10.1 Introduction

GAGS is a C++ class library designed to make programming a Genetic Algorithm (GA) [Heitkötter and Beasley, 1996] easy, and at the same time, flexible enough to add new objects, which are treated in the same way as native ones. As many other class libraries, GAGS includes the following features:

- **Chromosomes**, which are the basic building blocks of a genetic algorithms. Chromosomes are bit strings, and have a variable length.

- **Genetic operators**, which are not part of the chromosome class, but are outside it (actually, they are halfway outside: being friends, they conceptually belong to the same class); this way, operators are not reduced to mutation and crossover: there are many predefined genetic operators, like bitflip mutation, crossover (uniform, 1- and n-point, and gene-boundary respecting), creep (change gene value by 1), transpose (permute genes), and kill, remove, and randomAdd, which alter the length of the chromosome. New operators can be added and used in the same way as the predefined. Operators can be applied blindly, or to a certain part of a chromosome.

- **Views**, which represent the chromosome and are used to evaluate and print it; views are objects used to watch the chromosome as a float or int array or whatever is needed to
evaluate it.

- **Population**, which includes a list of chromosomes to be evaluated and sorts them in another list when they are; operators for generational replacement, and a list of genetic operators together with their rates; the number of chromosomes, the list of operators and their rates can be changed in runtime.

- The evaluation or fitness function can be any C or C++ function, and it can return any class or type; thus, fitness is not reduced to a floating point number: it can be an array or any other user-defined type; the only requirement is that it has ordering operators defined.

The second and last feature set it apart from other GA class libraries, like Matthew's GAlib [Wall, 1995] and TOLKIEN [Tang, 1994]. At the same time, it lacks some features like a user interface and provision for more than one kind of chromosomes.

In this chapter we will see the way a genetic algorithm is programmed using these building blocks. We will take a bottom-up approach, starting with a chromosome, and building up from there.

### 10.2 Creating chromosomes

Chromosomes are the basic building blocks of a genetic algorithm, and are used to represent a problem solution. You will not usually need to create them directly (that is the role of the Population class, but just in case, this is the way it would be done).

```cpp
#include <gagschrm.hpp>

main() {
    const unsigned numBits = 12;
    chrom aChromosome(numBits);
    cout << aChromosome << endl;
}
```

which would output something like

```
(kal-el) ~/txt/tex/gags/examples> ex1
00101111001
```

that is, a 12 bit string. Chromosomes are always bit strings, and
they are not much more than that. They have no structure, and no operators to work with them, other than `getLength()`, `operator[]` and `getValue`. Since chromosomes can only be changed at birth time, there are no mutators, only inspectors\(^\dagger\).

Actually, there are other ways of creating chromosomes: through copy constructors (from other chromosomes), or using a constructor from a `bitString`, which is the internal representation used for the chromosomes’ bit strings. A `bitString` is actually a set of ordered bits, whose value can be true or false, but they can be defined using `char` strings.

```cpp
text aBS: "101001111111";
chrom bChromosome ( aBS );
```

Actually, this can be done in a single sentence, since automatic conversion is applied to the character string; but the actual way it is done is as shown.

There is not much you can do with only a chromosome, so that we will go ahead to the next section, which deals with how to operate on a chromosome.

### 10.3 Genetic Operators

Genetic operators are used in genetic algorithms to generate diversity (mutation-like operators) and to combine existing solutions into others, decreasing diversity (crossover-like operators). The main difference among them is that the former operate on one chromosome, that is, they are unary, while the latter are binary operators. In GAGS, genetic operators need not be coerced into the mutation/crossover paradigm, they are just functions that map chromosomes onto chromosomes.

In GAGS, operators constitute a class, that is, they are not methods of the chromosome class as is usual. That means that they can be created, destroyed and changed in runtime, and besides, that they can be subclassed to create new operators. This class is designed to act as a functor, i.e., function-syntax objects, which have `operator()` overloaded; this is only syntactic sugar, but

\(^\dagger\) Actually they can be changed. We will later see the way it is done.
allows to approach the looks of the C++ implementation to the actual algorithm.

