Genetic Algorithm Programming Environments

Jose Ribeiro Filho, Cesare Alippi and Philip Treleaven
Department of Computer Science – University College London

ABSTRACT

Interest in Genetic algorithms is expanding rapidly. This paper reviews software environments for programming Genetic Algorithms (GAs). As background, we initially preview genetic algorithms' models and their programming. Next we classify GA software environments into three main categories: Application-oriented, Algorithm-oriented and Tool-Kits.

For each category of GA programming environment we review their common features and present a case study of a leading environment.

Keywords – Programming Environments, Genetic Algorithms.

To appear in the IEEE COMPUTER Journal.

Table of Contents

1. Introduction

Evolution is a remarkable problem solving machine. *Genetic Algorithms* are an attractive class of computational models that attempt to mimic the mechanisms of natural evolution to solve problems in a wide variety of domains.

The theory behind Genetic Algorithms was proposed by John Holland in his landmark book *Adaptation in Natural and Artificial Systems* published in 1975 [8]. In conjunction with the GA theory, he developed the concept of Classifier Systems, a machine learning technique. Classifier Systems are basically induction systems with a genetic component [3]. Holland's goal was two-fold: firstly, to explain the adaptive process of natural systems [3] and secondly, to design computing systems capable of embodying the important mechanisms of natural systems [3]. Pioneering work of Holland [8], Goldberg [3], De Jong [2], Grefenstette [5], Davis [1], Mühlenbein [10] and others is fuelling the spectacular growth of GAs.

GAs are particularly suitable for the solution of complex optimisation problems, and consequently are good for applications that require adaptive problem solving strategies[1]. In addition, GAs are inherently parallel, since their search for the best solution is performed over genetic structures (building blocks) which can represent a number of possible solutions. Furthermore GAs' computational models can be easly parallelised. Many parallel models have been proposed recently [4,11,17] which attempt to exploit GA's parallelism on massively parallel computers and distributed systems.

1.1. Classes of Search Techniques

Genetic Algorithms are one very important class of search techniques. Search techniques in general, as illustrated in figure 1 can be grouped into three broad classes [3]: Calculus-based, Enumerative and Guided Random search.

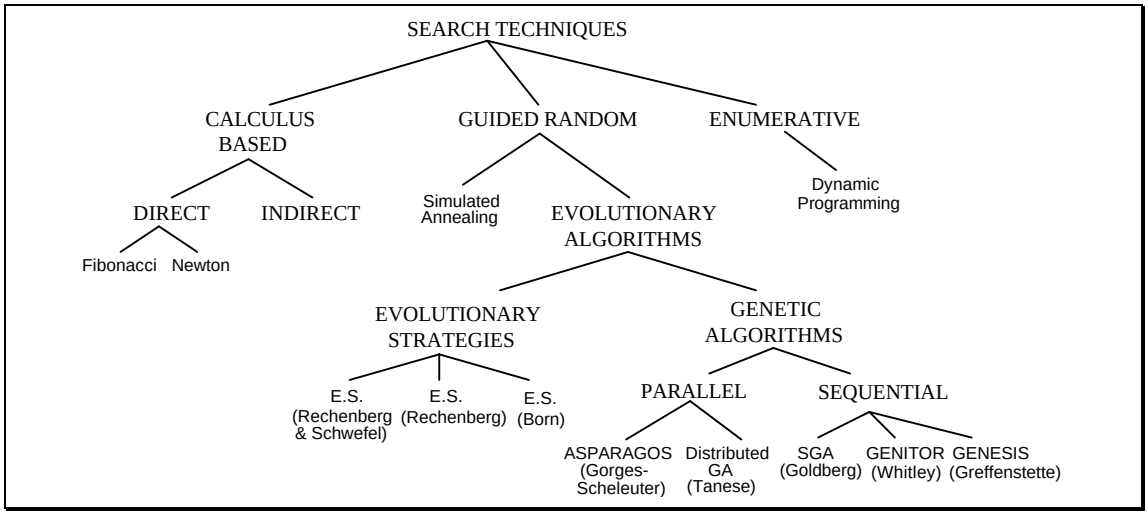

Figure 1 - Classes of Search Techniques

[1] A survey of GA applications is beyond the scope of this paper, and the interested reader is referred to [1,3]

Calculus-based techniques use a set of necessary/sufficient conditions to be satisfied by the optimal solutions of an optimisation problem. These techniques sub-divide into indirect and direct methods. *Indirect* methods look for local extrema by solving the usually non-linear 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 the search to points with zero slope in all directions. *Direct* methods, such as Newton and Fibonacci, seek extrema by "hopping" around the search space and assessing the gradient of the new point, which guides the direction of the search. This is simply the notion of *Hill-Climbing* which finds the best local point by "climbing" the steepest permissible gradient. However, these techniques can only be employed on a restricted set of "well behaved" problems.

Enumerative techniques search every point related to an objective function's domain space (finite or discretised), 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 on their scope, being able to solve very complex problems. Two major sub-classes are: Simulated Annealing and Evolutionary Algorithms, although both are evolutionary processes. 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 biological-inspired operations. These techniques sub-divide, in turn, into Evolution Strategies and Genetic Algorithms. *Evolution Strategies* were proposed by Rechenberg and Schwefel [12,15] in the early seventies. They present the ability to adapt the process of "artificial evolution" to the requirements of the local response surface[2]. This means that ESs are able to adapt their major strategy parameters according to the local topology of the objective function [7]. This represents a significant difference to traditional GAs.

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 [1] where the whole population is replaced in each generation. Conversely, the *steady-state* policy [1] used by many subsequent GAs employ a selective replacement for the population. It is possible, for example, to keep one or more individuals within the population for several generations, while those individuals sustain a better fitness than the rest of the population.

1.2. Survey Structure

Having reviewed search techniques, we next present our survey of GA programming environments. The environments presented here, are those most readily accessible in the literature.

To make the paper self-contained, we start by introducing GA models and their programming. This is followed by our survey of GA programming environments. We have grouped environments into three major classes according to their specific objectives: Application-oriented, Algorithm-oriented, and Tool Kits. *Application-oriented systems* are "black box" environments designed to hide the details of GAs and help the user in developing applications for specific domains, such as *Finance*, *Scheduling*, etc. These application domains form a natural subdivision. *Algorithm-oriented systems* are based on specific genetic algorithm models, such as the GENESIS algorithm. This class may be further sub-divided into:

[2]For a formal description on Evolutionary Strategy refer to[6].

Algorithm-specific systems which support a single genetic algorithm, and *Algorithm Libraries* which support a group of algorithms in a library format. Lastly, *Tool Kits* are flexible environments for programming a range of GAs and applications. These systems sub-divide into: *Educational systems* which introduce GA concepts to novice users, and *General-purpose systems* to modify, develop and supervise a wide range of genetic operators, genetic algorithms and applications.

