Genetic-Algorithm Programming Environments

José L. Ribeiro Filho and Philip C. Treleaven, University College London

Cesare Alippi, Politecnico di Milano

This review classifies genetic-algorithm environments into application-oriented systems, algorithm-oriented systems, and toolkits. It also presents detailed case studies of leading environments.

Evolution is a remarkable problem-solving machine. First proposed by John Holland in 1975,[1] genetic algorithms are an attractive class of computational models that mimic natural evolution to solve problems in a wide variety of domains. Holland also developed the concept of classifier systems, a machine learning technique using induction systems with a genetic component.[2] Holland's goal was twofold: to explain the adaptive process of natural systems and to design computing systems embodying their important mechanisms. Pioneering work by Holland,[1] Goldberg,[2] DeJong,[3] Grefenstette,[4] Davis,[5] Mühlenbein,[6] and others is fueling the spectacular growth of GAs.

GAs are particularly suitable for solving complex optimization problems and hence for applications that require adaptive problem-solving strategies. In addition, GAs are inherently parallel, since their search for the best solution is performed over genetic structures (building blocks) that can represent a number of possible solutions. Furthermore, GAs' computational models can be easily parallelized[7-9] to exploit the capabilities of massively parallel computers and distributed systems.

Classes of search techniques

Figure 1 groups search techniques into three broad classes.[2] *Calculus-based techniques* use a set of necessary and sufficient conditions to be satisfied by the solutions of an optimization problem. These techniques subdivide into indirect and direct methods. Indirect methods look for local extrema by solving the usually nonlinear set of equations resulting from setting the gradient of the objective function equal to zero. The search for possible solutions (function peaks) starts by restricting itself to points with zero slope in all directions. Direct methods, such as those of Newton and Fibonacci, seek extrema by "hopping" around the search space and assessing the gradient of the new point, which guides the search. This is simply the notion of "hill-climbing," which finds the best local point by climbing the steepest permissible gradient. These techniques can be used only on a restricted set of "well-behaved" problems.

Enumerative techniques search every point related to an objective function's domain space (finite or discretized), one point at a time. They are very simple to implement but may require significant computation. The domain space of many applications is too large to search using these techniques. Dynamic programming is a good example of an enumerative technique.

Guided random search techniques are based on enumerative techniques but use additional information to guide the search. They are quite general in scope and can solve very complex problems. Two major subclasses are simulated annealing and evolutionary algorithms. Both are evolutionary processes, but simulated annealing uses a thermodynamic evolution process to search minimum energy states. Evolutionary algorithms, on the other hand, are based on natural-selection principles. This form of search evolves throughout generations, improving the features of potential solutions by means of biologically inspired operations. These techniques subdivide, in turn, into evolutionary strategies and genetic algorithms. Evolutionary strategies were proposed by Rechenberg[10] and Schwefel[11] in the early 1970s. They can adapt the process of "artificial evolution" to the requirements of the local response surface.[12] This means that unlike traditional GAs evolutionary strategies can adapt their major strategy parameters according to the local topology of the objective function.[13]

Following Holland's original genetic-algorithm proposal, many variations of the basic algorithm have been introduced. However, an important and distinctive feature of all GAs is the population-handling technique. The original GA adopted a *generational replacement policy*,[5] according to which the whole population is replaced in each generation. Conversely, the *steady-state policy*[5] used by many subsequent GAs selectively replaces the population. It is possible, for example, to keep one or more population members for several generations, while those individuals sustain a better fitness than the rest of the population.

After we introduce GA models and their programming, we present a survey of GA programming environments. We have grouped them into three major classes according to their objectives: Application-oriented systems hide the details of GAs and help users develop applications for specific domains, algorithm-oriented systems are based on specific GA models, and toolkits are flexible environments for programming a range of GAs and applications. We review the available environments and describe their common features and requirements. As case studies, we select some specific systems for more detailed examination. To conclude, we discuss likely future developments in GA programming environments.

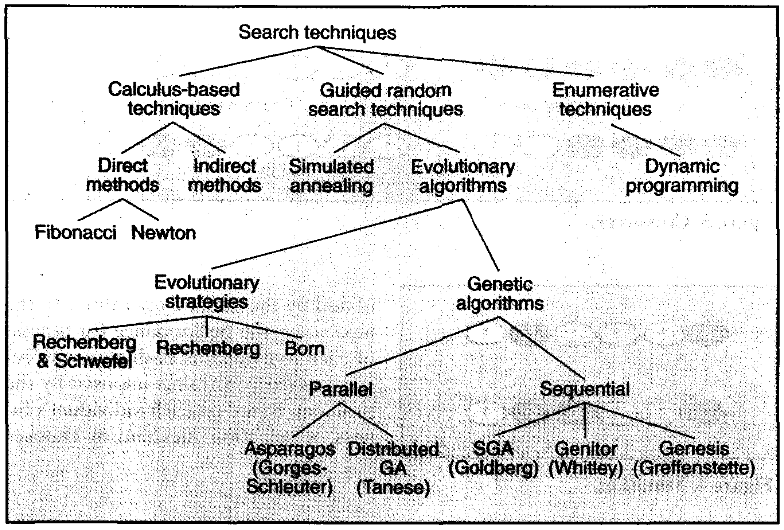


Figure 1. Classes of search techniques.

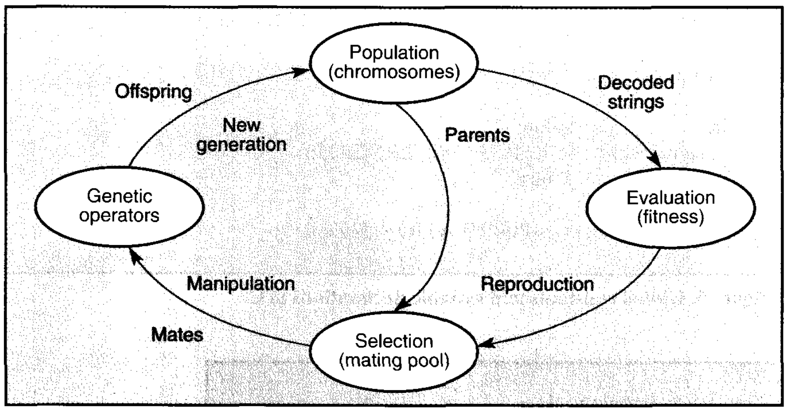


Figure 2. The GA cycle.

Genetic algorithms

A genetic algorithm emulates biological evolutionary theories to solve optimization problems. A GA comprises a set of individual elements (the population) and a set of biologically inspired operators defined over the population itself. According to evolutionary theories, only the most suited elements in a population are likely to survive and generate offspring, thus transmitting their biological heredity to new generations. In computing terms, a genetic algorithm maps a problem onto a set of (typically binary) strings, each string representing a potential solution. The GA then manipulates the most promising strings in its search for improved solutions. A GA operates through a simple cycle of stages:

(1) creation of a "population" of strings,
(2) evaluation of each string,
(3) selection of "best" strings, and
(4) genetic manipulation to create the new population of strings.

Figure 2 shows these four stages using the biologically inspired GA terminology. Each cycle produces a new generation of possible solutions for a given problem. At the first stage, an initial population of potential solutions is created as a starting point for the search. Each element of the population is encoded into a string (the chromosome) to be manip-

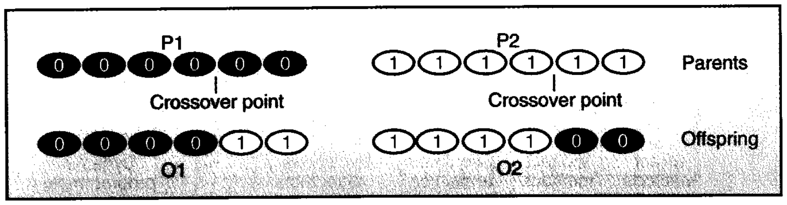


Figure 3. Crossover.