They act in two different ways: *non-directed*, which means that they act on a random part of the chromosome; and *directed*, that is, they act on a preselected part of the chromosome. Besides, many operators consider the chromosome is divided in chunks, usually called *genes*; the size of those chunks must be passed as a parameter to their constructors.

Some operators are variable-length operators: they alter the length of the chromosome; chromosome length is always computed in every method that needs it, which means that GAGS is prepared for variable-length chromosomes. Length is always changed by a discrete amount of genes. Note also that binary operators acting on variable length chromosomes can also change the length of the resulting chromosome.

An example that includes binary and unary operators is as follows

```cpp
#include <genop.hpp> // Chromosomes already included

main () {
const unsigned NUMGENES = 4;
const bitLength_t SIZEGENES = 3;
chrom aChromosome( NUMGENES, SIZEGENES ); // Create chromosome
cout << "aChromosome\t\t" << aChromosome << endl;
genOp creeper( SIZEGENES, genOp::CREEP ); // Apply unary genOp
creeper( &aChromosome );
cout << "aChromosome creeped\t\t" << aChromosome << endl;
genOp mutator( (mutRate_t) 0.1 ); // Another unary genOp
mutator( &aChromosome );
cout << "aChromosome mutated 0.1\t" << aChromosome << endl;
genOp SGA2pt( (unsigned char) 2 ); // Binary genOp, 2-point crossover
bitString aBS ="111111111111";
```
chrom anotherChrom( aBS ); // Define
other Chromosome from string

cout << "Crossovering with anotherChrom\n\t\t" << aChromosome << "\n\t\t" << anotherChrom << endl;

SGA2pt( &aChromosome, &anotherChrom ); // Result on the first

cout << "Result\n\t\t" << aChromosome << endl;

This program defines two unary operators: creeper, which changes a gene by plus or minus one and mutate, which does the usual thing; and a binary operator, SGA2pt, a simple two point crossover, and applies them to the chromosomes defined. A gene has been defined as a 3-bit segment, and each chromosome has got 4 genes. Output would look like this:

aChromosome 10011011110
aChromosome creeped 100010011110
aChromosome mutated 0.1 100010011010
Crossovering with anotherChrom 100010011010
111111111111
Result 100010011110

The genetic operator type is computed in two different ways: if its constructor has got unique parameters, the genetic operator type is deduced; if not, typed enums are used. There are 9 predefined operators, shown in box 1

Using the modes shown in table 10.1, genetic operators use three different constructors:

genOp( mutRate_t _rate,
    genOpMode_mode = MUT); // Mutation and uniform crossover

genOp( bitLength_t _lenSits,
    genOpMode_mode = TRANSP); // Most operators acting on genes

genOp( unsigned char _numPts ); // number of xOver points > 1

Genetic operators can also be applied in a directed way, by using the applyAt method:

    const unsigned toKill = 1;

©1999 by CRC Press LLC
chrom thisChrom( "111000111000" );

genOp killer( SIZEGENES, genOp::KILL );

killer.applyAt( toKill*SIZEGENES, &thisChrom );

cout << "Result	" << thisChrom << endl;

which would output something like

Result

111111000

genOp::DUP  duplicates a gene with mutation

genOp::KILL  eliminates a gene

genOp::RANDINC  adds a random gene

genOp::TRANSP  transposes two genes, i.e.,
permutes its contents

genOp::CLONE  copies the chromosome without
doing anything

genOp::MUT  usual bit-flip mutation

genOp::UXOVER  uniform crossover: interchanges
bits between the two chromosomes

genOp::XOVER  usual n-point crossover:
interchanges the part of the
chromosome between two or more
randomly generated points

genOp::GXOVER  same as before, but respects gene
boundaries, which means that only
whole genes are interchanged

genOp::ZAP  changes the value of a gene or part
of a chromosome to another value

genOp::CREEP  changes the value of a gene by
plus or minus one.

