Breeder user's and programmer's Manual

J. J. Merelo,
GeNeura and Swarm Teams,
Departamento de Electrónica y Tecnología de Computadores
Facultad de Ciencias
Campus Fuentenueva, s/n, 18071 Granada (Spain)
e-mail: `jmerelo@kal-el.ugr.es`

September 23, 1995

1 Introduction

Genetic algorithms are a widely used optimization technique inspired in the way natural evolution works [1]: by selecting the best from a population of possibie solutions, and changing the rest to generate new solutions, a *good enough solution* is usually found after a number of iterations.

Genetic Algorithms are widely used in artificial life to model or imitate the process of evolution. Most artificial life models include evolution of one kind of another. This library might be enough for a vast array of models, but it can be extended, and will be probably substituted by a new one with more functionality in some time.

The genetic algorithms library is composed of two main components, reflected in protocols, the `Chromosome` and the `Population`. The chromosome is used to represent a solution, with a fixed number of bytes for each parameter in the solution, and the Population is a collection of individuals, that includes also mechanisms to evaluate those individuals, and generate one population from another.

In the future, there can be several model of chromosomes and population, which can be interchangleable. Right now, a basic steady-state population with a binary chromosome are available.

2 Chromosomes and stuff

Right now, the chromosomic world is ruled by a `Chromosome` protocol, which, in theory, includes everything that can be do to a chromosome: all posible operators, ranges, and information you can extract about it. The code for this class has been adapted from the C++ code in the GAGS package [2]

There are no methods for modifying the chromosome in situ, the only thing a programmer can do is to create chromosomes by modifying existing ones, which is the way they work in most genetic algorithms.

Besides the normal unary operators found in the literature, like mutation, some other not-so-normal operators have been included. *Swap* transposes a group of bytes from one place in the chromosome to another. *Duplicate* and *kill* increase o decrease the length of the chromosome, so that it need not be fixed in advance. Some problems require chromosomes of different length to be evaluated at the same time, like for instance, variable architecture neural networks, iterated prisoner dilemma with variable depth memory, or finite state machines with a different number of states.

The only implementation available of `Chromosome` is `BinChromosome`, a binary chromosome in which solutions are represented by strings of 0 and 1. These strings are packed in bytes, instead of representing each 0/1 as a separate character. Some of the functions have not been implemented, but they will be. I promise.

[1] Where did I hear this before?

[2] Available at ENCORE sites, at `ftp://kal-el.ugr.es/GAGS`, and `http://kal-el.ugr.es/gags.html`

Other possible implementations would be floating point chromosomes and non binary alphabet chromosomes.

3 Population and crowds

Population is basically a container for everything related to population (you guessed it already, didn't you?). Well, as opposed to Chromosome, that contains function that map chromosomes into chromosomes, Population contains functions that map one population onto another. Different population models should be used for different generational replacement strategies. For instance, Roulete Wheel selection would be better off with a couple of collections for population, one for "current" and other for the "gene pool". That will be implemented eventually.

Population also contains code to evaluate all chromosomes, and storage for the results of this evaluation. A function pointer must be handed to it, in order to evaluate chromosomes; obviously, the evaluation function must be written by the programmer. This is done to basically differentiate user's code from library's code. Probably this mechanism will be substituted or complemented by *functors*[3] in some time.

Right now, there is only one implementation of this protocol, `SSPop` or steady state population. In this population model, all individuals in the population are ranked according to fitness, and a certain percentaje of them are eliminated. These are substituted by individuals resulting from the application of the unary and binary genetic operators available, with a certain rate, to the population. The percentage of population that is eliminated is equal to the sum of the rates of all operators. These rates can be altered during the execution of the GA, using `set` messages.

Fitness is always a floating point number, and is always maximized. If you want to minimize it, just make the evaluation function return the inverse or the negative of the value.

4 How to work with it

Breeder works more or less the same way that any other GA program: generate the population, evaluate all the chromosomes in the population, and create a new one according to the rules set.

The following steps should be followed to create a program that uses Breeder:

- Write the evaluation function. This function can be any size, any shape, and anywhere, but it must be called as is shown in Fig. 3.

  The structure of the evaluation function is always the same, although the name can be obviously different. Inside the function, the values of the chromosome can be retrieved by sending the message `getGene`.

- Create the population structure. The procedure to follow is depicted in Fig. 4 This follows the obvious steps: assignation of the number of individuals, length of the chromosome, setting of the rates for all the operators, passing the evaluation pointer to the population, and so on. It also follow the standard swarm object creation procedure. At the same time, this generates a random initial population.