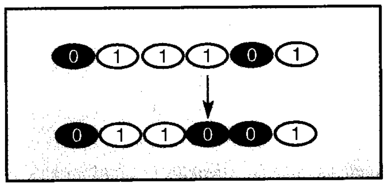


Figure 4. Mutation.

```
#define  POPULATION_SIZE    4       /* Size of the population */
#define  CHROM_LENGTH       5       /* String size            */
#define  PCROSS             0.6     /* Crossover probability */
#define  PMUT               0.001   /* Mutation probability  */

struct population
{
  int              value;
  unsigned char    string[CHROM_LENGTH];
  int              fitness;
};
struct population pool[POPULATION_SIZE];
```

Figure 5. Global constants and variable declarations in C.

```
initialize_population()
{
  randomize();              /* random generator setup */
  for (i=0; i < POPULATION_SIZE; i++)
    encode(i, random(pow(2.0,CHROM_LENGTH));
}
```

Figure 6. Initialization routine.

```
select(sum_fitness)
{
  parsum = 0;
  md = rand() % sum_fitness;                    /* spin the roulette */

  for (i=0; i < POPULATION_SIZE, parsum <= md; i++)
    parsum += pool[i].fitness;                  /* look for the slot */

  return (–i);                                  /* returns a selected string */
}
```

Figure 7. Selection function.

ulated by the genetic operators. In the next stage, the performance (or fitness) of each individual is evaluated with respect to the constraints imposed by the problem. Based on each individual's fitness, a selection mechanism chooses "mates" for the genetic manipulation process. The selection policy is ultimately responsible for assuring survival of the best fitted individuals. The combined evaluation and selection process is called reproduction.

The manipulation process uses genetic operators to produce a new population of individuals (offspring) by manipulating the "genetic information," referred to as genes, possessed by members (parents) of the current population. It comprises two operations: crossover and mutation. *Crossover* recombines a population's genetic material. The selection process associated with recombination assures that special genetic structures, called building blocks, are retained for future generations. The building blocks then represent the most fitted genetic structures in a population.

The recombination process alone cannot avoid the loss of promising building blocks in the presence of other genetic structures, which could lead to local minima. Also, it cannot explore search space sections not represented in the population's genetic structures. Here *mutation* comes into action. The mutation operator introduces new genetic structures in the population by randomly modifying some of its building blocks, helping the search algorithm escape from local minima's traps. Since the modification is not related to any previous genetic structure of the population, it creates different structures representing other sections of the search space.

The crossover operator takes two chromosomes and swaps part of their genetic information to produce new chromosomes. This operation is analogous to sexual reproduction in nature. As Figure 3 shows, after the crossover point has been randomly chosen, portions of the parent strings P1 and P2 are swapped to produce the new offspring strings O1 and O2. In Figure 3 the crossover operator is applied to the fifth and sixth elements of the string. Mutation is implemented by occasionally altering a random bit in a string. Figure 4 shows the mutation operator applied to the fourth element of the string.

A number of different genetic operators have been introduced since Holland proposed this basic model. They are, in general, versions of the recombination and genetic alteration processes adapted to the requirements of particular problems. Examples of other genetic operators are inversion, dominance, and genetic edge recombination.

The offspring produced by the genetic manipulation process are the next population to be evaluated. Genetic algorithms can replace either a whole population (generational approach) or its less fitted members only (steady-state approach). The creation-evaluation-selection-manipulation cycle repeats until a satisfactory solution to the problem is found or some other termination criterion is met.

This description of the computational model reviews the steps needed to design a genetic algorithm. However, real implementations take into account a number of problem-dependent parameters such as the population size, crossover and mutation rates, and convergence criteria. GAs are very sensitive to these parameters (a discussion of the methods for setting them up is beyond the scope of this article).

Sequential GAs. To illustrate the implementation of a sequential genetic algorithm we use Goldberg's simple function optimization example[2] and examine its programming in C. The first step in optimizing the function $f(x) = x^2$ over the interval (parameter set) [0-31] is to encode the parameter set x, for example, as a five-digit binary string {00000-11111}. Next we generate the initial population of four potential solutions, shown in Table 1, using a random number generator.

To program this GA function optimization, we declare the population pool as an array with four elements, as shown in Figure 5, and then initialize the structure using a random generator, as shown in Figure 6. Our next step is reproduction. Reproduction evaluates and selects pairs of strings for mating according to their relative strengths (see Table 1 and the associated C code in Figure 7). One copy of string 01101, two copies of 11000, and one copy of 10011 are selected by using a roulette wheel method.[2]

Next we apply the crossover operator, as illustrated in Table 2. Crossover operates in two steps (see Figure 8). First it determines whether crossover is to occur on a pair of strings by using a flip function: tossing a biased coin (with probability *pcross*). If the result is heads (true), the strings are swapped; the *crossover_point* is determined by a random number generator. If tails (false), the strings are simply copied. In the example, crossover occurs at the fifth position for the first pair and the third position for the other.

After crossover, the mutation operator is applied to the new population, which may have a random bit in a given string modified. The mutation function in Figure 9 on the next page uses the biased coin toss (flip) with probability *pmut* to determine whether to change a bit.

Table 3 shows the new population, to which the algorithm now applies a termination test. Termination criteria may include the simulation time being up, a specified number of generations exceeded, or a convergence criterion satisfied. In the example, we might set the number of generations to 50 and the con-

Table 1. Initial strings and fitness values.

Initial Population	x	$f(x)$ (fitness)	Strength (percent of total)
01101	13	169	14.4
11000	24	576	49.2
01000	8	64	5.5
10011	19	361	30.9
Sum_Fitness =		1,170	(100.0)

Table 2. Mating pool strings and crossover.

Mating Pool	Mates	Swapping	New Population
01101	1	0110[1]	01100
11000	2	1100[0]	11001
11000	2	11[000]	11011
10011	4	10[011]	10000

```
crossover (parent1, parent2, child1, child2)
{
  if (flip(PCROSS))
  {
    crossover_point = random(CHROM_LENGTH);

    for (i=0; i <= CHROM_LENGTH; i++)
    {
      if (i <= site)
      {
        new_pool[child1].string[i] = pool[parent1].string[i];
        new_pool[child2].string[i] = pool[parent2].string[i];
      }
      else
      {
        new_pool[child1].string[i] = pool[parent2].string[i];
        new_pool[child2].string[i] = pool[parent1].string[i];
      }
    }
  }
}
```

Figure 8. The crossover routine.

vergence as an average fitness improvement of less than 5 percent between generations. For the initial population, the average is 293, that is, $(169 + 576 + 64 + 361) \div 4$, while for the new population it has improved to 439, that is, 66 percent, (see the sidebar on Sequential GA C listing on page 34).

Parallel GAs. The GA paradigm offers intrinsic parallelism in searches for the best solution in a large search space, as demonstrated by Holland's schema theorem.[1] Besides the intrinsic parallelism, GA computational models can also exploit other levels of parallelism because of the natural independence of the genetic manipulation operations.

A parallel GA is generally formed by parallel components, each responsible for manipulating subpopulations. As was shown in Figure 1, there are two classes of parallel GAs: centralized and distributed. The first has a centralized selection mechanism: A single selection operator works synchronously on the global population (of subpopulations) at the selection stage. In distributed parallel GAs, each parallel component has its own copy of the selection operator, which works asynchronously. In addition, each component communicates its best strings to a subset of the other components. This process requires a migration operator and a migration frequency defining the communication interval.

The Asparagos algorithm[7] has a distributed mechanism. Figure 10 shows a skeleton C-like program, based on this algorithm, for the simple function optimization discussed for sequential algorithms. In this parallel program the statements for initialization, selection, crossover, and mutation remain almost the same as in the sequential program. For the main loop, parallel (PAR) subpopulations are set up for each component, as well as values for the new parameters. Each component then executes sequentially, apart from the parallel migration operator.

Taxonomy

To review programming environments for genetic algorithms, we use a simple taxonomy of three major classes: application-oriented systems, algorithm-oriented systems, and toolkits.

Application-oriented systems are essentially "black boxes" that hide the GA implementation details. Targeted at business professionals, some of these systems support a range of applications; others focus on a specific domain, such as finance.

Algorithm-oriented systems support specific genetic algorithms. They subdivide into

- algorithm-specific systems, which contain a single genetic algorithm, and
- algorithm libraries, which group together a variety of genetic algorithms and operators.

These systems are often supplied in source code and can be easily incorporated into user applications.