Table 10.1 Genetic operators and modes.
Once again, you will not be able to solve many problems with chromosomes if the only thing you can do is watch its binary face and change it at will. Usually, running a genetic algorithm involves decoding a chromosome, gene by gene, and applying some function to it. We will show how to do this in the next section.

10.4 Looking at chromosomes

The most straightforward way of doing it is using `chrom::getValue()` function, which turns the chromosome raw bits into unsigned longs; this obviously means that the gene size can never be bigger than the size of an unsigned in the particular machine, but don't worry, GAGS will warn you if you attempt to define huge genes. Thus, chromosomes can be fully decoded this way:

```cpp
cout << "Gene values: "; //Printing chromosome as unsigned
for ( unsigned i = 0; i < NUMGENES; i ++ ) {
    cout << aChromosome.getValue( i*SIZEGENES, SIZEGENES ) << " ";
}
```

But most usually, a chromosome will be watched as an array of floats, with fixed or variable range; each gene in the chromosome must be decoded to this range before evaluating it. `views` are used with this aim: decoding chromosomes to evaluate them; and they are not part of the chromosome, but separate objects, mainly so that new views can be defined, without delving into the `chrom` code.

For instance, if you want to print your `aChromosome` as if it were an array of floats in the range -1, 1 (that is: start -1, range 2), a view could be defined and used in this way

```cpp
view<float> vista( SIZEGENES, -1, 2 );
for ( unsigned i = 0; i < vista.size(&aChromosome); i ++ ) {
    cout << vista( &aChromosome, i ) << " ";
} cout << endl;
```

`views` are also functors, that is, they have function syntax, and

©1999 by CRC Press LLC
several are defined by default: fixedrange array, variable-range array, and travelling salesman (which prints letters, instead of numbers). size computes the size in genes of the chromosome that is being decoded, remember that GAGS always uses variable-length chromosomes. They are also defined in such a way that different types are created depending on the parameters you give it: fixed and variable-range array are defined using the same constructor.

The classes with which you have to instantiate views depend on the kind you have defined, but in any case they must have all arithmetic operations defined.

With views, which define a standard decoding, chromosomes can be easily evaluated and fitness computed, as explained in the next section.

10.5 Computing fitness

Once chromosomes have been created, massaged and watched in many different ways, it's about time we evaluate them. In genetic algorithms, success is usually defined by the evaluation of a function, which tells us how good the chromosome is at solving the problem it has been instructed to solve.

Following the spirit of the classes described in past sections, fitness will also be a class in most GA applications. Usually fitness is only a floating point number, but in some cases, there are several things to take into account when evaluating a chromosome: success rate and size, for instance. If fitness is an scalar number, nothing needs to be defined anew, but if we are going to use a structured fitness, some functions need to be defined: mainly comparison functions, since fitness is used to rank chromosomes. This would be an example of how to define and use fitness

```cpp
#include <gagsview.hpp>  //Chromosomes already included

struct fitness_t {
    float evaluation;  // Scalar fitness value
    unsigned len;     // Vector length

    fitness_t( float _d = 0, unsigned _l = 0 ): evaluation(_d), len(_l) {};

    int biggerThan( fitness_t_f ) const {
```
if (evaluation > _f.evaluation) {
    return 1;
} else if (evaluation == _f.evaluation) {
    return (len < _f.len) ? 1 : 0;
    return 0;
};

int operator > (fitness_t _f) const {
    return biggerThan(_f);
};

int operator < (fitness_t _f) const {
    return !biggerThan(_f);
};

int operator == (fitness_t _f) const {
    return ((evaluation == _f.evaluation) && (len == _f.len));
};