For each class and sub-class, a review of the available environments is presented with a description of their common features and requirements. As a case study, one specific system per class is examined in more detail. Finally, we discuss the likely future developments of GA programming environments.

2. Genetic Algorithms

A Genetic Algorithm is a computational model that emulates biological evolutionary theories to solve optimisation 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 on to a set of (binary) strings[3], each string representing a potential solution. The GA then manipulates the most promising strings searching for improved solutions. A GA operates typically through a simple cycle of four stages:

i) creation of a "population" of strings,
ii) evaluation of each string,
iii) selection of "best" strings, and
iv) genetic manipulation, to create the new population of strings.

Figure 2 shows these four stages using the biologically inspired GA terminology. In each cycle a *new generation* of possible solutions for a given problem is produced. At the first stage, an *initial population* of potential solutions is created as a starting point for the search process. Each element of the population is *encoded* into a string (the *chromosome*), to be manipulated by the *genetic operators*. In the next stage, the performance (or *fitness*) of each individual of the population 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/selection process is called *reproduction*.

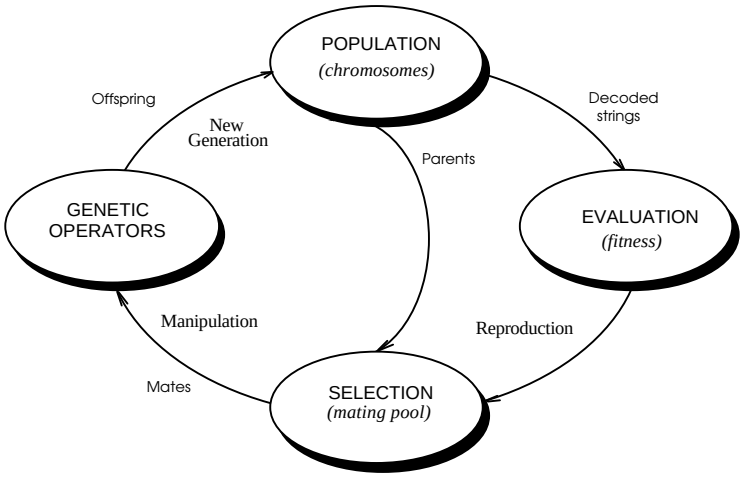

Figure 2 - The GA cycle

The manipulation process employs 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, namely crossover and mutation. *Crossover* is responsible for recombining the genetic material of a population. The selection process associated to recombination, assure that special genetic

[3]Although binary strings are typical, other alphabets such as real numbers are also used.

structures, called "building blocks", are retained for future generations. The building blocks then represent the most fitted genetic structures in a population. Nevertheless, the recombination process alone can not avoid the loss of promising building blocks in the presence of other genetic structures, which could lead to local minima. Also, it is not capable to explore search space sections not represented in the population's genetic structures. The *mutation* operator comes then into action. It introduces new genetic structures in the population by randomly modifying some of its building blocks. It helps the search algorithm to escape from local minima's traps. Since the modification introduced by the mutation operator is not related to any previous genetic structure of the population, it allows the creation of 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. After the *crossover point* has been randomly chosen, the portions of the *parent* strings P1 and P2 are swapped to produce the new *offspring* strings O1 and O2. For instance, figure 3 shows the crossover operator being applied to the fifth and sixth elements of the string.

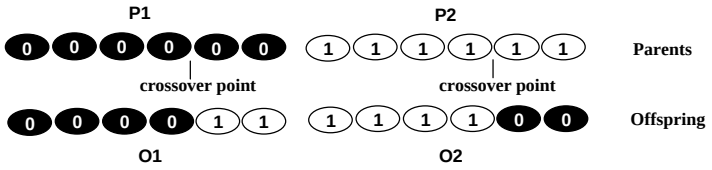

Figure 3 - Crossover

Mutation is implemented by occasionally altering a random bit in a string. Figure 4 presents the mutation operator being applied to the fourth element of the string.

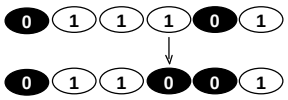

Figure 4 - Mutation

A number of different genetic operators have been introduced since this basic model was proposed by Holland. 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, genetic edge recombination, etc.

The offspring produced by the genetic manipulation process originate the next population to be evaluated. Genetic Algorithms can either replace a whole population (generational approach) or theirs less-fitted members only (steady-state approach). The *creation-evaluation-selection-manipulation* cycle is repeated until a satisfactory solution to the problem is found.

The description of the genetic algorithm computational model given in this section presented an overall idea of the steps needed to design a genetic algorithm. However, real implementations, as exemplified in the next section, have to consider a number of problem-dependent parameters such as the population size, crossover and mutation rates, convergence criteria, etc. GAs are very sensitive to most of these parameters, and the discussion of the methods to help in setting them up is beyond the scope of this paper.

2.1. Sequential GAs

To illustrate the implementation of a sequential genetic algorithm we will use the simple function optimisation example given in Goldberg [3], and examine its programming in C.

The first step in optimising the function $f(x)=x^2$, over the interval (i.e. parameter set) [0–31], is to encode the parameter set *x*, for example as a five digit binary string {00000–11111}. Next we need to generate our initial population of 4 potential solutions, shown in table 1, using a random number generator.

Table 1 - Initial strings and fitness values

Initial Population	x	f(x) (fitness)	strength (% of Total)
0 1 1 0 1	13	169	14.4
1 1 0 0 0	24	576	49.2
0 1 0 0 0	8	64	5.5
1 0 0 1 1	19	361	30.9
Sum_Fitness =		1170	(100.0)

To program this GA function optimisation we declare the population pool as an array with four elements, as in figure 5, and then initialise the structure using a random generator as shown in figure 6.

```
#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 variables declarations in C

```
initialise_population()
{
    randomise();                                    /* random generator set-up */
    for (i=0; i < POPULATION_SIZE; i++)
            encode(i, random(pow(2.0,CHROM_LENGTH));
}
```

Figure 6 - Initialisation routine

Having initialised the GA, the next stage is reproduction. Reproduction evaluates and selects pairs of strings for *mating* – for instance using a "roulette wheel" method [3] – 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 string 10011 are selected.

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

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

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

Figure 7 - Selection function

Next we apply the crossover operator, as illustrated in table 2.

Table 2 - Mating pool strings and crossover

Mating Pool	Mates	Swapping	New Population
0 1 1 0 1	1	0 1 1 0 [1]	0 1 1 0 0
1 1 0 0 0	2	1 1 0 0 [0]	1 1 0 0 1
1 1 0 0 0	2	1 1 [0 0 0]	1 1 0 1 1
1 0 0 1 1	4	1 0 [0 1 1]	1 0 0 0 0

Crossover operates in two steps (see figure 8). Firstly, it determines whether crossover is to occur on a pair of strings by using a *flip* function; tossing a biased coin. If heads (true) with probability *pcross*, the strings are swapped; the *crossover_point* being 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.

