area in the design space, warranting the use of a local search algorithm to find the best design within this region.

By calculating the coefficient of variance of the both the design variables and the objective function as the optimization progresses, it can also be used as a criterion to switch from the global to the local optimizer. As the variance of the objective values and design variables of the population increases, it may indicate that the optimizer is exploring new areas of the design space or hill climbing. If the variance is decreasing, the optimizer may be converging toward local minima and the optimization process could be made more efficient by switching to a local search algorithm.

The second method, regression analysis, used in this section helps us determine when to switch between the global and local optimizer. The design data present in the current population of the GA can be used to provide information as to the local topography of the design space by attempting to fit models of various order to it.

The use of regression analysis to augment optimization algorithms is not new. In problems in which the objective function or constraints are computationally expensive, approximations to the design space are created by sampling the design space and then using regression or other methods to create a simple mathematical model that closely approximates the actual design space, which may be highly nonlinear. The design space can then be explored to find regions of good designs or optimized to improve the performance of the system using the predictive surrogate approximation models instead of the computationally expensive analysis code, resulting in large computational savings. The most common regression models are linear and quadratic polynomials created by performing ordinary least squares regression on a set of analysis data.

To make clear the use of regression analysis in this way, consider Figure 15-11, which represents a complex design space. Our goal is to minimize this function, and as a first step the GA is run. Suppose that after a certain number of generations the population consists of

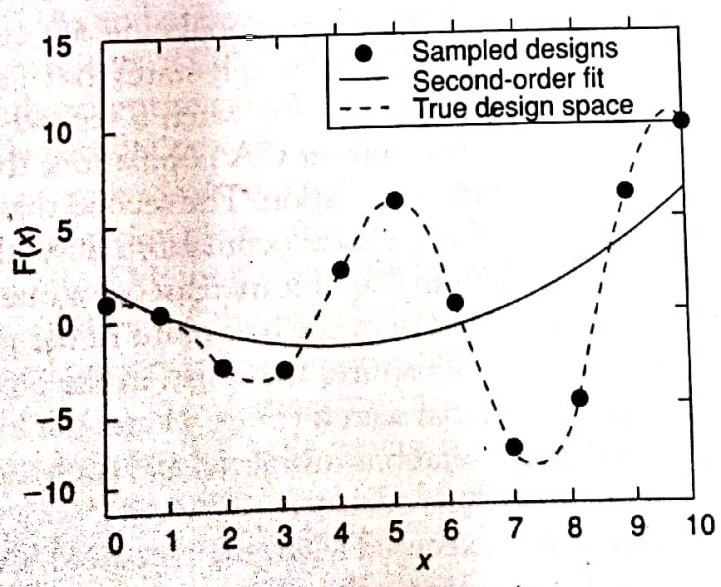



**Figure 15-11** Approximating multiple modes with a second-order model.

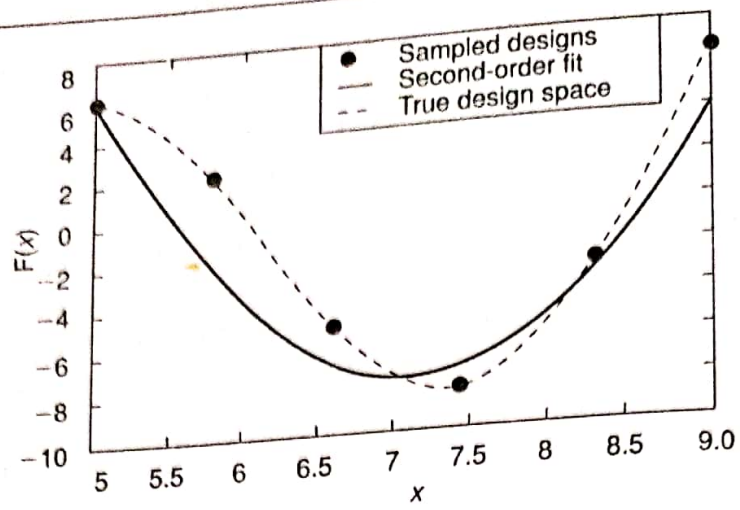



**Figure 15-12** Approximating a single mode with a second-order model.

the sampled points shown in the figure. Since the population of the GA is spread throughout the design space, having yet to converge into one of the local minima, it seems logical to continue the GA for additional generations. Ideally, before the local optimizer is run it would be beneficial to have some confidence that its starting point is somewhere within the mode that contains the optimum. Fitting a second-order response surface to the data and noting the large error (the R2 value is 0.13), there is a clear indication that the GA is currently exploring multiple modes in the design space.

