INTRODUCTION

TOLKIEN (**TO**o**LKI**t for g**EN**etics-based applications) is a C++ class library named in memory of J. R. R. Tolkien. The target users are those who involved in researches in Genetic Algorithms (GA) and classifier systems with working knowledge in C++. TOLKIEN is developed as a prototyping tool that enables genetics-based applications to be constructed easily.

TOLKIEN contains a lot of useful extensions to the generic genetic algorithm and classifier system architecture. Examples include :

- chromosomes of arbitrary types
- Gray code encoding and decoding;
- multi-point and uniform crossover;
- diploidy and dominance;
- various selection schemes such as tournament selection and linear ranking;
- linear fitness scaling and sigma truncation;

WHAT'S NEW IN THIS RELEASE ?

This version of TOLKIEN makes extensive use of the Standard Template Library (STL), which will become part of the standard library of the C++ language.

☞ This alpha release does not contain the classifier system classes and the code has only been tested using g++ 2.7.2 a Linux machine.

THE TOLKIEN GENETIC ALGORITHM

This section describes the characteristics of the genetic algorithm implemented by TOLKIEN. Note that this section is not a tutorial on genetic algorithms.

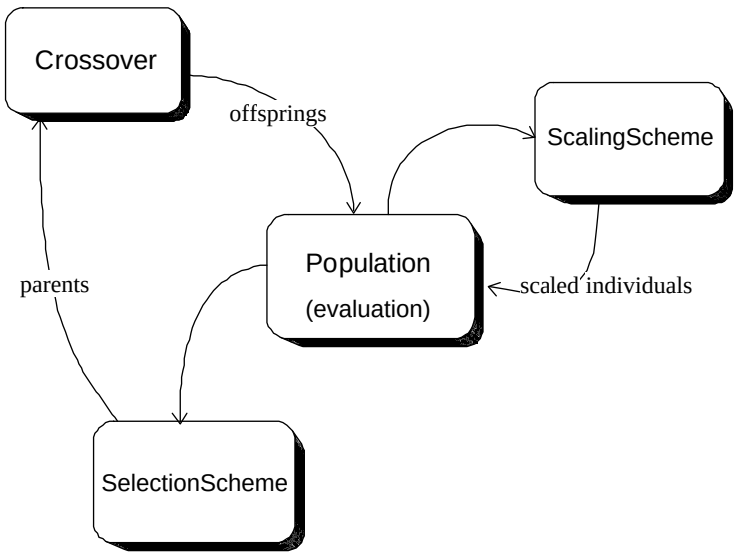

Figure 1 The GA cycle.

Class **GeneticAlgorithm** contains the execution cycle of the genetic algorithm (Figure 1) :

1. Create a temporary collection called *newInds* to store the new offsprings.

2. While more offsprings are required

   select two parents from the current population (class **SelectionScheme**).

   create two offspring by crossing over the parents (class **Crossover**).

   perform mutation on the offsprings

   add the offsprings to *newInds*

   end-while

3. Replace individuals in the current population by individuals in *newInds* (class **Population**).

4. Evaluate the raw fitness (objective value) of new offsprings, if required the fitness of the individuals not replaced are also evaluated.

5. Perform fitness scaling if necessary (class **ScalingScheme**).

Individuals

Class **TGAObj** is defined for objects that can be operated on by a genetic algorithm. Derived classes must define the clone (deep copy) and mutation functions :

```
class TGAObj {
public:
            .
            .
    virtual TGAObj* clone() const = 0; // deep copy
    virtual void mutate(double mutation_rate) = 0;

private :
    double      flFitness;
    double      flObjValue;
};
```

Two derived classes of **TGAObj** provided by TOLKIEN are **BinHaploid** and **BinDiploid**, the binary data structures used by traditional genetic algorithms. The class diagram is shown in Figure 2. Chromosomes of arbitrary genotype is created by instantiatng the template class **Haploid**.

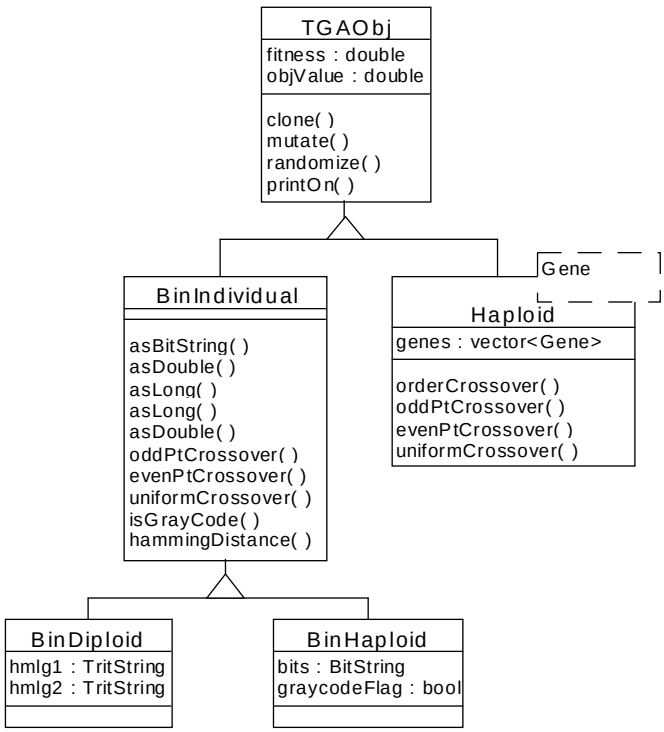

Figure 2 The TGAObj class hierarchy.