```
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

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 uses the biased coin toss (*flip*) with probability *pmut* to determine whether to change a bit or not.

```
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

A new population has now been generated (see table 3), and a termination test is applied. Termination criteria may include: the simulation time being up, the number of generations exceeded, or the convergence criterion satisfied. In the example, we might set the number of generations to 50, and the convergence as an average fitness improvement of less than 5%, between generations. For the initial population, the average is 293 (i.e. (169+576+64+361)÷4), while for the new population it has improved to 439 (i.e. 66%).

Table 3 - Second generation and its fitness values

Initial Population	x	f(x) (fitness)	strength (% of Total)
0 1 1 0 0	12	144	8.2
1 1 0 0 1	25	625	35.6
1 1 0 1 1	27	729	41.5
1 0 0 0 0	16	256	14.7
Sum_Fitness =		1754	(100.0)

A complete C code listing of this simple example is given in Appendix A, for the interested reader.

2.2. Parallel GAs

The Genetic Algorithms paradigm offer intrinsic parallelism when looking for the best solution on a large search space, as demonstrated on Holland's *schema theorem*[8]. Besides the intrinsic parallelism, GAs computational models can also exploit several levels of parallelism, due to the natural independence of the genetic manipulation operations.

A parallel GA is generally formed by *parallel components* each responsible for manipulating sub-populations. There are two classes of PGAs employing *centralised* and *distributed* selection mechanisms. With centralised selection mechanisms, a single selection operator works on the global population (of sub-populations). Thus the PGA has a synchronous selection stage.

With distributed mechanisms, each parallel component has its own copy of the selection operator. In addition, each component communicates its best strings to a sub-set of the other components. This is supported by a *migration* operator, and a *migration frequency* defining the communication interval. These PGAs have an asynchronous selection stage. The ASPARAGOS [4] algorithm is an excellent example of this class.

As an illustration of parallel GAs, figure 10 shows a skeleton *C-like* program, based on the ASPARAGOS algorithm, for the simple function optimisation discussed in the previous section. In this parallel program the statements for *initialisation, selection, crossover* and

mutation remain almost the same as for the sequential program. For the main loop, PARallel sub-populations 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.

```
#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_NEIGHBOURS       2
#define     MIGRATION_FREQ          5
#define     NUM_OF_EMIGRANTS        2

main()
{
PAR         for (i=0; i<SUB_POP_SIZE; i++)      /* Parallel execution      */
                                                /* over sub-populations */
SEQ         {       initialise(); }