};

inline ostream& operator <<(class ostream& s, fitness_t _f) {
    return s << _f.evaluation << " & " << _f.len << endl;
}

// Fitness function
fitness_t fitness(const chrom& _chrom, const view<float>& _vista) {
    // Evaluates fitness by adding gene values
    for (unsigned i = 0, float eval = 0; i < _vista.size(&_chrom); i++) {
        eval += _vista(&_chrom, i);
    }
    return fitness_t{eval, _vista.size(&_chrom)};
}

main() {
    const unsigned NUMGENES = 4; // Usual prologue
    const bitLength_t SIZEGENES = 3;
    seed_random(time((time_t)0)); // From randcl
    chrom aChromosome(NUMGENES*SIZEGENES); // Create
Obviously, you don't always need to go to these lengths to define fitness; most of the times, a simple scalar fitness is used. This fitness will usually be returned by an evaluation function, or fitness function, will usually be a method of an object that will be constructed from a chromosome.

In any case, the previous example describes how to define a fitness class, which in this case is defined as a struct to have all members public by default; three operators: <, > and == are defined, a default constructor, and a function to print it; then, a function to evaluate fitness is defined; in this case, it adds the total value of the genes and sets it as fitness, and then the length of the chromosome, which will be the second component in the fitness.

The main program then creates a chromosome and evaluates it. A view must be created so that the fitness function can evaluate the chromosome, or else write the fitness function using the getValue method of chrom; both, chromosome and instantiated view, are passed to the fitness function.

Most of the GA components are already in place, so that we can now go ahead to creating a population, which is the class that really deals with the genetic algorithm in the next section.

### 10.6 Population

A population has got a list of chromosomes to evaluate, an usually ranked list of evaluated chromosomes, and a list of operators; since they are all lists, they can be changed at runtime. Creating a population is just a matter, then, of giving parameters to the creation of those list. The easiest way of creating a population is as follows:

```cpp
const unsigned MAXGENES = 10, MINGENES = 10;
const bitLength_t SIZEGENES = 4;
```
const unsigned NUMCHROMS = 20;
mutRate_t mutProb = 1.0/(SIZEGENES*NUMGENES);
Population<float>
    people( NUMCHROMS, SIZEGENES, MINGENES,
            MAXGENES, mutProb );

i.e., default parameters are number of chromosomes in the
population, the size of each gene, and the max and min number of
genes in each chromosome; remember that GAGS by default works
with variable-size chromosomes. The next parameter is the
mutation probability, that is, the number of bits that will be flipped
each time a chromosome is selected for mutation. Besides,
Population is a class template that must be instantiated with the
fitness type; in this case fitness is simply a floating-point number,
but it could be something more complex; in any case, Population
will be instantiated with fitness_t, which must have been defined
in advance as the type returned by the fitness evaluation function.

At the same time, since Population is a container class, it must
have iterators to run on it. Instead of defining an iterator class by
default for each Population object, as it is done in the standard
template library, it must be created by hand (but this might changed
in the future). In any case, there are three iterator classes defined
for the population, corresponding to the three lists defined before.

All in all, here is an example of how to use the Population class

#include <gagspop.hpp> // For Population class
and iterators
#include <gagsview.hpp> // For views
main () {
    const unsigned NUMGENES = 10;
    const bitLength_t SIZEGENES = 4;
    const unsigned GENERATIONS = 100;
    const unsigned numChroms = 20;
    seed_random( time( (time_t) 0 ) ); // From randcl
    // Set up population
    Population<float>
        people( numChroms, SIZEGENES, NUMGENES, NUMGENES,
                1.0/(SIZEGENES*NUMGENES) );
    people.setElite( 0.6 ); // Sets Elite rate
genOp adder( SIZEGENES, genOp::DUP); // Creates gene duplication operator...