Toolkits provide many programming utilities, algorithms, and genetic operators for a wide range of application domains. These programming systems subdivide into

- educational systems that help novice users obtain a hands-on introduction to GA concepts, and
- general-purpose systems that provide a comprehensive set of tools for programming any GA and application.

Table 4 lists the GA programming environments examined in the next sections, according to their categories. For each category we present a generic system overview, then briefly review example systems, and finally examine one system in more detail, as a case study. The parallel environments GAUCSD, Pegasus, and GAME are also covered, but no commercial parallel environments are currently available. See the sidebar "Developers address list" on page 37 for a comprehensive list of programming environments and their developers.

```
mutation ()
{
  for (i=0; i < POPULATION_SIZE; i++)
  {
    for (j=0; j < CHROM_LENGTH; j++)
      if (flip(PMUT))
        pool[i].string[j] = ~ new_pool[i].
          string[j] & 0x01;
      else
        pool[i].string[j] = new_pool[i].string[j];
  }
}
```

Figure 9. The mutation operator C implementation.

Table 3. Second generation and its fitness values.

Initial Population	x	$f(x)$ (fitness)	Strength (percent of total)
01100	12	144	8.2
11001	25	625	35.6
11011	27	729	41.5
10000	16	256	14.7
Sum_Fitness =		1,754	(100.0)

Application-oriented systems

Many potential users of a novel computing technique are interested in applications rather than the details of the technique. Application-oriented systems are designed for business professionals who want to use genetic algorithms for specific purposes without having to acquire detailed knowledge about them. For example, a manager in a trading company may need to optimize its delivery scheduling. By using an application-oriented programming environment, the manager can configure an application for schedule optimization based on the traveling-salesman problem without having to know the encoding technique or the genetic operators.

Overview. A typical application-oriented environment is analogous to a spreadsheet or word-processing utility. Its menu-driven interface (tailored to business users) gives access to parameterized modules (targeted at specific domains). The user interface provides menus to configure an application, monitor its execution, and, in certain cases, program an application. Help facilities are also provided.

Survey. Application-oriented systems have many innovative strategies. Systems such as PC/Beagle and XpertRule GenAsys are expert systems that use GAs to generate new rules to expand their knowledge base of the application domain. Evolver is a companion utility for spreadsheets. Omega is targeted at financial applications.

Evolver. This add-on utility works within the Excel, Wingz, and Resolve spreadsheets on Macintosh and PC computers. Axcelis, its marketer, describes it as "an optimization program that extends mechanisms of natural evolution to the world of business and science applications." A user starts with a model of a system in the spreadsheet and calls the Evolver program from a menu. After the user fills a dialog box with the information required (the cell to minimize or maximize), the program starts working, evaluating thousands of scenarios automatically until it has found an optimal answer. The program runs in the background, freeing the user to work in the foreground.

When Evolver finds the best result, it notifies the user and places the values into the spreadsheet for analysis. This is an excellent design strategy, given the importance of spreadsheets in business. In an attempt to improve the system and ex-

```
#define MAX_GEN                    50
#define POPULATION_SIZE            32
#define SUB_POP_SIZE                8
#define NUM_OF_GAS                  POPULATION_SIZE/
                                    SUB_POP_SIZE

#define NUM_OF_NEIGHBORS            2
#define MIGRATION_FREQ              5
#define NUM_OF_EMIGRANTS            2

main ()
{
PAR    for (i=o; i<SUB_POP_SIZE; i++)      /* Parallel execution   */
                                           /* over subpopulation */
SEQ    {       initialize(); }

       do;
       {
         for (j=0; j<MIGRATION_FREQ; j++)
SEQ      {            selection(..)        /* evaluate and select  */
                      crossover(..);       /* of the standard GA */
                      mutation(..);
         }
         for (j=0; j<NUM_OF_EMIGRANTS; j++)
SEQ      {            emigrant(j) = select_emigrant(..);}
         for (j=0; j<NUM_OF_NEIGHBORS; j++)
PAR      {            send_emigrants(..);
                      receive_emigrants(..);
         }
       }
       while (generations <= MAX_GEN);
}
```

Figure 10. Parallel GA with migration.

Table 4. Programming environments and their categories.

	Algorithm-Oriented Systems		Toolkits	
Application-Oriented Systems	Algorithm-specific systems	Algorithm libraries	Educational systems	General-purpose systems
Evolver Omega PC/Beagle	Escapade GAGA GAUCSD	EM	GA Workbench	Engeneer GAME MicroGA Pegasus Splicer
XpertRule GenAsys	Genesis Genitor	OOGA		

pand its market. Axcelis introduced Evolver 2.0, which has many toolkit-like features. The new version can integrate with other applications in addition to spreadsheets. It also offers more flexibility in accessing the Evolver engine: This can be done from any Microsoft Windows application that can call a Dynamic Link Library.

Omega. The Omega Predictive Modelling System, marketed by KiQ, is a powerful approach to developing predictive models. It exploits advanced GA techniques to create a tool that is "flexible, powerful, informative and straightforward to use," according to its developers. Geared to the financial domain, Omega can be applied to direct marketing, insurance, investigations (case scoring), and credit management. The envi-

Sequential GA C Listing

```
/*
**********************************************************
 *       Simple Genetic Algorithm
**********************************************************
 */
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <math.h>

#define RAND_MAX           0x7FFFFFFF
#define random(num)     (rand()%(num))
#define randomize()     srand((unsigned)time(NULL))

#define POPULATION_SIZE  10
#define CHROM_LENGTH     4
#define PCROSS         0.6
#define PMUT        0.050
#define MAX_GEN        50

struct population
{
   int            value;
   unsigned char   string[CHROM_LENGTH];
   unsigned int    fitness;
};

struct population pool[POPULATION_SIZE];
struct population new_pool[POPULATION_SIZE];

int selected[POPULATION_SIZE];
int generations;

main()
{
   int i;
   double sum_fitness, avg_fitness, old_avg_fitness;
   generations = 1;
   avg_fitness = 1;

   initialize_population();

   do
   {
      old_avg_fitness = avg_fitness;
      sum_fitness = 0;

      /* fitness evaluation */
      for (i=0; i<POPULATION_SIZE; i++)
      {
         pool[i].value = decode(i);
         pool[i].fitness = evaluate(pool[i].value);
         sum_fitness += pool[i].fitness;
      }

      avg_fitness = sum_fitness / POPULATION_SIZE;

      for (i=0; i<POPULATION_SIZE; i++)
         selected[i] = select(sum_fitness);

      for (i=0; i<POPULATION_SIZE; i=i+2)
crossover(selected[i],selected[i+1],i,i+1);

      mutation();

      statistics();
    printf ("\nImprovment: %f\n", avg_fitness/old_avg_fitness);
   }
   while ((++generations < MAX_GEN) &&
        ((avg_fitness/old_avg_fitness) > 1.005) ||
       ((avg_fitness/old_avg_fitness) < 1.0));
}

/*
**********************************************************
 *       initialize_population
 *    Creates and initializes a population
**********************************************************
 */
initialize_population()
{
   int i;

   randomize();
   for (i=0; i < POPULATION_SIZE; i++)
      encode(i, random(2^CHROM_LENGTH));
}
/*
**********************************************************
 *       select
 *    Selects strings for reproduction
**********************************************************
 */
select(sum_fitness)
double sum_fitness;
{
   int i;
   double r, parsum;

   parsum = 0;

   r = (double)(rand() % (int)sum_fitness);  /* spin the roulette */

   for (i=0; i < POPULATION_SIZE, parsum <= r; i++)
      parsum += pool[i].fitness;

   return (—i);    /* returns a selected string */
}
/*
**********************************************************
 *       crossover
 *    Swaps 2 sub-strings
**********************************************************
 */
crossover (parent1, parent2, child1, child2)
int parent1;
int parent2;
int child1;
int child2;
{
   int i, site;

   if (flip(PCROSS))
      site = random(CHROM_LENGTH);
```

ronment offers facilities for automatic handling of data; business, statistical, or custom measures of performance; simple and complex profit modeling; validation sample tests; advanced confidence tests; real-time graphics; and optional control over the internal genetic algorithm.

PC/Beagle. Produced by Pathway Research, this rule-finder program applies machine learning techniques to create a set of decision rules for classifying examples previously extracted from a database. It has a module that generates rules by natural selection. Further details are given in the case study section.