            do;
            {
                    for (j=0; j<MIGRATION_FREQ; j++)
SEQ                 {       selection(..)       /* evaluate & 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_NEIGHBOURS; j++)
PAR                 {       send_emigrants(..);
                            receive_emigrants(..);
                    }
            }
            while (generations <= MAX_GEN);
}
```

Figure 10 - Parallel GA with migration

Following this brief review of GAs' concepts and implementations, we next survey GA programming environments.

3. Taxonomy for GA Programming Environments

For our review of genetic algorithms programming environments, we utilise a simple taxonomy of three major classes: Application-oriented systems, Algorithm-oriented systems and Tool Kits.

- **Application-oriented systems** are essentially "black box" programming systems, hiding the GA implementation details, and are targeted at business professionals. Some of these systems support a range of applications, such as *scheduling, telecommunications*, etc. (a good example being PC/BEAGLE); others focus on a specific domain, such as *finance* (as with OMEGA).
- **Algorithm-oriented systems** are programming systems which support specific genetic algorithms. They sub-divide into:
  - *Algorithm-specific systems* – which contain a single genetic algorithm; the classic example being GENESIS [5].
  - *Algorithm Libraries* – where a variety of genetic algorithms and operators are grouped in library; as in Lawrence Davis' OOGA[1].

  Algorithm-oriented systems are often supplied in source code and can be easily incorporated into user applications.
- **Tool Kits** are programming systems that provide many programming utilities, algorithms and genetic operators that can be used for a wide range of application domains. These programming systems sub-divide into:
  - *Educational systems* – to help the novice user to obtain a hands-on introduction on GA concepts. Typically these systems support a small set of options for configuring an algorithm. See GA Workbench [9].
  - *General-purpose systems* – to provide a comprehensive set of tools for programming any GA and application. These systems may even allow the expert user to customise any part of the software, as in Splicer.

As an illustration of our Taxonomy, table 4 lists the GA programming environments examined in the next sections, according to their categories. For each category we initially present a generic system overview, then briefly review example systems, and finally examine one system in more detail, as a case study. Examples of parallel environments like GAUCSD, PeGAsuS, and GAME are also covered, but there are no commercial parallel environments currently available. A more comprehensive list of programming environments and their developers is given in appendix B.

Table 4 - Programming Environments and their categories

Application Oriented	**Algorithm Oriented**		**Tool Kits**	
	Algorithm-Specific	Algorithm-Libraries	Educational	General Purpose
EVOLVER	ESCAPADE	*EM*	GA Workbench	EnGENEer
OMEGA	GAGA			GAME
PC/BEAGLE	GAUCSD			*Micro*GA
XpertRule	GENESIS	OOGA		PeGAsuS
GenAsys	GENITOR			Splicer

4. Application-oriented systems

Application-oriented systems are designed for use by business professionals who wish to utilise genetic algorithms in specific applications domains, without having to acquire detailed knowledge of the workings of genetic algorithms.

As we have seen with expert systems and neural networks, many potential users of a novel computing technique, such as genetic algorithms, are only interested in the applications, rather than the details of the technique. For example, a manager in a trading company may wish to optimise its delivery scheduling. By using an application-oriented programming environment, it is possible to configure a particular application for schedule optimisation based on the Travelling Salesman Problem (TSP) model, without having to know the encoding technique nor the genetic operators involved.

Overview

A typical Application-oriented environment is analogous to a Spreadsheet or Word-processing utility. It comprises a menu-driven interface (tailored to business users) giving access to a suite of parameterised modules (targeted at specific domains). The user interfaces provide menus to configure an application, monitor its execution, and, in certain cases, program an application. Help facilities are also provided.

Survey

Application-oriented systems follow many innovative strategies. Systems, such as PC/BEAGLE and XpertRule GenAsys, are expert systems using GAs to generate new rules to expand their knowledge base of the application domain. EVOLVER is a companion utility for Spreadsheets; and systems like OMEGA, are targeted at financial applications.

EVOLVER — is an add-on utility that works within the Excel, WingZ and Resolve spreadsheets on Macintosh and PC computers. It is being marketed by Axcélis Inc., who describes it as "an optimisation program that extends mechanisms of natural evolution to the world of business and science applications". The user starts with a model of his system in the spreadsheet and calls the EVOLVER program from a menu. After filling a dialogue box with the information required (e.g. cell to minimise/maximise) the program starts working, evaluating thousands of scenarios automatically until it is sure it has found an optimal answer. The program runs in background freeing the user to work in the foreground. When the program 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 interfacing with spreadsheet in business. In an attempt to improve the system and expand its market, Axcélis introduced Evolver 2.0 that is being shipped with many tool-kit-like features. The new version is capable to integrate with other applications, besides spreadsheets. Also it offers more flexibility in accessing the "Evolver Engine" from any MS-Windows application capable of calling a Dynamic Link Library (DLL).

OMEGA — the OMEGA Predictive Modelling System, marketed by KiQ Limited, is a powerful approach to developing predictive models. It exploits advanced genetic algorithms techniques to create a tool which is "flexible, powerful, informative and straightforward to use". OMEGA is geared to the financial domain and can be applied in the following sectors: Direct Marketing, Insurance, Investigations (case scoring) and Credit Management. The

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

PC/BEAGLE — produced by Pathway Research Ltd, is a rule-finder program that 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 — is an expert system shell with embedded genetic algorithms, marketed by Attar Software. This GA expert system is targeted to solve scheduling and design applications. The system combines the power of genetic algorithms in evolving solutions with the power of rule-base programming in analysing the effectiveness of solutions. Rule-base programming can also be used to generate the initial solutions for the genetic algorithm and for post optimisation planning. Some examples of design and scheduling problems that can be solved by this system includes: optimisation of design parameters in electronic and avionics industries, route optimisation in the distribution sector, production scheduling in manufacturing, etc.

As our case study, we will examine PC/BEAGLE.

Case Study — **PC/BEAGLE**

PC/BEAGLE is a rule-finder program that examines a database of examples and uses machine-learning techniques to create a set of decision rules for classifying those examples, turning data into knowledge. In particular, the software analyses 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 that are 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 Optimisation Tester*) tests an initial batch of user-suggested rules.
- **HERB** (*Heuristic Evolutionary Rule Breeder*) generates decision rules by Naturalistic Selection, using GA philosophy (*ranking)* mechanisms are also supported).
- **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 be used to convert a BEAGLE rule-file into a language such as Pascal or Fortran; In this form the knowledge gained may be used by other software.

PC/BEAGLE accepts data in ASCII format, with items delimited either by commas, spaces or tabs. Rules are produced as logical expressions. The system is a highly versatile package covering a wide range of applications. Insurance, weather forecasting, finance and forensic science are some examples.

The software requirements are an IBM/PC compatible computer with at least 256K bytes RAM and MS-DOS or PC-DOS operating system version 2.1 or later.

5. Algorithm-oriented systems

5.1. Algorithm-specific systems

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.

Overview

In general, Algorithm-specific systems come in source code and allow the expert user to make alterations for specific requirements. They present a modular structure providing a high degree of modifiability. In addition, user interfaces are frequently rudimentary, often command-line driven. Typically the code has been developed in universities and research centres, and are available free over world-wide computer research networks.

Survey

The most well known programming system in this category is the pioneering GENESIS [5] 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 [19] is another influential system that has been successfully used. GAUCSD allows parallel execution by distributing several copies of a GENESIS-based GA into UNIX machines in a network. Finally, ESCAPADE [7] employs a somewhat different approach – being based on an Evolutionary Strategy (see Section 1.1) – as discussed below.

ESCAPADE — **E**volution **S**trategies **cap**able of **ad**aptive **e**volution — this software package provides a sophisticated environment for a particular class of Evolutionary Algorithms, called *Evolution Strategies*. ESCAPADE is based upon KORR, Schwefel's implementation of a $(\mu \overset{+}{,} \lambda)$ – evolutionary strategy. The system provides an elaborated set of monitoring tools to gather data from an optimisation run of KORR. According to the author, it should be possible to incorporate a different implementation of an ES or even a GA into the system using its runtime support. The program structure is separated into several rather independent components that support the various tasks during a simulation run. The major modules are: Parameter Set-up, Runtime Control, KORR, Generic Data Monitors, Customised Data Monitors, and Monitoring Support.

During an optimisation run the monitoring modules are invoked by the main algorithm (KORR or some other ES or GA implementation) to realise the logging of internal quantities. The system is not equipped with any kind of graphics interface. All parameters for a particular simulation are passed over as command line options. In the output, each data monitor writes its data into separate log files.

GAGA — **G**enetic **A**lgorithms for **G**eneral **A**pplication — was originally programmed by Hillary Adams, University of York, in Pascal. The program was later modified by Ian Poole and translated to the C language by Jon Crowcroft at University College London. It is a task independent genetic algorithm. The user must supply the target function to be optimised (minimised/maximised) and some technical GA parameters, and wait for the output. It is suitable for the minimisation of many "difficult" cost functions.

