libGAGS 0.91 Programmer´s Manual

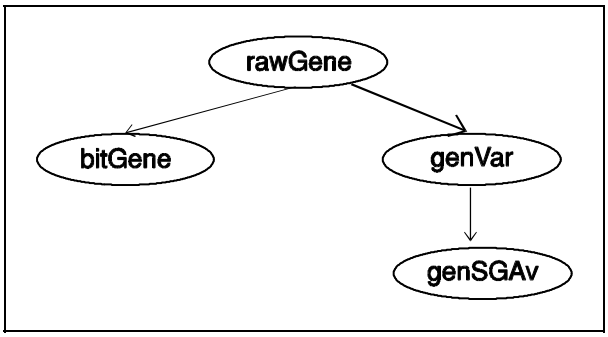
1. Philosophy

The philosophy behind libGAGS is to match as much as possible the way genetic algorithms work. Bearing this in mind, there is not much of a choice. Genetic algorithms act on population of chromosomes; besides, they need to interact with the user, showing results, and getting input from files or from the user. Thus, libGAGS is divided in the *chromosome* hierarchy, the *population* hierarchy, and some auxiliary classes for system and user interaction.

The only object-oriented GA implementation the author has heard of is, besides a commercial product, OOGA, which uses an object oriented version of LISP. Making an object oriented implementation of genetic operator in C++ seemed the wise thing to do, drawing some experience from all the programming the author did for his PhD Thesis.

2. *Chromosome* hierarchy

This philosophy applies clearly to the *chromosome* hierarchy, which is shown in Ilustrac. 1. The usual operations applied during a GA to a chromosome are only applied at birth; after that, you can only view the results. In this way, chromosomes cannot be altered after birth, and changes can only occur when new chromosomes areo created.



Ilustrac. 1 Chromosome hierarchy, with class names.

Chromosomes are composed of *genes*. For the present class, a gene is a a bit string, being all genes in the chromosome of the same length. Genes are decoded, usually when the chromosome fitness is evaluated, giving a number with a certain type and a certain range.

Chromosomes are usually used through the lowest members of the hierarchy, i.e., `bitGene` and `rawGene`. Besides constructors, that are quite numerous and complex, since all genetic operator are called from them, all publicly accesible member of the hierarchy have got:

*`unsigned getLength( void )`: which gives the number of genes in the chromosome.

*`<type> operator [] (unsigned _idx)`: which gives the _idx-th gene in the chromosome. A typical loop to access all member of the chromosome would be

```
bitGene myGen();
for ( unsigned i = 0; i < myGen.getLength(); i ++ )
      foo( myGen[i] );
```

would apply to all genes of the chromosome `myGen` the operation `foo`.

*Standard conversion is also defined for each chromosome, in such a way that, if the type of each genome is `returnType`, there is a standard conversion from the chromosome to an array of the same length than the chromosome. For instance

```
bitGene myGen( );
unsigned len = myGen.getLength();
unsigned * genArr = myGen;
```

This conversion is seldom used explicitly, and is more usually found in function calls, where temporaries are created.

*A friend function is also defined, in such a way that the chromosomes can be printed to standard output. It will output all genes in the chromosome separated by whitespace.

```
cout << myGen;
```

would be perfectly legal independently of the place `myGen` is situated inside the *chromosome* hierarchy.

Some members have also particular functions, that account for some individual feature; these functions, nevertheless, have been reduced to a minimum.

Almost all the complexity of the library is inside the class constructors. A chromosome can be born in many ways: randomly, in the beginning of all times (and the GA run), from two parents, and from a single parent. In any case, some genetic operators are applied. Available genetic operators are:

*2-point crossover.

*Uniform crossover.

*Bit-flip mutation.

*Transposition.

*Variable-length operators.

Almost all of them can be applied, although the most usual ways are using 2-point crossover or uniform crossover with mutation with two parents, and mutation with only 1 parent.

`bitGene` objects are chromosomes composed of genes of fixed size in bits (*n*). Each gene returns an unsigned integer, and, as such, has a range between 0 and 2^n-1. It can be used to represent parameters that can have a narrow range of values. Fixed bit length may be a problem, but allow some operations, like transposition and variable-length operators, to be done easily.

`genSGAv` is a template class used to represent a chromosome, whose genes have variable range, useful for implementing Goldberg´s SGA (simple genetic algorithm). The types used to instantiate the class must be `double` or `float`. It has not been tested on other types, but it will probably work if there is a standard conversion from `char *`. Each gene is a double or float with the range that has been defined at birth. Ranges are inherited.