XpertRule GenAsys. XpertRule GenAsys is an expert system shell with embedded genetic algorithms. Marketed by Attar Software, this GA expert system solves scheduling and design problems.

```
   else
      site = CHROM_LENGTH-1;

   for (i=0; i < CHROM_LENGTH; i++)
   {
      if ((i <= site) || (site==0))
      {
         new_pool[child1].string[i] = pool[parent1].string[i];
         new_pool[child2].string[i] = pool[parent2].string[i];
      }else
      {
         new_pool[child1].string[i] = pool[parent2].string[i];
         new_pool[child2].string[i] = pool[parent1].string[i];
      }
   }
}
/*
 ***************************************************
 *        mutation
 *     Changes the values of string position
 ***************************************************
 */
mutation ()
{
   int i, j;

   for (i=0; i < POPULATION_SIZE; i++)
   {
      for (j=0; j < CHROM_LENGTH; j++)
         if (flip(PMUT))
            pool[i].string[j] = ~new_pool[i].string[j] & 0x01;
         else
            pool[i].string[j] = new_pool[i].string[j] & 0x01;
   }
}
/*
 ***************************************************
 *        encode
 *     Code a integer into binary string
 ***************************************************
 */
encode(index, value)
int index;
int value;
{
   int i;

   for (i=0; i < CHROM_LENGTH; i++)
      pool[index].string[CHROM_LENGTH-1-i] = (value >> i) &
0x01;
}
/*
 ***************************************************
 *        decode
 *     Decode a binary string into an integer
 ***************************************************
 */
decode(index)
int index;
{
   int i, value;

   value = 0;
   for (i=0; i < CHROM_LENGTH; i++)
      value += (int)pow(2.0,(double)i) *
pool[index].string[CHROM_LENGTH-1-i];

   return(value);
}
/*
 ***************************************************
 *        evaluate
 *     Objective function f(x)=x^2
 ***************************************************
 */
evaluate(value)
int value;
{
   return(pow((double)value,2.0));
}
/*
 ***************************************************
 *        flip
 *     Toss a biased coin
 ***************************************************
 */
flip(prob)
double prob;
{
   double i;

   i=((double)rand())/RAND_MAX;

   if ((prob == 1.0) || (i < prob))
      return (1);
   else
      return (0);
}
/*
 ***************************************************
 *        statistics
 *     Print intermediary results
 ***************************************************
 */
statistics()
{
   int i, j;

   printf("\n;Generation: %d\n;Selected Strings\n;", generations);
   for (i=0; i< POPULATION_SIZE; i++)
      printf(" %d", selected[i]);

   printf("\n");

   printf("\n;X\tf(x)\t New_String\tX");

   for (i=0; i< POPULATION_SIZE; i++)
   {
      printf("\n %d\t%u\t;", pool[i].value, pool[i].fitness);

      for (j=0; j<CHROM_LENGTH; j++)
         printf(" %d",pool[i].string[j]);

      printf("\t%d", decode(i));
   }
}
```

The system combines the power of genetic algorithms in evolving solutions with the power of rule-base programming in analyzing the effectiveness of solutions. Rule-base programming can also be used to generate the initial solutions for the genetic algorithm and for postoptimization planning. Problems this system can solve include optimization of design parameters in the electronics and avionics industries, route optimization in the distribution sector, and production scheduling in manufacturing.

Case study: PC/Beagle. PC/Beagle is a rule-finder program that examines a database of examples and uses machine learning techniques to create decision rules for classifying those examples, turning data into knowledge. The software analyzes an expression via a historical database and develops a series of rules to explain when the target expression is false or true. The system contains six main components generally run in sequence:

- SEED (selectively extracts example data) puts external data into a suitable format and may append leading or lagging data fields as well.
- ROOT (rule-oriented optimization tester) tests an initial batch of user-suggested rules.
- HERB (heuristic evolutionary rule breeder) generates decision rules by natural selection, using GA philosophy and ranking mechanisms.
- STEM (signature table evaluation module) makes a signature table from the rules produced by HERB.
- LEAF (logical evaluator and forecaster) uses STEM output to do forecasting or classification.
- PLUM (procedural language utility maker) can convert a Beagle rule file into a language such as Pascal or Fortran so other software can use the knowledge gained.

PC/Beagle accepts data in ASCII format, with items delimited by commas, spaces, or tabs. Rules are produced as logical expressions. The system is highly versatile, covering a wide range of applications. Insurance, weather forecasting, finance, and forensic science are some examples. PC/Beagle requires an IBM PC-compatible computer with at least 256 Kbytes of RAM and an MS-DOS or PC-DOS operating system, version 2.1 or later.

Algorithm-oriented systems

Our taxonomy divides algorithm-oriented systems into algorithm-specific systems that contain a single algorithm and algorithm libraries, which group together a variety of genetic algorithms and operators.

Algorithm-specific environments embody a single powerful genetic algorithm. These systems have typically two groups of users: system developers requiring a general-purpose GA for their applications and researchers interested in the development and testing of a specific algorithm and genetic operators.

Algorithm-specific environments embody a single powerful genetic algorithm.

Overview of algorithm-oriented systems. In general, these systems come in source code so expert users can make alterations for specific requirements. They have a modular structure for a high degree of modifiability. In addition, user interfaces are frequently rudimentary, often command-line driven. Typically the codes have been developed at universities and research centers, and are available free over worldwide computer research networks.

System survey. The most well known programming system in this category is the pioneering Genesis,[4] which has been used to implement and test a variety of new genetic operators. In Europe probably the earliest algorithm-specific system was GAGA. For scheduling problems, Genitor[14] is another influential and successful system. GAUCSD permits parallel execution: It distributes several copies of a Genesis-based algorithm to Unix machines in a network. Escapade[13] uses a somewhat different approach — an evolutionary strategy.

Escapade. Escapade (Evolutionary Strategies Capable of Adaptive Evolution) provides a sophisticated environment for a particular class of evolutionary algorithms, called evolutionary strategies. Escapade is based on Korr, Schwefel's implementation of a $(\mu, +\lambda)$-evolutionary strategy, where the μ best individuals of the λ offspring, added to their parents, survive and become the parents of the new generation. The system provides an elaborate set of monitoring tools to gather data from an optimization run of Korr. According to Escapade's author, it should be possible to incorporate a different implementation of an evolutionary strategy or even a GA into the system using its runtime support. The program is separated into several independent components that support the various tasks during a simulation run. The major modules are parameter setup, runtime control, Korr, generic data monitors, customized data monitors, and monitoring support.

During an optimization run, the monitoring modules are invoked by the main algorithm (Korr or some other evolutionary strategy or GA implementation) to log internal quantities. The system is not equipped with any kind of graphical interface. Users must pass all parameters for a simulation as command-line options. For output, each data monitor writes its data into separate log files.

GAGA. The Genetic Algorithms for General Application were originally programmed in Pascal by Hillary Adams at the University of York. The program was later modified by Ian Poole and translated into C by Jon Crowcroft at University College London. GAGA is a task-independent genetic algorithm. The user must supply the target function to be optimized (minimized or maximized) and some technical GA parameters, and wait for the output. The program is suitable for the minimization of many difficult cost functions.

GAUCSD. This software was developed by Nicol Schraudolph at the University of California, San Diego (hence UCSD).[15] The system is based on Genesis 4.5 and runs on Unix, MS-DOS, Cray operating system, and VMS platforms, but it presumes a Unix environment. GAUCSD comes with an *awk* script called "wrapper," which provides a higher level of abstraction for defining the evaluation function. By supplying the code for decoding and printing this function's parameters automatically, it allows the direct use of most C functions as evaluation functions, with few restrictions. The software also includes a dynamic parameter encoding technique developed by Schraudolph, which radically reduces the gene length while keeping the desired level of precision for the results. Users can run the system in the background at low priority using the *go* command.

The *go* command can also be used to execute GAUCSD on remote hosts. The results are then copied back to the user's local directory, and a report is produced if appropriate. If the host is not binary compatible, GAUCSD compiles the whole system on the remote host. Experiments can be queued in files, distributed to several hosts, and executed in parallel. The experiments are distributed according to a specified loading factor (how many programs will be sent to each host), along with the remote execution arguments to the *go* command. The *ex* command notifies the user via write or mail when all experiments are completed. GAUCSD is clearly a very powerful system.