In Figure 15-12, the same design space is shown but after the GA has begun to converge into the part of the design space containing the optimal design. Once again a second-order approximation is fit to GA's population. The dotted line connects the points predicted by the response surface. Note how much smaller the error is in the approximation (the R2 is 0.96), which is a good indication that the GA is currently exploring a single mode within the design space. At this point, the local optimizer can be made to quickly converge to the best solution within this area of the design space, thereby avoiding the slow convergence properties of the GA.

After each generation of the global optimizer, the values of the coefficient of determination and the coefficient of variance of the entire population are compared with the designer specified threshold levels. The first threshold simply states that if coefficient of determination of the population exceeds a designer set value when a second-order regression analysis is performed on the design data in the current GA population, then a local search is started from the current 'best design' in the population. The second threshold is based on the value of the coefficient of variance of the entire population. This threshold is also set by the designer and can range upwards from 0%. If it increases at a rate greater than the threshold level then a local search is executed from the best point in the population.

The flowchart in Figure 15-13 illustrates the stages in the algorithm. The algorithm can switch repeatedly between the global search (Stage 1) and the local search (Stage 2) during execution. In Stage 1, the global search is initialized and then monitored. This is also where the regression and statistical analysis occurs.

In Stage 2 the local search is executed when the threshold levels are exceeded, and then this solution is passed back and integrated into the global search. The algorithm stops when convergence is achieved for the global optimization algorithm.