people.addOp( &adder, 0.1 ); // ...and adds it to the population

// Starts GA
popIter<float> censo( people ); // Creates iterator
view<float> vista( SIZEGENES, -1.0, 2.0 ); // View with range -1.0, 1.0

for ( unsigned i = 0; i < GENERATIONS; i ++ ) {
    censo.reset( Iter::FIRST); // Resets iterator
    while( censo ) { // Evaluates fitness
        float val = 0.0;
        chrom. trap = censo.current(); // Chromosome to evaluate
        for ( unsigned j = 0; j < tmp->getLength()/SIZEGENES; j ++ ) {
            val += vista( tmp, j);
        }
        censo.setFitness( val ); // This erases it from censo
        censo++; // Ahead to the next chromosome
    }
}

evalIter<float> eval( people ); // Iterator on evaluated chromosomes
// mapItem contains a pointer to the chromosome, key the // fitness value
for ( unsigned j = 0; j < eval.current().mapItem->getLength()/SIZEGENES; j ++ ) {
    cout << vista(eval.current().mapItem, j) << " ";
}
cout << " = " << eval.current().key << endl;
if ( i < GENERATIONS - i)
    people.newGeneration(); // Performs selection and genetic op application
At the same time, it is an example of how to run a complete genetic algorithm. The Population class, which could actually be a Genetic Algorithm class, is defined in three steps,

- Creation of a population object, giving it information about the number of chromosomes, its size, and parameters for the mutation operator. The population has got a mutation and a 2-point crossover operator in its list by default, with equal priority (see below).

- Setting of the elite parameter, the number of chromosomes that will remain in each generation. Only steady-state selection is supported: (l-elite)*population will be eliminated, and substituted by the offspring of the (l-elite)*population best. Operators are applied according to priority, which means that, depending on the operator rate, a roulette wheel is created, with slices corresponding to operator probability, and each time a chromosome is selected for reproduction, a random number is drawn and a genetic operator selected.

- Creating new genetic operators, which at the same time are added to the population list. In this step, operator probabilities can be changed (in fact, they can be changed at any time during the genetic algorithm) using the setRate function. Rates are actually more like priorities, since all rates are added and normalized to one before applying them in each generation. That is, if you have rates of 0.5, 1 and 1.5, each operator will be applied to 18, 33 and 72% of the population that is being created each generation, approximately.

The genetic algorithm part of the program is just a for loop over the number of generations that has been selected in advance, with an inner loop to evaluate each member of the population in turn. Note that the loop uses the non-evaluated population iterator, which means that each chromosome is only evaluated once; using the setFitness function moves the chromosome from the non-evaluated list to the evaluated list. Finally, the best chromosome,
that is, the one which is bigger than any other, is printed using the evaluated-population iterator.

This is more or less how far you can go by using GAGS in your own programs, that is, you need to write a fitness function, then select the population parameters, and then the genetic operators and their rates. However, as any good-behaviored class library, you should be able to extend GAGS to adapt it to your own purposes. Next section explains how to do it.

10.7 Extending GAGS

Both genetic operators and view use a letter/envelope structure (see [Coplien, 1994]); different "letter" and thus, different implementations are used for different views and genetic operators, but the client-programmer need not worry about it, since all letters can be accessed using envelopes, that is, genOp and view. This is done mainly because views and genOps are friends of chromosomes, and since friendship is not inherited, only view base objects can access tim chromosome internal representation (i.e. tile bitstring). This causes a problem with templates, since in gnu's g++, there is no such a thing as friendship-to-template. Currently, only templates instantiated with float and char can be used; to use any other, it must be added to the friendship list at the beginning of chrom.

This letter/envelope structure, at the same time, makes it easier to program new objects, since the interface is simplified, and most, if not all, protected functions needed are already defined.

10.7.1 Extending views

If you want to use views properly, the same interface should be kept, which means that the envelope class will stay more or less the same, except for new constructors, and only a new "letter" will be created. To do so, several steps have to be taken. We will build a Travelling Salesman view, in which there are log₂n size genes, for n cities; and they are represented as letters, a being the first city, b the second, and so on.