Genesis. The Genetic Search Implementation System, or Genesis, was written by John Grefenstette[4] to promote the study of genetic algorithms for function optimization. It has been under development since 1981 and widely distributed to the research community since 1985. The package is a set of routines written in C. To build their own genetic algorithms, users provide only a routine with the fitness function and link it with the other routines. Users can also modify modules or add new ones (for example, genetic operators and data monitors) and create a different version of Genesis. In fact, Genesis has been used as a base for test and evaluation of a variety of genetic al-

Developers address list

C Darwin II
Attar Software
Newlands Road
Leigh, Lancashire, UK
Telephone: +44 94 2608844
Fax: +44 94 2601991
E-mail: 100166.1547
@CompuServe.com

EM — Evolution Machine
H.M. Voigt and J. Born
Technical University of Berlin
Bionics and Evolution
Techniques Laboratory
Bio and Neuroinformatics
Research Group
Ackerstasse 71-76 (ACK1)
D-13355 Berlin, Germany
Telephone: +49 303 147 2677
E-mail: voigt@fb10.tu-berlin.de
born@fb10.tu-berlin.de

Escapade
Frank Hoffmeister
University of Dortmund
System Analysis Research Group, LSXI
D-44221 Dortmund, Germany
Telephone: +49 231 755 4678
Fax: +49 231 755 2450
E-mail: hoffmeister@ls11.
informatik.uni-dortmund.de

Engeneer
George Robbins
Systems Intelligence Division
Logica Cambridge Ltd.
Betjeman House
104 Hills Road
Cambridge CB2 1LQ, UK
Telephone: +44 71 6379111
Fax: +44 223 322315

Evolver
Axcelis Inc.
4668 Eastern Avenue North
Seattle, WA 98103
Telephone: (206) 632-0885
Fax: (206) 632-3681

GA Workbench
Mark Hughes
Cambridge Consultants Ltd.
Science Park, Milton Rd.
Cambridge CB4 4DW, UK
Telephone: +44 223 420024
Fax: +44 223 423373
E-mail: mrh@camcon.co.uk

GAGA
Jon Crowcroft
University College London
Gower St.
London WC1E 6BT, UK
Telephone: +44 71 387 7050
Fax: +44 71 387 1398
E-mail: jon@cs.ucl.ac.uk

GAME
José L. Ribeiro Filho
Computer Science Department
University College London
Gower St.
London WC1E 6BT, UK
Telephone: +44 71 387 7050
Fax: +44 71 387 1398
E-mail: j.ribeirofilho@cs.ucl.ac.uk

GAUCSD
N.N. Schraudolph
Computer Science and Engineering
Department
University of California, San Diego
La Jolla, CA 92093-0114
Fax: (619) 534-7029
E-mail: nici@cs.ucsd.edu

Genesis
J.J. Grefenstette
The Software Partnership
PO Box 991
Melrose, MA 02176
Telephone: (617) 662-8991
E-mail: gref@aic.nrl.navy.mil

Genitor
Darrel Whitley
Computer Science Department
Colorado State University
Fort Collins, CO 80523
E-mail: whitley@cs.colostate.edu

MicroGA
Steve Wilson
Emergent Behavior
635 Wellsbury Way, Palo Alto, CA 94306
Telephone: (415) 494-6763
E-mail: emergent@aol.com

Omega
David Barrow
KiQ Ltd.
Easton Hall, Great Easton
Essex CM6 2HD, UK
Telephone: +44 371 870254

OOGA
Lawrence Davis
The Software Partnership
PO Box 991
Melrose, MA 02176

PC/Beagle
Richard Forsyth
Pathway Research Ltd.
59 Cronbrook Rd.
Bristol BS6 7BS, UK
Telephone: +44 272 428692

Pegasus
Dirk Schlierkamp-Voosen
German National Research Center
for Computer Science — GMD
Research Group for Adaptive Systems
PO Box 1316
D-53731 Sankt Augustin, Germany
Telephone: +49 224 114 2466
E-mail: dirk.schlierkamp-voosen@
gmd.de

Splicer
Cosmic
382 E. Broad St.
Athens, GA 30602
Telephone: (404) 542-3265
Fax: (706) 542-4807
E-mail: bayer@galileo.jsc.nasa.gov

XpertRule GenAsys
Attar Software
Newlands Road
Leigh, Lancashire, UK
Telephone: +44 94 2608844
Fax: +44 94 2601991
E-mail: 100166.1547@CompuServe.com

Xype
Ed Swartz
Virtual Image Inc.
75 Sandy Pond Road 11
Ayer, MA 01432
Telephone: (508) 772-0800

gorithms and operators. It was primarily developed to work in a scientific environment and is a suitable tool for research. Genesis is highly modifiable and provides a variety of statistical information on output.

Genitor. The modular GA package Genitor (Genetic Implementor) has examples for floating-point, integer, and binary representations. Its features include many sequencing operators, as well as subpopulation modeling. The software package is an implementation of the Genitor algorithm developed by Darrel Whitley.[14]

Genitor has two major differences from standard genetic algorithms. The first is its explicit use of ranking. Instead of using fitness-proportionate reproduction, Genitor allocates reproductive trials according to the rank of the individual in the population. The second difference is that Genitor abandons the generational approach (in which the whole population is replaced with each generation) and reproduces new genotypes on an individual basis. Using the steady-state approach, Genitor lets some parents and offspring coexist. A newly created offspring replaces the lowest ranking individual in the population rather than a parent. Because Genitor produces only one new genotype at a time, inserting a single new individual is relatively simple. Furthermore, the insertion automatically ranks the individual in relation to the existing pool — no further measure of the relative fitness is needed.

Case study: Genesis. Genesis[4] is the most well known software package for GA development and simulation. It runs on most machines with a C compiler. Version 5.0, now available from the Software Partnership, runs successfully on both Sun workstations and IBM PC-compatible computers, according to its author. The code is designed to be portable, but minor changes may be necessary for other systems.

Genesis provides the fundamental procedures for genetic selection, crossover, and mutation. The user is only required to provide the problem-dependent evaluation function.

Genesis has three levels of representation for the structures it evolves. The lowest level, packed representation, maximizes both space and time efficiency in manipulating structures. In general, this level of representation is transparent to the user. The next level, the string representation, represents structures as null-terminated arrays of characters, or "chars." This structure is for users who wish to provide an arbitrary interpretation of the genetic structures, for example, nonnumeric concepts. The third level, the floating-point representation, is appropriate for many numeric optimization problems. At this level the user views genetic structures as vectors or real numbers. For each parameter, or gene, the user specifies its range, number of values, and output format. The system then automatically lays out the string representation and translates between the user-level genes and lower representation levels.

Algorithm libraries provide a powerful collection of parameterized genetic algorithms and operators.

Genesis has five major modules:

- *Initialization.* The initialization procedure sets up the initial population. Users can "seed" the initial population with heuristically chosen structures, and the rest of the population is filled with random structures. Users can also initialize the population with real numbers.
- *Generation.* This module executes the selection, crossover, mutation, and evaluation procedures, and collects some data.
- *Selection.* The selection module chooses structures for the next generation from the structures in the current generation. The default selection procedure is stochastic, based on the roulette wheel algorithm, to guarantee that the number of offspring of any structure is bounded by the floor and ceiling of the (real-valued) expected number of offspring. Genesis can also perform selection using a ranking algorithm. Ranking helps forestall premature convergence by preventing "super" individuals from taking over the population within a few generations.
- *Mutation.* After Genesis selects the new population, it applies mutation to each structure. Each position is given a chance (according to the mutation rate) of undergoing mutation. If mutation is to occur, Genesis randomly chooses 0 or 1 for that position. If the mutated structure differs from the original one, it is marked for evaluation.
- *Crossover.* The crossover module exchanges alleles between adjacent pairs of the first n structures in the new population. The result of the crossover rate applied to the population size gives the number n of structures to operate on. Crossover can be implemented in a variety of ways. If, after crossover, the offspring are different from the parents, then the offspring replace the parents and are marked for evaluation.