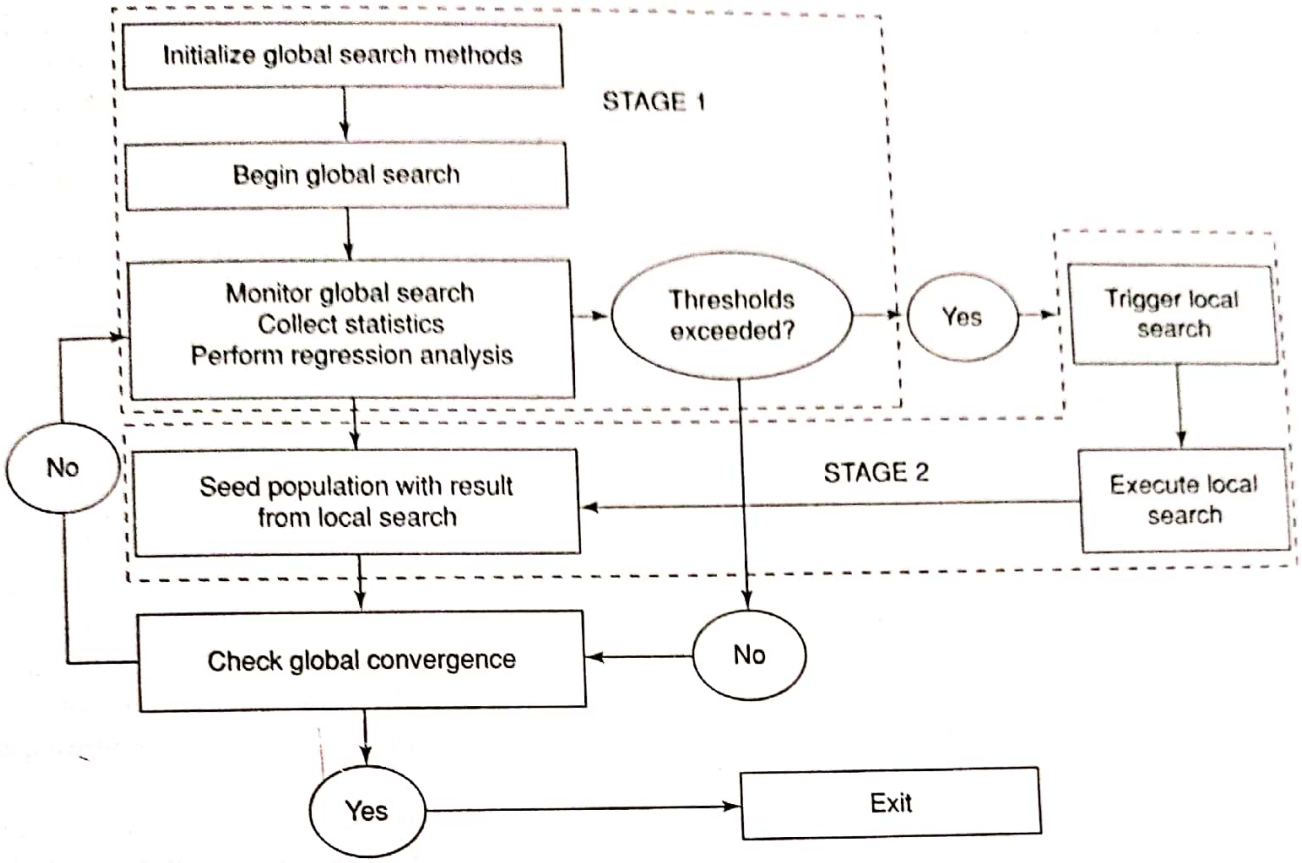



**Figure 15-13** Steps in two-stage hybrid optimization approach.

### 15.7.4 *Parallel Genetic Algorithm*

GAs are powerful search techniques that are used successfully to solve problems in many different disciplines. Parallel GAs (PGAs) are particularly easy to implement and promise substantial gains in performance. As such, there has been extensive research in this field. The section describes some of the most significant problems in modeling and designing multi-population PGAs and presents some recent advancements.

One of the major aspects of GA is their ability to be parallelized. Indeed, because natural evolution deals with an entire population and not only with particular individuals, it is a remarkably highly parallel process. Except in the selection phase, during which there is competition between individuals, the only interactions between members of the population occur during the reproduction phase, and usually, no more than two individuals are necessary to engender a new child. Otherwise, any other operations of the evolution, in particular the evaluation of each member of the population, can be done separately. So, nearly all the operations in a genetic algorithm are implicitly parallel.

PGAs simply consist in distributing the task of a basic GA on different processors. As those tasks are implicitly parallel, little time will be spent on communication; and thus, the algorithm is expected to run much faster or to find more accurate results.

It has been established that GA's efficiency to find optimal solution is largely determined by the population size. With a larger population size, the genetic diversity increases, and so the algorithm is more likely to find a global optimum! A large population requires more

memory to be stored; it has also been proved that it takes a longer time to converge. If $n$ is the population size, the convergence is expected after $n \log(n)$ function evaluations.

The use of today's new parallel computers not only provides more storage space but also allows the use of several processors to produce and evaluate more solutions in a smaller amount of time. By parallelizing the algorithm, it is possible to increase population size, reduce the computational cost, and so improve the performance of the GA.

Probably the first attempt to map GAs to existing parallel computer architectures was made in 1981 by John Grefenstette. But obviously today, with the emergence of new high-performance computing (HPC), PGA is really a flourishing area. Researchers try to improve performance of GAs. The stake is to show that GAs are one of the best optimization methods to be used with HPC.

#### 15.7.4.1 Global Parallelization

The first attempt to parallelize GAs simply consists of global parallelization. This approach tries to explicitly parallelize the implicit parallel tasks of the 'sequential' GA. The nature of the problems remains unchanged. The algorithm still manipulates a single population where each individual can mate with any other, but the breeding of new children and/or their evaluation are now made in parallel. The basic idea is that different processors can create new individuals and compute their fitness in parallel almost without any communication among each other.