1. Add a new constructor to the view class. This constructors should have different calling structure than the others; if it
does not, an enum will have to be created to differentiate it from the constructors for other objects with the same calling convention. The declaration will be included in gagsview.hpp, and the definition in gagsview.icc, which includes all the template definitions. Once the name of the view is decided, the constructor includes a forward to the constructor of the derived class, and a check.

```cpp
template<class Type>
view<Type>::view( unsigned _numCities ) {
    inview = new TSPView( _numCities );
    check( inview );
}
```

In this case, TSPView is not itself a template, since it will always return characters; usually, Type will be used for the construction of the letter object.

2. Declaration and definition of TSPView itself. The public virtual functions operator() and size have to be redefined; the first one will return the ith gene, and the other the total number of genes. Take into account that only methods in the base class can access private parts of the chromosome, so that usually these functions will forward request to the base class. In this case, it accesses the getValue protected method, that in turn accesses the internal representation of the chromosome. Since the base and envelope class has only a pointer to the letter class, its default constructor is called. Check GAGS sources, to which TSPview has already been incorporated, to see the way it is implemented: it basically involves converting strings to a suitable representation.

3. Testing the new view. Insert an object of the new class in the program pgview.cc, which tests view objects, compile and run it.

```cpp
//pgview.cc
#include "gagsview.hpp"
main () {
    const unsigned NUMGENES = 5;
    const bitLength_t SIZEGENES = 8;
```
chrom aChrom( NUMGENES.SIZEGENES );
view<unsigned long> vista( SIZEGENES, -1, 2 );
for ( unsigned i = 0; i < vista.size(&aChrom); i ++ ) {
    cout << vista( &aChrom, i ) << " ";
}
cout << endl;

const unsigned numCities = 8;
const unsigned numBits = 3;
chrom anotherChrom( numCities • numBits );
view<char> TSPvista( numCities );
for ( unsigned i = 0; i <
    TSPvista.size(&anotherChrom); i ++ ) { cout <<
    TSPvista( &anotherChrom, i ) << " _";
}
cout << endl;

10.7.2 Extending genetic operators

Extending genetic operators involves the same operations than views: it is a matter of creating new letter classes, and changing some stuff in the envelope class. We will create a new operator, shift, which adds or substract one from one gene, substracting or adding one to the next at the same time.

- The operator name, for instance, SHIFT should be added to the enum in the genOp class header, that is, line 30 of the genop.hpp file.
- A new genOp constructor for the new class will be needed. If so, add it to the genOp header and declare it; if parameters are more or less the same as already defined operators, use one of the existing constructors. Not in this case, since shift only needs the gene size, which use the second constructor. This means that this constructor must be modified anyway. Go to the genop.cc file, around line 77, and add something like this:

  case SHIFT:
  
  op = new shiftOp( _lenBits );

©1999 by CRC Press LLC
arity = FALSE;
break;

which creates the letter object for the new operator class, and assigns it FALSE arity, that is, arity one. Binary operators would have TRUE arity.

• Go to the genopimp.hpp header, to create the class interface, and copy whatever other class interface suits you the best. This means that you already have operator () with arity one and two defined, constructor, but not copy constructor. It does not make much sense to copy genetic operators. In this case, you can directly copy creepOp declaration and change names.

• Write an implementation of the genetic operator, and put in genopimp.cc or your own file. In this implementation, try to use the functions in genop.cc: remember that the letters cannot access chroms directly, only the envelope. Take a look at the implementation of the other genetic operators, so that you have an idea of how things are done.