These basic modules are added to the evaluation function supplied by the user to create the customized version of the system. The evaluation procedure takes one structure as input and returns a double-precision value.

To execute Genesis, three programs are necessary: *set-up*, *report*, and *ga*. The setup program prompts for a number of input parameters. All the information is stored in files for future use. Users can set the type of representation, number of genes, number of experiments, trials per experiment, population size, length of the structures in bits, crossover and mutation rates, generation gap, scaling window, and many other parameters. Each parameter has a default value.

The report program runs the genetic algorithm and produces a description of its performance. It summarizes the mean, variance, and range of several measurements, including on-line performance, off-line performance, average performance of the current population, and current best value.

Overview of algorithm libraries. Algorithm libraries provide a powerful collection of parameterized genetic algorithms and operators, generally coded in a common language, so users can easily incorporate them in applications. These libraries are modular, letting users select a variety of algorithms, operators, and parameters to solve particular problems. They allow parameterization so users can try different models and compare the results for the same problem. New algorithms coded in high-level languages like C or Lisp can be easily incorporated into the libraries. The user interface facilitates model configuration and manipulation, and presents the results in different shapes (tables, graphics, and so on).

Library survey. The two leading algorithm libraries are EM and OOGA. Both provide a comprehensive selection of genetic algorithms, and EM also supports evolutionary strategy simulation. OOGA can be easily tailored for specific problems. It runs in Common Lisp and CLOS (Common Lisp Object System), an object-oriented extension of Common Lisp.

EM. Developed by Hans-Michael Voigt, Joachim Born, and Jens Treptow[16] at the Institute for Informatics and Computing Techniques in Germany, EM (Evolution Machine) simulates natural evolution principles to obtain efficient optimization procedures for computer models. The authors chose different evolutionary methods to provide algorithms with different numerical characteristics. The programming environment supports the following algorithms:

- Rechenberg's evolutionary strategy,[10]
- Rechenberg and Schwefel's evolutionary strategy,[10,11]
- Born's evolutionary strategy,
- Goldberg's simple genetic algorithm,[2] and
- Voigt and Born's genetic algorithm.[16]

To run a simulation, the user provides the fitness function coded in C. The system calls the compiler and linker, which produce an executable file containing the selected algorithm and the user-supplied fitness function.

EM has extensive menus and default parameter settings. The program processes data for repeated runs, and its graphical presentation of results includes on-line displays of evolution progress and one-, two-, and three-dimensional graphs. The system runs on an IBM PC-compatible computer with the MS-DOS operating system and uses the Turbo C (or Turbo C++) compiler to generate the executable files.

OOGA. The Object-Oriented Genetic Algorithm is a simplified version of the Lisp-based software developed in 1980 by Lawrence Davis. He created it mainly to support his book,[5] but it can also be used to develop and test customized or new genetic algorithms and genetic operators.

Case study: OOGA. This algorithm is designed so each technique used by a GA is an object that can be modified, displayed, or replaced in an object-oriented fashion. It provides a highly modular architecture in which users incrementally write and modify components in Common Lisp to define and use a variety of GA techniques. The files in the OOGA system contain descriptions of several techniques used by GA researchers, but they are not exhaustive. OOGA contains three major modules:

Toolkits contain educational systems for novice users and general-purpose systems with a comprehensive set of tools.

- The *evaluation module* has the evaluation (or fitness) function that measures the worth of any chromosome for the problem to be solved.
- The *population module* contains a population of chromosomes and the techniques for creating and manipulating that population. There are a number of techniques for population encoding (binary, real number, and so on), initialization (random binary, random real, and normal distribution) and deletion (delete all and delete last).
- The *reproduction module* has a set of genetic operators for selecting and creating new chromosomes. This module allows GA configurations with more than one genetic operator. The system creates a list with user-selected operators and executes their parameter settings, before executing them in sequence. OOGA provides a number of genetic operators for selection (for example, roulette wheel), crossover (one- and two-point crossover, mutate-and-crossover), and mutation. The user can set all parameters, such as the bit-mutation and crossover rates.

The last two modules are, in fact, libraries of different techniques enabling the user to configure a particular genetic algorithm. When the genetic algorithm is run, the evaluation, population, and reproduction modules work together to evolve a population of chromosomes toward the best solution. The system also supports some normalization (for example, linear normalization) and parameterization techniques for altering the genetic operators' relative performance over the course of the run.

Toolkits

Toolkits subdivide into educational systems for novice users and general-purpose systems that provide a comprehensive set of programming tools.

Educational systems overview. Educational programming systems help novices gain a hands-on introduction to GA concepts. They typically provide a rudimentary graphical interface and a simple configuration menu. Educational systems are typically implemented on PCs for portability and low cost. For ease of use, they have a fully menu-driven graphical interface. GA Workbench[17] is one of the best examples of this class of programming environment.

Case study: GA Workbench. This environment was developed by Mark Hughes of Cambridge Consultants to run on MS-DOS/PC-DOS microcomputers. With this mouse-driven interactive program, users draw evaluation functions on the screen. The system produces runtime plots of GA population distribution, and peak and average fitness. It also displays many useful population statistics. Users can change a range of parameters, including the settings of the genetic operators, population size, and breeder selection.

GA Workbench's graphical interface uses a VGA or EGA adapter and divides the screen into seven fields consisting of menus or graphs. The *command menu* is a menu bar that lets the user enter the target function and make general commands to start or stop a GA execution. After selecting "Enter Targ" from the command menu, the user inputs the target function by drawing it on the *target* *function graph* using the mouse cursor.

The *algorithm control chapter* can contain two pages (hence "chapter"), but only one page is visible at a time. Clicking with the mouse on screen arrows lets the user flip pages forward or backward. The initial page, the "simple genetic algorithm page," shows a number of input variables used to control the algorithm's operation. The variable values can be numeric or text strings, and the user can alter any of these values by clicking the left mouse button on the up or down arrows to the left of each value. The "general program control variables page" contains variables related to general program operation rather than a specific algorithm. Here the user can select the source of data for plotting on the output plot graph, set the scale for the x or y axis, seed the random number generated, or determine the frequency with which the population distribution histogram is updated.

The *output variables box* contains the current values of variables relating to the current algorithm. For the simple genetic algorithm, a counter of generations is presented as well as the optimum fitness value, current best fitness, average fitness, optimum x, current best x, and average x. The *population distribution histogram* shows the genetic algorithm's distribution of organisms by value of x. The histogram is updated according to the frequency set in the general program control variables page. The *output graph* plots several output variables against time.

From any graph, the user can read the coordinate values of the point indicated by the mouse cursor. When the user moves the cursor over the plot area of a graph, it changes to a cross hair and the *axis value box* displays the coordinate values.

By drawing the target function, varying several numeric control parameters, and selecting different types of algorithms and genetic operators, the novice user can practice and see how quickly the algorithm can find the peak value, or indeed if it succeeds at all.

General-purpose programming systems overviw. General-purpose systems are the ultimate in flexible GA programming. Not only do they let users develop their own GA applications and algorithms; they also let users customize the system.

These programming systems provide a comprehensive toolkit, including

- a sophisticated graphical interface,
- a parameterized algorithm library,
- a high-level language for programming GAs, and
- an open architecture.

Users access system components via a menu-driven graphical interface. The algorithm library is normally "open," letting users modify or enhance any module. A high-level language — often object-oriented — may be provided for programming GA applications, algorithms, and operators through specialized data structures and functions. And because parallel GAs are becoming important, systems provide translators to parallel machines and distributed systems, such as networks of workstations.

General-purpose systems let programmers develop applications and algorithms and customize the system.

General-purpose survey. The number of general-purpose systems is increasing, stimulated by growing interest in GA applications in many domains. Systems in this category include Splicer, which presents interchangeable libraries for developing applications; MicroGA, which is an easy-to-use object-oriented environment for PCs and Macintoshes; and the parallel environments Engeneer, GAME, and Pegasus.

Engeneer. Logica Cambridge developed Engeneer[18] as an in-house environment to assist in GA application development in a wide range of domains. The C software runs on Unix systems as part of a consultancy and systems package. It supports both interactive (X Windows) and batch (command-line) operation. Also, it supports a certain degree of parallelism for the execution of application-dependent evaluation functions.