GAUCSD — This software package was developed by Nicol Schraudolph at the University of California, San Diego [14]. The system is based on GENESIS 4.5 and runs on Unix, MS-DOS, CrayOs and VMS platforms; but presumes a Unix environment. It 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 the evaluation function 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 (DPE) technique developed by Schraudolph, which facilitates a radical reduction of the gene length while keeping the desired level of precision for the results. It is possible to run the system in background at low priority using the *go* command. This command can also be used to execute GAUCSD in 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 in the remote host. Experiments can be queued in files, distributed to several hosts and executed in parallel. The *ex* command will notify the user via write or mail when all experiments are completed. 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. GAUCSD is clearly a very powerful system.

GENESIS — **GENE**tic Search Implementation System — was written by John Grefenstette [5] to promote the study of genetic algorithms for function optimisation. It has been under development since 1981, and has been widely distributed to the research community since 1985. The software package is a set of routines written in the "C" language. To build their own genetic algorithm, the user has only to provide a routine with the fitness function and link it with the other routines. It is also possible to modify or add new modules (e.g. genetic operators, 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 algorithms and operators. It was primarily developed to work in a scientific environment offering a suitable software tool for research. It provides a high modifiability degree and a variety of statistical information on outputs.

GENITOR — **GEN**etic **I**mplemen**TOR** — is a modular GA package containing examples for floating-point, integer and binary representations. Its features include many sequencing operators as well as sub-population modelling. This software package is, in fact, the implementation of the GENITOR algorithm developed by Darrel Whitley [19]. The algorithm presents two major differences from standard genetic algorithms. The first one is the explicit use of ranking. Reproductive trials are allocated according to the rank of the individual in the population rather than using fitness proportionate reproduction. The second difference is that GENITOR abandons the generational[4] approach and reproduces new genotypes on an individual basis. It does so in such a way that parents and offspring can, typically, co-exist. The newly created offspring replaces the lowest ranking individual in the population rather than a parent. This approach is known as Steady State. GENITOR only produces one new genotype at a time, so inserting a single new individual is relatively simple. Furthermore, the insertion automatically ranks the individual with relation to the existing pool — no further measure of the relative fitness is needed.

Case Study — **GENESIS**

GENESIS [5] is the most well known software package for genetic algorithm development and simulation. It is now on version 5.0, which is available from The Software Partnership company. GENESIS runs on most machines with a C compiler. The present version runs successfully on both Sun workstations and IBM/PC's compatible computers, according to the

[4]The whole population is replaced in each generation.

author. The code has been designed to be portable, but minor changes may be necessary for other systems. The system provides the fundamental procedures for *genetic selection, crossover* and *mutation*. Since GAs are task independent optimisers, the user must provide only an evaluation function which returns a value when given a particular point in the search space.

GENESIS has three levels of representation for the structures it is evolving. The lowest level, or *packed* representation, is used to maximise both space and time efficiency in manipulating structures. In general, this level of representation is transparent to the user. The next level, or the *string* representation, represents structures as null-terminated *arrays of chars*. This structure is provided for users who wish to provide an arbitrary interpretation of the genetic structures, for example, non-numeric concepts. The third level, or *floating-point* representation, is the appropriate level for many numeric optimisation problems. At this level, the user can think about genetic structures as vectors or real numbers. For each parameter, or *gene*, the user specifies its range, its number of values, and its output format. The system then automatically lays out the string representation, and translates between the user-level genes and lower representation levels. The system contains five major modules:

- *Initialisation* – the initialisation procedure sets up the initial population. It is possible to "seed" the initial population with heuristically chosen structures. The rest of the population is filled with random structures. It is also possible to initialise the population with real numbers.
- *Generation* – this is responsible for the execution of the selection, crossover, mutation, and evaluation procedures; and perform some data collection.
- *Selection* – this is the process of choosing structures for the next generation from the structures in the current generation. The default selection procedure is a stochastic procedure, which guarantees that the number of offspring of any structure is bounded by the floor and the ceiling of the (real-valued) expected number of offspring. The procedure is based on the roulette wheel algorithm. It is also possible to perform selection based on a ranking algorithm. Ranking helps prevent premature convergence by preventing *super* individuals from taking over the population within a few generations.
- *Mutation* – after the new population is selected, mutation is applied to each structure in the new population. Each position is given a chance (*mutation rate*) of undergoing mutation. If mutation does occur, a random value is chosen from {0,1} for that position. If the mutated structure differs from the original one, it is marked for evaluation.
- *Crossover* – exchanges alleles among 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. 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 customised 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 set-up program prompts for a number of input parameters. All the information is stored in files for future use. It is possible to set the type of representation, the 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 ga and produces a description of the algorithm performance. It summarises the mean, variance and range of several measurements, including on-line performance, off-line performance, average performance of the current population, and the current best value.

5.2. Algorithm Libraries

Algorithm Libraries provide a powerful collection of parameterised genetic algorithms and operators generally coded in a common language, and so are easily incorporated in user applications.

Overview

These systems are modular, allowing the user to select a variety of algorithms, operators and parameters to solve a particular problem. Their parameterised libraries provide the ability to use different models (algorithms, operators and parameter settings) to 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 is designed to facilitate the configuration and manipulation of the models as well as to present the results in different shapes (tables, graphics, etc.).

Survey

The two leading algorithm-libraries are *EM* and OOGA. Both systems provide a comprehensive library for genetic algorithms, and *EM* also supports evolution strategies simulation. In addition, OOGA can be easily tailored for specific problems. It runs in Common Lisp and CLOS (Common Lisp Object System), an object oriented extension of the Common Lisp.

EM — ***E***volution ***M***achine — has been developed by Hans-Michael Voigt, Joachim Born and Jens Treptow [18] at the Institute for Informatics and Computing Techniques in Germany. The *EM* simulates natural evolution principles to obtain efficient optimisation procedures for computer models. The evolutionary methods included in *EM* were chosen to provide algorithms with different numerical characteristics. The programming environment supports the following algorithms:

i) Evolution Strategy by Rechenberg [12],

ii) Evolution Strategy by Rechenberg & Schwefel [12,15],

iii) Evolution Strategy by Born,

iv) Simple Genetic Algorithm by Goldberg [3], and

v) Genetic Algorithm by Voigt and Born [18].

To run a simulation session the user provides the fitness function coded in the "C" programming language. The system calls the compiler and the linker to produce an executable file containing the selected algorithm and the user supplied fitness function.

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

OOGA — **O**bject **O**riented **G**enetic **A**lgorithm — is a simplified version of the Lisp-based software that has been developed since 1980 by Lawrence Davis. It was mainly created as a support for Davis' book [1] but can also be used to develop and test customised or new genetic algorithms and genetic operators.

Case Study — **OOGA**

OOGA is a system designed so that each of the techniques employed by a GA is an object that can be modified, displayed or replaced in an object-oriented fashion. The highly modular OOGA architecture makes it easy for the user to define and use a variety of genetic algorithm techniques, by incrementally writing and modifying components in Common Lisp. The files in the OOGA system contain descriptions of several techniques used by genetic algorithm researchers, but they are not exhaustive. OOGA contains three major modules:

- *Evaluation Module* which has the evaluation (or fitness) function that measures the worth of any chromosome on the problem to be solved;
- *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 (e.g. binary, real number, etc.), initialisation (e.g. random binary, random real, normal distribution, etc.) and deletion (e.g. delete all, delete last, etc.);
- *Reproduction Module* has a set of genetic operators responsible for selecting and creating new chromosomes during the reproduction process. This module allows genetic algorithm configurations with more than one genetic operator as well as its parameters' settings. The system creates a list with the user selected operators and executes then in sequence. There are a number of genetic operators for selection (e.g. roulette wheel), crossover (e.g. one- and two-point crossover, mutate-and-crossover) and mutation. All the parameters such as bit mutation rate and crossover rate, can be set by the user.

The last two modules are, in fact, a library of several different techniques which enables the user to configure a particular genetic algorithm. When the genetic algorithm is run, the Evaluation, Population and Reproduction modules work together to effect the evolution of a population of chromosomes towards the best solution.