10.8 Solving real problems with GAGS

Some applications can be solved in an straightforward way using GAGS, but some others need some programming, although much of the problems come from the fitness function, not the genetic algorithm itself. Here is a showcase of some problems to which GAGS has been applied successfully:

10.8.1 Optimizing Neural Networks

Designing a good neural network (NN) for pattern recognition has always been a problem in the neural net community [Ripley, 1994]. Most of the times, NN parameters and policies (like initialization, or selection of the training samples) are chosen resorting to past experience. A good algorithm that automatically designs at least a kind of neural networks would be desirable. That is why we applied genetic algorithms to Kohonen’s LVQ, Learning Vector Quantization, a supervised codebook training algorithm, yielding G-LVQ, an optimized version of the same algorithm [Merelo and Prieto, 1995, Merelo et al., 1997a], which creates a population of variable-size neural networks, trains them, and selects them.
This application is the kind of problems GAGS is designed for: complex-fitness, variable-length chromosome problems. However, the main program is quite straightforward. Some sections have been suppressed and substituted by a short description

```c
main( int argc, char **argv ) {
    //Variable declaration...
    //Command line processing...
    //NN Training and test file loading...
    //Population setup
    Population<fitness_t> myPop( popSize, SIZEGENES, numGenes / 2, rangeMax, mutRate );
    myPop.setElite( elitePerc );   //Never forget to do this
    popIter<fitness_t> censo( myPop );

    //Genetic operator setup
    genOp adder( SIZEGENES, genOp::DUP);
    genOp killer( SIZEGENES, genOp::KILL);
    if ( ! directed ) {      // Non-directed genop application
        myPop.addOp( &adder, addRate*100 );  // Command line rates have 0-1 range
        myPop.addOp( &killer, killRate*100 );
    }

    //Genetic algorithm with directed operator application
    for ( unsigned g = 0; g < GENERATIONS; g++ ) {
        censo.reset( Iter::FIRST);               // Evaluate now,
        while( censo ) {                          // Compute fitness and correct
            unsigned minWinner, maxWinner;
            chrom* tmp = censo.current();
            censo.setFitness(fitness( tmp,
```
trSample, tstSample, dim, numClasses, startR, endR, epochs, minWinner, maxWinner, NULL));

if ( ( g < GENERATIONS - 1 ) && directed) {  
  // Before last generation
  if ((myrand(1000)/1000.0) < killRate ) {  
    if ( (SIZEGENES < tmp->getLength())  
      && ( minWinner*SIZEGENES < tmp->getLength() ) ) {  
      // If not too small
      killer.applyAt( minWinner*SIZEGENES, tmp );  
    }  
  }  

  if ((myrand(1000)/1000.0) < addRate ) {  
    if ( maxWinner*SIZEGENES < tmp->getLength() )  
      adder.applyAt( maxWinner*SIZEGENES, tmp );  
  }  

  censo++;  
}

//Print out best  

if ( ( g < GENERATIONS - 1) myPop.newGeneration();  
}

As it can be seen, population is set up in the usual way, except  
that the chromosome size range is effectively used, giving the max  
and min number of genes for each chromosome. Then, besides  
the two default operators, crossover and mutation, the operators that  
eliminate and add randomly or not genes to the chromosome are  
added to the list. The GA loop is also the usual, except that for each  
chromosome, the gene that gets most and least hits (maxWinner,  
minWinner) is stored, and used with some probability to be  
duplicated (using adder) or eliminated (using killer). Fitness  
takes into account three things: first, accuracy achieved by the  
neural network in classification of the test file; then neural net  
length, and then distortion, which represents the average distance.
from the test file to the codebook.

This application, at the same time, combines genetic algorithms and neural networks, that is, global search procedures and local search procedures, in a meaningful way; instead of making the GA set the NN weights, it only sets the NN initial weights, which makes search, to a certain point, faster and more precise.

10.8.2 Optimizing ad placement

The problem can be defined in this way: given $M$ media, which can be printed, broadcasted, or other, place $N$ ads in such a way that the audience is maximized, taking into account several constraints, like a maximum or approximate bound for money spent, and a maximum audience reached. Different media have got different ratings, or audience, and obviously, different prices for an advertising unit, or module. This makes advertising placement a combinatorial optimization problem. Besides, the objective is not only to reach the possible consumer, but to reach him or her a certain number of times (called impacts), so that he or she will afterwards remember the ad, and modify his or her behavior accordingly.