Engeneer provides flexible mechanisms that let the developer rapidly bring the power of GAs to bear on new problem domains. Starting with the Genetic Description Language, the developer can describe, at a high level, the structure of the "genetic material" used. The language supports discrete genes with user-defined cardinality and includes features such as multiple models of chromosomes, multiple species models, and nonevolvable parsing symbols, which can be used for decoding complex genetic material.

A descriptive high-level language, the Evolutionary Model Language, lets the user describe the GA type in terms of configurable options including population size, population structure and source, selection method, crossover type and probability, mutation type and probability, inversion, dispersal method, and number of offspring per generation.

An interactive interface (with on-line help) supports both high-level languages. Descriptions and models can be defined "on the fly" or loaded from audit files, which are automatically created during a GA run. Users can monitor GA progress with graphical tools and by defining intervals for automatic storage of results. Automatic storage lets the user restart Engeneer from any point in a run, by loading both the population at that time and the evolutionary model.

To connect Engeneer to different problem domains, a user specifies the name of the program to evaluate the problem-specific fitness function and constructs a simple parsing routine to interpret the genetic material. Engeneer provides a library of standard interpretation routines for commonly used representation schemes such as gray coding and permutations. The fitness evaluation can then be run as the GA's slave process or via standard handshaking routines. Better still, it can be run on the machine hosting Engeneer or on any sequential or parallel hardware capable of connecting to a Unix machine.

GAME. The Genetic Algorithm Manipulation Environment is being developed as part of the European Community (ESPRIT III) GA project called Papagena. It is an object-oriented environment for programming parallel GA applications and algorithms, and mapping them onto parallel machines. The environment has five principal modules.

The *virtual machine* (VM) is the module responsible for maintaining data structures that represent genetic information and providing facilities for their manipulation and evaluation. It isolates genetic operators and algorithms from dealing directly with data structures through a set of low-level commands implemented as a collection of functions called the VM Application Program Interface (VM-API). The VM also supports fine-grained parallelism and can execute several commands simultaneously. It comprises three modules: the production manager, the fitness evaluation module, and the parallel support module. The first executes genetic manipulation commands over the data structures residing in the VM population pools. The VM-API includes commands for swapping, inverting, duplicating, and modifying genetic structures. The fitness evaluation module performs the actual evaluation of genetic structures and such related calculations as total, average, highest, and lowest fitness values. The problem-dependent objective function is only "connected" to the fitness evaluation module at link time. Finally, the parallel support module schedules commands received by the VM among several copies of the population manager and fitness evaluation modules.

The *parallel execution module* (PEM) implements a hardware/operating system-independent interface that supports multiple, parallel computational models. It provides straightforward API-containing functions for process initiation, termination, synchronization, and communication. It is responsible for integrating application components (algorithms, operators, user interface, and virtual machine) defined as GAME components. The PEM is implemented in two layers. The upper layer defines the standard interface functions used by all GAME components of an application. The lower layer implements the functions that map the upper layer requests into the particular environment. PEM's design permits porting GAME applications to diverse sequential and parallel machines by simply linking with the PEM library implemented for the required machine/operating system.

A *graphical user interface* module containing simple graphic widgets for MS-Windows and X Windows environments is also provided. It enables applications to input and output data in a variety of formats. GAME's GUI contains standard dialog boxes, buttons, and charting windows that can be associated by the user with events reported by the monitoring control module.

The *monitoring control module* (MCM) collects and displays (through the GUI) events that occur during a simulation session. Each GAME component notifies the MCM about messages received or any modification of the data elements it maintains. Users can select the level of monitoring for each component. The MCM can also inform other GAME components about particular events through its "lists of interests" mechanism.

The *genetic algorithm libraries* comprise a collection of hierarchically organized modules containing predefined, parameterized applications; genetic algorithms; and genetic operators. New applications and algorithms can be created by simply combining the required components from the libraries and setting their parameters in a configuration file.

New applications and algorithms can be created by combining components from libraries and setting their parameters.

The environment is programmed in C++ and is available in source code for full user modification.

MicroGA. Marketed by Emergent Behavior, MicroGA is designed for a wide range of complex problems. It is small and easy to use, but expandable. Because the system is a framework of C++ objects, several pieces working together give the user some default behavior. In this, MicroGA is far from the library concept, in which a set of functions (or classes) is offered for incorporation in user applications. The framework is almost a ready-to-use application. MicroGA needs only a few user-defined parameters to start running. The package comprises a compiled library of C++ objects, three sample programs, a sample program with an Object Windows Library user interface (from Borland), and the Galapagos code-generation system. MicroGA runs on IBM PC-compatible systems with Microsoft Windows 3.0 (or later), using Turbo or Borland C++. It also runs on Macintosh computers.

The application developer can configure an application manually or by using Galapagos. This Windows-based code generator produces, from a set of custom templates and a little user-provided information, a complete stand-alone MicroGA application. It helps with the creation of a subclass derived from its "TIndividual" class, required by the environment to create the genetic data structure to be manipulated. Galapagos requests the number of genes for the prototype individual, as well as the range of possible values they can assume. The user can specify the evaluation function, but the Galapagos notation does not allow complex or nonmathematical fitness functions. Galapagos creates a class, derived from TIndividual, which contains the specific member functions as required by the user application.

Users can manually define applications requiring complex genetic data structures and fitness functions by having them inherit from the TIndividual class and writing the code for its member functions. After creating the application-dependent genetic data structure and fitness function, MicroGA compiles and links everything using the Borland or Turbo C++ compiler, and produces a file executable in Microsoft Windows.

MicroGA is very easy to use and lets users create GA applications quickly. However, for real applications the user must understand basic concepts of object-oriented programming and Windows interfacing.

Pegasus. The Programming Environment for Parallel Genetic Algorithms, or Pegasus, was developed at the German National Research Center for Computer Science. The toolkit can be used for programming a wide range of genetic algorithms, as well as for educational purposes. The environment is written in ANSI-C and is available for many different Unix-based machines. It runs on multiple instruction, multiple data parallel machines, such as transputers, and distributed systems of workstations. Pegasus is structured in four hierarchical levels:

- the user interface,
- the Pegasus kernel and library,
- compilers for several Unix-based machines, and
- the sequential and distributed or parallel hardware.

The user interface consists of three parts: the Pegasus script language, a graphical interface, and a user library. The user library has the same functionality as the Pegasus GA library. It lets the user define application-specific functions not provided by the system library, using the script language to specify the experiment. The user defines the application-dependent data structures, attaches the genetic operators to them, and specifies the I/O interface. The script language specifies the construction of subpopulations connected via the graphical interface.

The kernel includes base and frame functions. The *base functions* control the execution order of the genetic operators, manage communication among different processes, and provide I/O facilities. They build general frames for simulating GAs and can be considered as autonomous processes. They interpret the Pegasus script, create appropriate data structures, and describe the order of frame functions. Invoked by a base function, a *frame function* controls the execution of a single genetic operator. Frame functions prepare the data representing the genetic material and apply the genetic operators to it, according to the script specification. The library contains genetic operators, a collection of fitness functions, and I/O and control procedures. Hence, it gives the user validated modules for constructing applications.

Currently Pegasus can be compiled with the GNU C, RS/6000 C, ACE-C, and Alliant FX/2800 C compilers. It runs on Sun and IBM RS/6000 workstations, as well as on the Alliant FX/28 MIMD architecture.

Splicer. Created by the Software Technology Branch of the Information Systems Directorate at NASA Johnson Space Flight Center, with support from the Mitre Corporation,[19] Splicer is one of the most comprehensive environments available. We present it in the case study.

Case study: Splicer. The modular architecture includes three principal parts — the genetic-algorithm kernel, interchangeable representation libraries, and interchangeable fitness modules — and user interface libraries. It was originally developed in C on an Apple Macintosh and then ported to Unix workstations (Sun 3 and 4, IBM RS/6000) using X Windows. The three modules are completely portable.

The *genetic-algorithm kernel* comprises all functions necessary to manipulate populations. It operates independently from the problem representation (encoding), the fitness function, and the user interface. Some functions it supports are creation of populations and members, fitness scaling, parent selection and sampling, and generation of population statistics.