To start with, doing the evaluation of the population in parallel is something really simple to implement. Each processor is assigned a subset of individuals to be evaluated. For example, on a shared memory computer, individuals could be stored in shared memory, so that each processor can read the chromosomes assigned and can write back the result of the fitness computation. This method only supposes that the GA works with a generational update of the population. Of course, some synchronization is needed between generations.

Generally, most of the computational time in a GA is spent calling the evaluation function. The time spent in manipulating the chromosomes during the selection or recombination phase is usually negligible. By assigning to each processor a subset of individuals to evaluate, a speed-up proportional to the number of processors can be expected if there is a good load balancing between them. However, load balancing should not be a problem as generally the time spent for the evolution of an individual does not really depend on the individual. A simple dynamic scheduling algorithm is usually enough to share the population between each processor equally.

On a distributed memory computer, we can store the population in one 'master' processor responsible for sending the individuals to the other processors, i.e. 'slaves'. The master processor is also responsible for collecting the result of the evaluation. A drawback of this distributed memory implementation is that a bottleneck may occur when slaves are idle while only the master is working. But a simple and good use of the master processor can improve the load balancing by distributing individuals dynamically to the slave processors when they finish their jobs.

A further step could consist in applying the genetic operators in parallel. In fact, the interaction inside the population only occurs during selection. The breeding, involving only two individuals to generate the offspring, could easily be done simultaneously over $n/2$ pairs of individuals. But it is not that clear if it worth doing so. Crossover is usually very simple and not so time-consuming; the point is not that too much time will be lost

during the communication, but that the time gain in the algorithm will be almost nothing compared to the effort produced to change the code.

This kind of global parallelization simply shows how easy it can be to transpose any GA onto a parallel machine and how a speed-up sublinear to the number of processors may be expected.

### 15.7.4.2 *Classification of Parallel GAs*

The basic idea behind most parallel programs is to divide a task into chunks and to solve the chunks simultaneously using multiple processors. This divide-and-conquer approach can be applied to GAs in many different ways, and the literature contains many examples of successful parallel implementations. Some parallelization methods use a single population, while others divide the population into several relatively isolated subpopulations. Some methods can exploit massively parallel computer architectures, while others are better suited to multicomputers with fewer and more powerful processing elements.

There are three main types of PGAs:

1. global single-population master-slave GAs,
2. single-population fine-grained,
3. multiple-population coarse-grained GAs.

In a master-slave GA there is a single panmictic population (just as in a simple GA), but the evaluation of fitness is distributed among several processors (see Figure 15-14). Since in this type of PGA, selection and crossover consider the entire population it is also known as global PGA. Fine-grained PGAs are suited for massively parallel computers and consist of one spatially structured population. Selection and mating are restricted to a small neighborhood, but neighborhoods overlap permitting some interaction among all the individuals (see Figure 15-15 for a schematic of this class of GAs). The ideal case is to have only one individual for every processing element available.

Multiple-population (or multiple-deme) GAs are more sophisticated, as they consist in several subpopulations which exchange individuals occasionally (Figure 15-16 has a schematic). This exchange of individuals is called migration and, as we shall see in later sections, it is controlled by several parameters. Multiple-deme GAs are very popular, but also are the class of PGAs which is most difficult to understand, because the effects of migration are not fully understood. Multiple-deme PGAs introduce fundamental changes in the operation of the GA and have a different behavior than simple GAs.

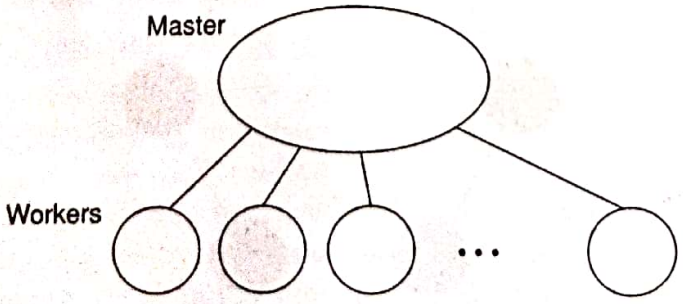



**Figure 15-14** A schematic of a master-slave PGA. The master stores the population, executes GA operations and distributes individuals to the slaves. The slaves only evaluate the fitness of the individuals.