This application [Merelo et al., 1997b] is even more straightforward than before. In this case, all elements intervening in the fitness was combined in a formula, so no vectorial fitness was used, default operator rates had to be adjusted, and creepOp was added to the mix, so that number of ad placements changed smoothly.

```c
// include files ...
main( int argc, char** argv ) {
  unsigned
    popSize = 400,       // population size
    generations = 100,   // number of generations
    sizeGenes = 4;       // size in bits of each locus/gene
    mutRate_t mutRate = 0.1; // mutation rate; xOver is uniform with prob 0.01
  // Command line checking ...
```

©1999 by CRC Press LLC
// Creation of media objects ...

// Population declaration and setup
Population<float>
    myPop( popSize, sizeGenes, chromSize, chromSize, 
        mutRate );
myPop.setElite( 0.6 ); // Never forget to do

// Add new operators
genOp creeper( sizeGenes, genOp::CREEP);
myPop.addOp( &creeper, 0.1 );  // A new

// Change rates using the genOp iterator
popIter<float> censo( myPop );
opIter<float> oi( myPop );
while (oi) {
    switch ( oi.current().mapItem->getMode() ) {
    case genOp::MUT:
        oi.setRate( 0.12 );
        break;
    case genOp::XOVER:
        oi.setRate( 0.1 );
        break;
    default:
        break;
    }
    oi++;
}

// Genetic algorithm loop ...........
for ( unsigned g= 0; g < generations ; g++ ) {
    censo.reset( Iter::FIRST); // Evaluate fitness
    while( censo ) { // Compute fitness and correct
        chrom* tmp = censo.current();
}
censo.setFitness( unMedio->fitness( tmp, sizeGenes ));
censo++;
}

// Print best ...
if ( g < generations - 1)
    myPop.newGeneration();
}

10.8.3 Playing Mastermind

Solving the game of MasterMind using GAs is quite a difficult problem [Bernier et al., 1996], since there is only one correct solution. Along the game, there are several partial solutions, and the GA will strive to find them. This problem required a lot of tweaking of the GA, mainly to keep diversity, but also to overcome the problem of having discrete fitness, and to keep the number of generated solutions to a minimum, since the success of a MasterMind solving program lays not only in the number of guesses made before the final solution, but also on the number of combinations generated to find it. Using GAGS, the following design decisions were taken:

- Population was huge, in order to keep diversity, around 500 individuals, and it increased with the length of the combination, that is, the size of the space to search.

- Besides usual operators, transposition was also used; since it permutes the values of two gene positions, it was quite adequate for combinatorial optimization problems.

- Some operators were not adequate for some phases of the search: for instance, it did not make much sense to use mutation when the combination was correct except for pin position, that is why the application rate of all the operators changed with the number of correct positions and colors, to become zero except for transposition when all the colors where correct.

This program has been working online for a long time at the
10.9 Conclusion and future work

GAGS is a C++ class library which can be easily used for solving many problems using Genetic Algorithms, and at the same time can be easily added new operators, or new interpretations of the chromosome. So far, it has been used in several applications, allowing the rapid development of new applications, usually in less than one week for an expert C++ programmer.

This does not mean that it lacks more things. Some of these features might be added in the future, by order of importance

- **STL compliance** STL has been recently adopted as the standard C++ library. GAGS could use many of its data structures, like lists, maps, vectors and so on. STL involves some changes in mentality, and obviously in interface.
- Adding new selection strategies as functors, and taking selection strategies out of the Population class. Conceptually, selection and reproduction operators should be outside the Population class, and besides, this would allow changing population operators in the same way that chromosome-level operators can be changed now.
- Adding a user interface.

References