Interchangeable *representation libraries* store a variety of predefined problem-encoding schemes and functions, permitting the GA kernel to be used for any representation scheme. There are representation libraries for binary strings and permutations. These libraries contain functions for the definition, creation, and decoding of genetic strings, as well as multiple crossover and mutation operators. Furthermore, the Splicer tool defines interfaces to let the user create new representation libraries.

We expect the number and diversity of application-oriented systems to expand rapidly in the next few years.

Fitness modules are interchangeable and store fitness functions. They are the only component of the environment a user must create or alter to solve a particular problem. Users can create a fitness (scoring) function, set the initial values for various Splicer control parameters (for example, population size), and create a function that graphically displays the best solutions as they are found.

There are two user interface libraries: one for Macintoshes and one for X Windows. They are event-driven and provide graphical output in windows.

Stand-alone Splicer applications can be used to solve problems without any need for computer programming. However, to create a Splicer application for a particular problem, the user must create a fitness module using C. Splicer, Version 1.0, is currently available free to NASA and its contractors for use on government projects. In the future it will be possible to purchase Splicer for a nominal fee.

Future developments

As with any new technology, in the early stages of development the emphasis for tools is on ease of use. Application-oriented systems have a crucial role in bringing the technology to a growing set of domains, since they are targeted and tailored for specific users. Therefore, we expect the number and diversity of application-oriented systems to expand rapidly in the next few years. This development, coupled with the discovery of new algorithms and techniques, should bring an increase in algorithm-specific systems, possibly leading to general-purpose GAs. Algorithm libraries will provide access to efficient versions of these algorithms.

Interest in educational systems and demonstrators of GAs is rapidly growing. The contribution of such systems comes at the start of a new technology, but their usage traditionally diminishes as general-purpose systems mature. Thus we expect a decline in educational systems as sophisticated general-purpose systems become available and easier to use. General-purpose systems appeared very recently. With the introduction of Splicer, we expect commercial development systems in the near future. We should see programming environments for an expanding range of sequential and parallel computers, and more public-domain open-system programming environments from universities and research centers.

One high-growth area should be the association of genetic algorithms and other optimization algorithms in hybrid systems. Recently there has been considerable interest in creating hybrids of genetic algorithms and expert systems or neural networks. If a particularly complex problem requires optimization and either decision-support or pattern-recognition processes, then using a hybrid system makes sense. For example, neural networks and genetic algorithms have been used to train networks and have achieved performance levels exceeding that of the commonly used back-propagation model. GAs have also been used to select the optimal configurations for neural networks, such as learning rates and the number of hidden units and layers. By the end of the century, hybrid GA neural networks will have made significant progress with some currently intractable machine learning problems. Promising domains include autonomous vehicle control, signal processing, and intelligent process control.

Genetic algorithms are robust, adaptive search techniques that may be immediately tailored to real problems. The two major trends in future environments will be the exploitation of parallel GAs and the programming of hybrid applications linking GAs with neural networks, expert systems, and traditional utilities such as spreadsheets and databases. ■

Acknowledgments

We thank Lawrence Davis, Darrel Whitley, and Nicol Schraudolph for recommending GA programming environments for us to survey. We also thank Frank Hoffmeister, Hans-Michael Voigt, and Joachim Born for their advice. Finally, we acknowledge our colleagues Jason Kingdon and Suran Goonatilake for commenting on early drafts of this article.

References

1. J.H. Holland, *Adaptation in Natural and Artificial Systems*, Univ. of Michigan Press, Ann Arbor, Mich., 1975.

2. D.E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, Mass., 1989.

3. K.A. DeJong, *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, doctoral dissertation, Univ. of Michigan, Ann Arbor, Mich., 1975.

4. J.J. Grefenstette, "Genesis: A System for Using Genetic Search Procedures," *Proc. Conf. Intelligent Systems and Machines*, 1984, pp. 161-165.

5. L. Davis, *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, 1991.

6. H. Mühlenbein, "Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization," *Proc. Third Int'l Conf. Genetic Algorithms*, Morgan Kaufmann, San Mateo, Calif., 1989, pp. 416-421.

7. M. Gorges-Schleuter, "Asparagos: An Asynchronous Parallel Genetic Optimisation Strategy," *Proc. Third Int'l Conf. Genetic Algorithms*, Morgan Kaufmann, San Mateo, Calif., 1989, pp. 422-427.

8. H. Mühlenbein, "Evolution in Time and Space — The Parallel Genetic Algorithm," in *Foundations of Genetic Algorithms*, G. Rawlins, ed., Morgan Kaufmann, San Mateo, Calif., 1991, pp. 316-337.

9. R. Tanese, "Distributed Genetic Algorithms," *Proc. Third Int'l Conf. Genetic Algorithms*, Morgan Kaufmann, San Mateo, Calif., 1989, pp. 434-440.

10. I. Rechenberg, *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution* [*Evolutionary Strategy: Optimization of Technical Systems According to the Principles of Biological Evolution*], Frommann-Holzboog Verlag, Stuttgart, Germany, 1973.

11. H.P. Schwefel, "Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie" [Numerical Optimization of Computer Models by Means of the Evolutionary Strategy], *Interdisciplinary Systems Research*, Vol. 26, Birkäuser, Basel, Switzerland, 1977.

12. F. Hoffmeister and T. Bäck, "Genetic Algorithms and Evolution Strategies: Similarities and Differences," Tech. Report "Grüne Reihe," No. 365, Dept. of Computer Science, Univ. of Dortmund, Germany, 1990.

13. F. Hoffmeister, "The User's Guide to Escapade 1.2: A Runtime Environment for Evolution Strategies," Dept. of Computer Science, Univ. of Dortmund, Germany, 1991.

14. D. Whitley and J. Kauth, "Genitor: A Different Genetic Algorithm," *Proc. Rocky Mountain Conf. Artificial Intelligence*, 1988, pp. 118-130.

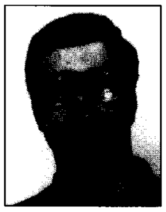
15. N.N. Schraudolph and J.J. Grefenstette, "A User's Guide to GAUCSD 1.2," Computer Science and Eng. Dept., Univ. of California, San Diego, 1991.

16. H.M. Voigt, J. Born, and J. Treptow, "The Evolution Machine Manual — V 2.1," Inst. for Informatics and Computing Techniques, Berlin, 1991.

17. M. Hughes, "Genetic Algorithm Workbench Documentation," Cambridge Consultants, Cambridge, UK, 1989.

18. G. Robbins, "Engeneer — The Evolution of Solutions," *Proc. Fifth Ann. Seminar Neural Networks and Genetic Algorithms*, IBC Technical Services Ltd., London, 1992, pp. 218-232.

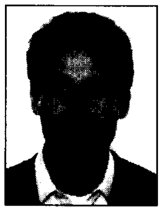
19. NASA Johnson Space Flight Center, "Splicer — A Genetic Tool for Search and Optimization," *Genetic Algorithm Digest*, Vol. 5, Issue 17, 1991, p. 4.



José L. Ribeiro Filho is a research staff member in the Núcleo de Computação Eletrônica at the Universidade Federal do Rio de Janeiro, Brazil. His research interests include computer architectures, parallel processing, communication systems, and optimization techniques such as genetic algorithms.

Ribeiro Filho received an MS in computer science in 1989 from the Federal University of Rio de Janeiro and is now working on a PhD at University College London.

Philip C. Treleaven is a professor of computer science at University College London. His research interests are in neural computing, computing applications in finance, and fifth-generation computers for artificial intelligence. He has consulted for IBM, DEC, GEC, Fujitsu, Mitsubishi, Philips, Siemens, and Thomson, and acted as adviser to government ministers in Japan, Germany, France, Korea, and other countries. Among the European collaborative research projects he is involved with is the Galatea neural computing project.



Cesare Alippi is working on a PhD in artificial intelligence at Politecnico di Milano, where he is analyzing the sensitivity of neural networks to neural value quantization. His other research interests include genetic algorithms and fault tolerance. Previously he was a researcher in the Department of Computer Science at University College London. Alippi received a BS degree in electronic engineering from Politecnico di Milano in 1990.

Readers can contact Ribeiro Filho at the Department of Computer Science, University College London, Gower St., London WC1E 6BT, UK; e-mail j.ribeirofilho@cs.ucl.ac.uk.