The system also supports some normalisation techniques (e.g. linear normalisation) and parameterisation techniques, allowing the alteration of the genetic operators relative fitness over the course of the run.

6. Tool Kits

6.1. Educational systems

Educational programming systems are designed for the novice user to obtain hands-on introduction to genetic algorithms concepts. They typically provide a rudimentary graphic interface and a simple configuration menu.

Overview

Educational systems are typically implemented on PCs for portability and low cost reasons. For ease of use, they have a nice graphical interface and are fully menu-driven. GA Workbench is one of the best examples of this class of programming environments.

Case Study — **GA Workbench**

GA Workbench [9] has been developed by Mark Hughes from Cambridge Consultants Ltd. It is a mouse-driven interactive GA program that runs on MS-DOS/PC-DOS microcomputers. The system is aimed at people wishing to understand and get hands-on GA practice. Evaluation functions are drawn on screen, using a mouse. The system produces run-time plots of GA population distribution, peak and average fitness. Many useful population statistics are also displayed. It is possible to change a range of parameters including the settings of the genetic operators, the population size, breeder selection, etc.

Its graphical interface needs a VGA or EGA graphic adapter and it divides the screen into seven fields:

- *A Command Menu* - this is a menu-bar that has general commands to start or stop a GA execution, as well as let the user enter the target function.
- *Target Function Graph* - after selecting the "Enter Targ" command from the command menu, the user inputs the target function by drawing it on a graph using the mouse cursor.
- *Algorithm Control Chapter* - this field is called chapter because it can contain several pages, but only one page is visible at a time. It initially displays a page called "Simple Genetic Algorithm". Pages can be flipped through, forwards or backwards, by clicking the left mouse button on the arrows in the top high hand corner of the chapter. Following is a brief description of the available pages:
    - Simple Genetic Algorithm Page - this page shows a number of input variables used to control the operation of the algorithm. 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.
    - General Program Control Variables Page - this 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, determine the frequency with which the population distribution histogram is updated or seeds the random number generator.

- *Output Variables Box* - this contains the current values of a number of variables relating to the current algorithm. For the Simple Genetic Algorithm, a counter of generations is presented as the optimum fitness value, the current best fitness, the average fitness, the optimum *x*, current best *x*, and the average *x*.
- *Population Distribution Histogram* - this graph shows the genetic algorithm's distribution of organisms by value of *x*. The histogram is updated according to the frequency set in the general control variables page.
- *Output Graph* - this field is used to display plots of several output variables against time.
- *Axis Value Box* - this box is used in combination with the mouse cursor to read values from any of the graphs described above. When the mouse is moved over the plot area of any graph, it changes to a cross hair and causes the Axis Value box to display the coordinate values of the corresponding graph at the point indicated by the cursor.

By drawing the *Target Function*, varying several numeric control parameters, and selecting different types of algorithms and genetic operators, the novice user can practise and have a good idea on how quickly the algorithm is able to find the peak value, or indeed if it succeeds at all.

6.2. General-purpose programming systems

General-purpose systems are the ultimate in flexible GA programming systems. Not only do they allow the user to develop their own GA applications and algorithms, but also provide users with the opportunity to customise the system to suit their own purposes.

Overview

These programming systems provide a comprehensive tool kit, including:

- a sophisticated graphic interface;
- a parameterised algorithm library;
- a high level language for programming GAs; and
- an open architecture.

Access to the system components is via a menu-driven graphic interface, and a graphic display/monitor. The algorithm library is normally "open", allowing the user to modify or enhance any module. A high level language — often object-oriented — may be provided which supports the programming of GA applications, algorithms and operators through specialised data structures and functions. Lastly, due to the growing importance of parallel GAs, systems provide translators to parallel machines and distributed systems, such as networks of workstations.

Survey

The number of general-purpose systems is increasing, stimulated by growing interest in the application of GAs in many domains. Examples of systems in this category include Splicer, which presents interchangeable libraries for developing applications, *Micro*GA that is an easy to use object oriented environment for PCs and Macintoshes, and parallel environments like EnGENEer, GAME and PeGAsusS.

EnGENEer — Logica Cambridge Ltd. developed EnGENEer [13] as an in-house Genetic Algorithm environment to assist the development of GA applications on a wide range of

domains. The software was written in "C" and runs on Unix systems as part of a consultancy and systems package. It supports both interactive (X-Windows) and batch (command-line) modes of operation. Also a certain degree of parallelism is supported for the execution of application dependent evaluation functions.

EnGENEer provides a number of flexible mechanisms which allow the developer to rapidly bring the power of GAs to bear on new problem domains. Starting with the Genetic Description Language, the developer can describe, at 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 non-evolvable parsing symbols, which can be used for decoding complex genetic material.

A descriptive high level language, the Evolutionary Model Language, is also available to the user. It allows the description of the GA type used in terms of configurable options including: population size, population structure and source, selection method, crossover type and probability, mutation type and prob., inversion, dispersal method, and number of offspring per generation.

Both the Genetic Description Language and the Evolutionary Model Language are fully supported within the interactive interface (including on-line help system) and can be defined either "on the fly" or loaded from audit files, which are automatically created during a GA run.

Monitoring of GA progress is provided via both graphical tools and automatic storage of results (at user defined intervals). This allows the user to restart EnGENEer from any point in a run, by loading both the population at that time and the evolutionary model that was being used.

Connecting EnGENEer to different problem domains is achieved by specifying the name of the program used to evaluate the problem specific fitness function and constructing a simple parsing routine to interpret the genetic material. A library of standard interpretation routines is also provided for commonly used representation schemes such as gray-coding, permutations, etc. The fitness evaluation can then be run as either a slave process to the GA or via standard handshaking routines. Better still, it can be run on either the machine hosting the EnGENEer or on any, sequential or parallel, hardware capable of connecting to a Unix machine

GAME — **G**enetic **A**lgorithm **M**anipulation **E**nvironment — being developed as part of the main European Community (ESPRIT III) GA project, called PAPAGENA. It is an object-oriented environment for programming parallel GAs applications and algorithms, and mapping them on to parallel machines. The programming environment comprises 5 major modules:

- Virtual Machine — the machine independent low level code responsible for the management and execution of a GA application. For parallel execution the virtual machine should provide communication mechanisms for information exchange between all the virtual machines in the system.
- Genetic Algorithms Libraries — parameterised algorithms, applications and operators' libraries written in the high level language, and providing the user with a number of validated modules for constructing applications.
- Graphical Monitor (using X Windows) — the software environment for controlling the execution and monitoring of a genetic algorithm application simulation. This includes tools for configuring the graphical interface and a monitoring support from the virtual machine that can be used by application specific data monitors to visualise data and change its values.

- High Level Language (GA-HLL) — the object-oriented programming language for defining, in conjunction with the algorithm library, new genetic algorithm models and applications.
- Compilers — to various UNIX-based workstations and parallel machines.

The environment is being programmed in C++, and will be available in source code form to allow full user-modification.

***Micro*GA** — marketed by Emergent Behavior, is designed to be used on a wide range of complex problems, while at the same time being small and easy to use. The environment is also designed to be expandable. The system is a framework of C++ objects. As such, it is designed so that several pieces are used in conjunction with each other to give the user some default behaviour. Therefore, it goes far from the library concept where a set of functions (or classes) is offered to be incorporated into the user application. The framework is almost a ready-to-use application, needing 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. *Micro*GA runs on IBM-PCs compatible systems with Microsoft Windows 3.x, using Turbo/Borland C++. It also runs on Macintosh computers.

The application developer can configure his application either using Galapagos or manually. The Galapagos is a windows-based code generator that produces, from a set of custom templates and a little information provided by the user, a complete standalone *Micro*GA application. It helps on the creation of a subclass derived from its "TIndividual" class, required by the environment to create the genetic data structure to be manipulated. The number of genes for the prototype individual, as well as the range of possible values they can assume is requested by Galapagos. The evaluation function can be specified, but the notation used does not allow complex, or non-mathematical fitness functions to be entered via Galapagos. As a result, Galapagos creates a class, derived from TIndividual, which contains specific member functions according to user's requirements.

Applications requiring complex genetic data structures and fitness functions can be defined manually by inheriting from the TIndividual class, and writing the code for its member functions. After creating the application dependent genetic data structure and fitness function, *Micro*GA compiles and links everything using the Borland C++ or Turbo C++ compiler, and produces a MS-Windows executable file.