- Evaluation loop: basically consists of two messages: `evaluate` and `newGeneration` There could be more stuff in this loop, like representation of the best individual, computing best and average fitness, changing of rates, and so on.

5 Running the demo

There is a simple demo, that tries to optimize the number of ones in a 32-bit string. It can be compiled by issuing the command `make demos`, and run by typing `kkga` and allows the user to change stuff like the mutation rate, the number of individuals in the population and so on. It's usually so fast that it will be in the screen for barely some seconds.

[3] I don't really know what they are, but it sounds cool, doesn't it?

This demonstration uses several Swarm capabilities, like graphing and raster windows. The elongated window represents the best string in each generation.

Acknowledgments

This work has been partially supported by grant PB92-0456 from the DGICYT (Spain).

```
#ifndef __CHROMOSOME_P
#define __CHROMOSOME_P

#import <collections.h>

@protocol Chromosome

//@create
-setLength: (unsigned) len;
-setRandomInit;

-setRange: (double *) range;
-setRange: (double *) range Locus: (unsigned) l; // Range for only one gene

-setBinaryRep;
-setCharStringRep: (char *) chars;
-setFloatArrayRep;
-setPermutationRep: (unsigned) cardinality;
//@final

// unary operators - probably changed afterwards into "create" operators
- Clone : (id<Chromosome>) _chrom withMutation: (float) _mutProb;

- Duplicate : (id<Chromosome>) _chrom
        From: (unsigned) _l1
          To: (unsigned) _l2
withMutation: (float) _mutProb;

- Kill : (id<Chromosome>) _chrom
    From: (unsigned) _l1
      To: (unsigned) _l2;

- Swap: (id<Chromosome>) _chrom
       : (unsigned) _l1
   with: (unsigned) _l2
   size: (unsigned) _bytes; // Swap whole byte/loci

- RemoveReinsert : (id<Chromosome>) _chrom
                       : (unsigned) _l1
                     To: (unsigned) _l2
                   size: (unsigned) _bytes;

// binary operators
-Mate: (id<Chromosome>) _chrom1
   And: (id<Chromosome>) _chrom2
withMutation: (float) _mutationRate ; // 2-point crossover

- crossOver: (id<Chromosome>) chrom1
            : (id<Chromosome>) chrom2
      withArg: xOverFunc;

// orgy operators
- orgy: (id<Collection>) parents withArg: orgyFunc;

// info functions
- (unsigned) getLength;
- getGenotype;
- (unsigned char) getGene: (unsigned) offset;
- hexaPrint; // Print binary chrom in hexa

// Killing functions
- free;
@end

#endif
```

Figure 1: `Chromosome` protocol. Messages commented out have not been implemented.

```
#ifndef _POPULATION_P
#define _POPULATION_P

#import "Chromosome.h" // Chromosome protocol

@protocol Population

//@create
-setSize: (unsigned) _size genLength: (unsigned) _len ;
-setMut: (float ) _mut;

-setSteadyState;

//-setRouleteWheel;
//-setTournamentSelection: (unsigned) _tournSize;
-setEvaluationFunc: (float(*)( id<Chromosome> )) evalFunc;
//@end

// Genetic operators
-setMatingProb: (float) _matingProb;
-setCloneProb: (float) _cloneProb;
-setKillGeneProb: (float) _killGeneProb;
-setDupGeneProb: (float) _dupGeneProb;
-setSwapProb: (float) _swapProb;
-setRemReiProb: (float) _remReiProb;

-evaluate; // Evaluates all member of the population
-newGeneration;
- (id<Chromosome>) getBest; // Returns the best genome
-(float) getBestFitness; // Fitness of the best genome
@end

#endif
```

Figure 2: `Population` protocol

```
float evalFunc( id<Chromosome> a ) {
    unsigned i;
    float ones = 0;
    unsigned long gen, mask = 1;

    gen = ((long)[a getGene:0]<<24) + ((long)[a getGene:1]<<16)
            + ((long)[ a getGene:2 ]<<8) + [ a getGene:3 ];
    for ( i = 0; i < 8*GENLEN; i ++ ) {
        ones += ( ( mask & gen ) != 0 );
        mask <<= 1;
    }

    return ones;
}
```

Figure 3: Evaluation function that counts the number of ones in a 32-bit number.

```
gaPop = [ SSPopulation createBegin: aZone];
[gaPop setSize: popSize genLength: GENLEN];
[gaPop setMut: mutRate];
[gaPop setMatingProb: killProb];
[gaPop setEvaluationFunc: evalFunc];
gaPop = [gaPop createEnd];
```

Figure 4: Creation of the population.