*Micro*GA is very easy to use and allows fast creation of genetic algorithms applications. However, for real applications the user has to understand basic concepts of object oriented programming and Windows interfacing.

PeGAsuS — is a Programming Environment for Parallel Genetic Algorithms developed at the German National Research Center for Computer Science. In fact, it is a tool kit which 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 MIMD parallel machines, such as transputers, and distributed systems with 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/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 of the PeGAsuS GA library. It allows the user to define application specific functions that are not provided by the system library. The script language is used to specify the experiment. The user can use it to define the application dependent data structures, attaches the genetic operators to them and specifies the input/output interface. Whereas the script language specifies the construction of a sub-population, the connections between these are specified through the graphical interface.

The Kernel includes the "base" and the "frame" functions. The "base" functions control the execution order of the genetic operators, manage communication between different processes and provide input/output 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 the frame functions. A "frame" function controls the execution of a single genetic operator, and is invoked by a base function. They 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 input/output and control procedures. It provides the user with a number of validated modules for constructing applications.

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

Splicer — This software environment was created by the Software Technology Branch of the Information Systems Directorate at NASA/Johnson Space Center, with support from the MITRE Corporation [16]. It is one of the most comprehensive environment currently available, and forms the case study below.

Case Study — **Splicer**

Splicer presents a modular architecture that includes: a Genetic Algorithm Kernel, interchangeable Representation Libraries, Fitness Modules, and user interface Libraries. It was originally developed in "C" on an Apple Macintosh and has been subsequently ported to UNIX workstations (SUN3 and 4, IBM RS/6000) using X-Windows. The Genetic Algorithm Kernel, Representation Libraries, and Fitness Modules are completely portable. The following is a brief description of the major modules:

- *Genetic Algorithm Kernel* - the GA kernel comprises all functions necessary for the manipulation of populations. It operates independently from the problem representation (encoding), fitness function and user interface. Some of its supported functions are: creation of populations and members, fitness scaling, parent selection and sampling, and generation of population statistics.
- *Representation Libraries* - interchangeable representation libraries are able to store a variety of pre-defined problem-encoding schemes and functions. This allows the GA kernel to be used for any representation scheme. There are representation libraries for binary strings and for 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 the appropriate interfaces to allow the user to create new representation libraries.

- *Fitness Modules* - these are interchangeable modules where fitness functions are defined and stored. They are the only component of the environment a user will be required to create or alter to solve a particular problem. It is possible to create a fitness (scoring) function, set the initial values for various Splicer control parameters (e.g. population size), create a function that graphically displays the best solutions as they are found, and provide descriptive information about the problem.
- *User Interface Libraries* - there are two user interface libraries: a Macintosh and an X-Window System user interface. They are event-driven interfaces and provide a graphic 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, a Fitness Module must be created using the C programming language.

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.

Having surveyed some of the available GA environments, we now speculate on the likely future developments of genetic algorithms programming environments.

7. Future Developments

The following are some trends and possible future directions in each of the GA programming environments.

Application-oriented

As with any new technology, in the early stages of development the emphasis for tools is on the ease of use. Application-oriented systems have a crucial role in bringing the technology to a growing set of application domains, since they are targeted and tailored for specific groups of users. Therefore we would expect the number and diversity of application-oriented systems to expand rapidly in the next few years. One high growth area should be the association of genetic algorithms and other optimisation algorithms in hybrid systems. By the end of the century, hybrid GA-Neural Networks will have made significant progress toward solving some currently intractable machine learning problems (promising domains include autonomous vehicle control, signal processing, and intelligent process control).

Algorithm-specific & Libraries

With the further development of application-oriented systems, coupled with the discovery of new algorithms and techniques, we expect to see an increase in algorithm specific systems possibly leading to general-purpose GAs. Access to efficient versions of these algorithms will be provided by algorithm libraries.

Educational

Interest in educational systems and demonstrators of GAs is rapidly growing. Their contribution is at the start of a new technology, but their usage traditionally diminishes as general-purpose systems mature. Thus we should expect a decline in educational systems as sophisticated general-purpose systems become available and easy to use.

General-purpose

General-purpose systems for GA programming are very recent. With the introduction of Splicer, a growing number of commercial development systems are expected to appear in the near future. We should find programming environments on an expanding range of sequential and parallel computers. An increasing number of public domain open system programming environments from universities and research centres is also expected.

Hybrid Systems

Recently there has been considerable interest in creating hybrid systems, of genetic algorithms with expert systems, and genetic algorithms with neural networks. If a particularly complex problem involves the use of both optimisation and either decision support or pattern recognition processes, then hybrid systems are powerful candidates. For example with neural networks, 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 the number of hidden units, layers and learning rates.

To conclude, genetic algorithms are robust search and adaptive algorithms that may be immediately tailored to real problems. The explosion of interest in GA applications is driving the development of GA programming environments, and many powerful commercial environments can be expected in the near future. The two major influences on future environments will be, we believe, firstly exploitation of parallel GAs, and secondly the

programming of Hybrid applications linking GAs with Neural Networks, expert systems and traditional utilities such as spreadsheets and database packages.

Acknowledgements

The authors 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 paper.

References

[1] L. Davis "Handbook of Genetic Algorithms", Van Nostrand Reinhold, New York – 1991.

[2] K.A. De Jong "An Analysis of the Behavior of a class of Genetic Adaptive Systems",PhD thesis, University of Michigan – 1975.

[3] D. E. Goldberg. "Genetic Algorithms in Search, Optimization & Machine Learning", Addison-Wesley Publishing Company] – 1989.

[4] M. Gorges-Schleuter "ASPARAGOS An Asynchronous Parallel Genetic Optimisation Strategy", in H. Schaffer, editor, *3rd International Conference on Genetic Algorithms*, pp. 422-427, Morgan Kaufmann – 1989.

[5] J. J. Grefenstette "GENESIS: A System for Using Genetic Search Procedures" in *Proceedings of the 1984 Conference on Intelligent Systems and Machines*, pp. 161-165 – 1984.

[6] F. Hoffmeister & T. Bäck "Genetic Algorithms and Evolution Strategies: similarities and differences" in *Technical Report "Grüne Reihe" No. 365*, Department of Computer Science, University of Dortmund, Germany – 1990.

[7] F. Hoffmeister "The User's Guide to ESC^{A}P$_{A}$DE 1.2 A Runtime Environment for Evolution Strategies", Department of Computer Science, University of Dortmund, Germany – 1991.

[8] J. H. Holland. "Adaptation in natural and artificial systems" in Ann Arbor: The University of Michigan Press – 1975.

[9] M. Hughes "Genetic Algorithm Workbench Documentation", Cambridge Consultants Ltd. – 1989.

[10] H. Mühlenbein "Parallel genetic algorithms, population genetics and combinatorial optimization", in J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 416-421, San Mateo – 1988.

[11] H. Mühlenbein "Evolution in Time and Space – The Parallel Genetic Algorithm" in G. Rawlins, editor, *Foundations of Genetic Algorithms*, Morgan Kaufmann – 1991.

[12] I. Rechenberg " Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution", Fromman-Holzboog Verlag, Stuttgart– 1973.

[13] G. Robbins "EnGENEer – The Evolution of Solutions" in *Proceedings of the 5th Annual Seminar on Neural Networks and Genetic Algorithms* – 1992.

[14] N. N. Schraudolph & J. J. Grefenstette "A User's Guide to GA$_{UCSD}$ 1.2", Computer Science & Engineering Department, University of California, San Diego – 1991.

[15] H. P. Schwefel "Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie" in *Interdisciplinary Systems Research*, Vol. 26, Birkäuser, Basel – 1977.

[16] NASA – Johnson Space Center "Splicer – A Genetic Tool for Search and Optimization" in *Genetic Algorithm Digest*, Vol: 5, Issue: 17 – 1991.

[17] R. Tanese "Distributed Genetic Algorithms", in H. Schaffer, editor, *3rd International Conference on Genetic Algorithms*, pp. 434-440, Morgan Kaufmann – 1989.

[18] H. M. Voigt, J. Born & J. Treptow "The Evolution Machine Manual – V 2.1", Institute for Informatics and Computing Techniques, Berlin – 1991.

[19] D. Whitley & J. Kauth "*GENITOR*: a different genetic algorithm" in *Proc. of the Rocky Mountain Conference on Artificial Intelligence*, pp. 118-130, Denver, CO – 1988.

Appendix A — 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;

    initialise_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));
}
```

```
/*
 ****************************************
 *           initialise_population      *
 * Creates and initialize a population  *
 ****************************************
 */
initialise_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);
    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 entermediary 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));
    }
}
```

Appendix B — Developers Address List

C Darwin II
Attar Software
Newlands Road
Leigh, Lancashire
England
Tel: +44 94 2608844
Fax: +44 94 2601991

EM — Evolution Machine
H.M. Voigt & J. Born
Institute for Informatics and Computing Techniques
Rudower Chaussee 5
D — 1199 Berlin
Germany
Tel: +49 372 674 5958
Fax: +49
E-mail: voigt@iir-berlin.adw.dbp.de
born@iir-berlin.adw.dbp.de

ESCAPADE
Frank Hoffmeister
University of Dortmund
Department of Computer Science
Chair of Systems Analysis
P.O.Box 500500
D - 4600 Dortmund 50
Germany
Tel: +49 231 755 4678
Fax: +49 231 755 2047
E-mail: iwan@ls11.informatik.uni-dortmund.de

EnGENEer
George Robbins
Systems Intelligence Division
Logica Cambridge Ltd.
Betjeman House
104 Hills Road
Cambridge CB2 1LQ
U.K.
Tel: +44 71 6379111
Fax: +44 223 322315

EVOLVER
Axcélis Inc.
4668 Eastern Avenue North
Seattle, WA 98103
U.S.A.
Tel: +1 206 632 0885
Fax: +1 206 632 3681

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

GAGA
Jon Crowcroft
University College London
Gower Street
London WC1E 6BT
U.K.
Tel: +44 71 387 7050
Fax: +44 71 387 1398
E-mail: jon@cs.ucl.ac.uk

GAUCSD
N.N. Schraudolph
Computer Science &
Engineering Department
University of California, San Diego
La Jolla. CA 92093-0114
U.S.A.
Fax: +1 619 534 7029
E-mail: nici@cs.ucsd.edu

GAME
Jose L. Ribeiro Filho
Computer Science Department
University College London
Gower Street
London WC1E 6BT
U.K.
Tel: +44 71 387 7050
Fax: +44 71 387 1398
E-mail: j.ribeirofilho@cs.ucl.ac.uk

GENESIS
J.J. Grefenstette
The Software Partnership
P.O. Box 991
Melrose, MA 02176
U.S.A.
Tel: +1 617 662 8991
E-mail: gref@aic.nrl.navy.mil

GENITOR
Darrel Whitley
Computer Science Department
Colorado State University
Fort Collins, Colorado 80523
U.S.A.
E-mail: whitley@cs.colostate.edu

***Micro*GA**
Steve Wilson
Emergent Behavior
635 Wellsbury Way
Palo Alto, CA 94306
U.S.A.
Tel: +1 415 494-6763
E-mail: emergent@aol.com

OMEGA
David Barrow
KiQ Ltd.
Easton Hall
Great Easton
Essex CM6 2HD
U.K.
Tel: +44 371 870254

OOGA
Lawrence Davis
The Software Partnership
P.O. Box 991
Melrose, MA 02176
U.S.A.

PC-BEAGLE
Richard Forsyth
Pathway Research Ltd.
59 Cronbrook Rd.
Bristol BS6 7BS
U.K.
Tel: +44 272 428692

PeGAsuS
Dirk Schlierkamp-Voosen
Research Group for Adaptive Systems
German National Research Center for
Computer Science - GMD
P.O. Box 1316
D-5205 Sankt Augustin 1
F.R.G.
Tel: +49 2241 / 14 2466
E-mail: dirk.schlierkamp-voosen@gmd.de

Splicer
COSMIC
382 E. Broad St.
Athens, GA 30602
U.S.A.
Tel: +1 404 5423265
Fax: +1 706 542 4807
E-mail: bayer@galileo.jsc.nasa.gov

XpertRule GenAsys
Attar Software
Newlands Road
Leigh, Lancashire
U.K.
Tel: +44 942 608844
Fax: +44 942 601991

XYpe
Ed Swartz
Virtual Image, Inc.
75 Sandy Pond Road \#11
Ayer, MA 01432
U.S.A.
Tel: +1 508 772 0888
Fax: +1