Population

Population classes differ from each other by the way new individuals replace old ones. When the *generation gap* is one (i.e. the whole population is replaced in each generation) all population classes behave in the same manner.

The population classes provided by TOLKIEN are :

SimplePopulation - individuals are replaced in a random fashion.

ElitePopulation - the worst individuals in the population are replaced.

CrowdingPopulation - use De Jong's crowding scheme to replace individuals.

In this version class the default collection class to storage individuals is the STL `vector` class. Each of the classes **Population**, **SelectionScheme**, **ScalingScheme** and **GeneticAlgorithm** is a template class with a default argument of vector<TGAObj*>. Any random accessible collection class can be used instead of class `vector`. The class hierarchy of population classes is shown in Figure 3.

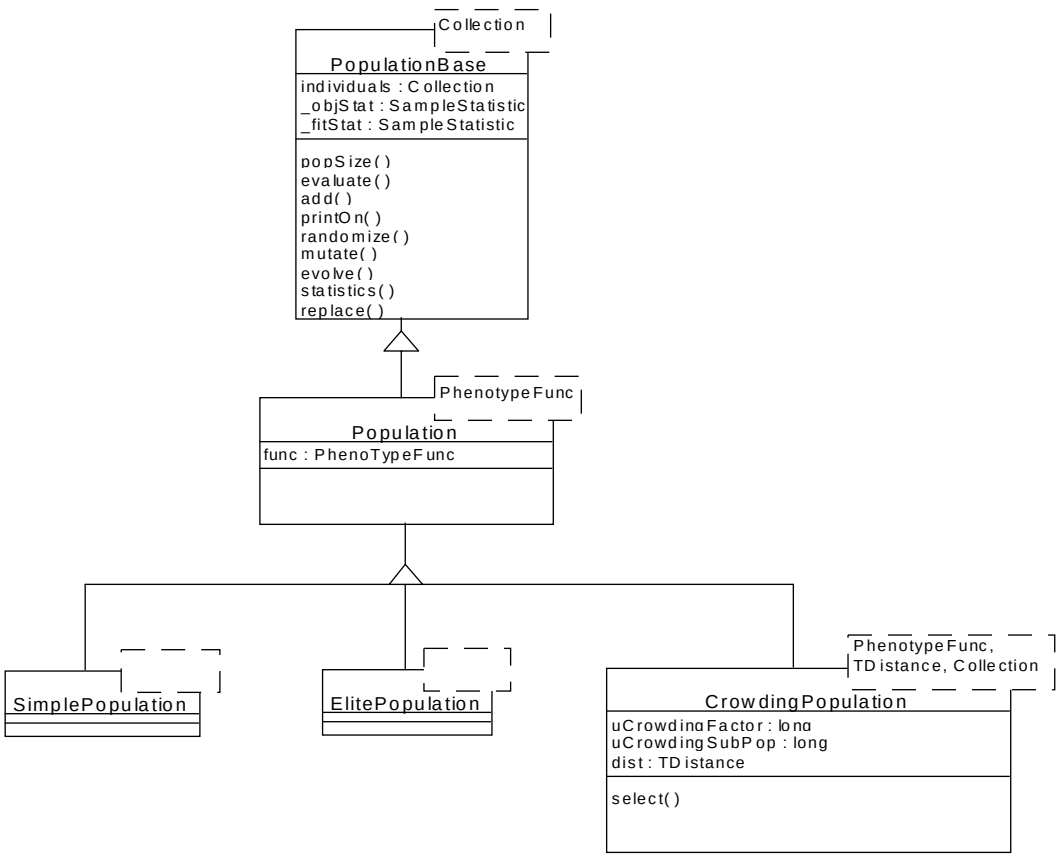

Figure 3 The Population class hierarchy.

Objective function

The objective function is a function object (a function encapsulated by a class) returning the raw fitness of an individual :

```
class F1
{
public:
    double operator()(const TGAObj* ind)
    {
      .
      .
      .
    }
};
```

As shown in Figure 3, a function object is one of the parameters of the **Population** template class.

Each instance of class **TGAObj** contains two member variables, `flObjValue` and `flFitness`, which store respectively the objective value and fitness value. The variable `flObjValue` stores the value returned by the objective function, and `flFitness` is the scaled value of `flObjValue` (if scaling is required). It is the value `flFitness` used for selection for crossover.

Selection scheme

The selection methods provided by TOLKIEN are : roulette-wheel selection (with and without replacement), tournament selection, linear ranking and stochastic universal sampling.

Scaling scheme

In some situations the raw objective values cannot be used for selection, cases include :

- the range of the objective function includes negative values
- function minimization
- the selective pressure is not high enough

TOLKIEN provides three scaling schemes : linear scaling, sigma truncating, power scaling.

FUNCTION OPTIMIZATIONS USING TOLKIEN

The task of this sample application is to optimize the following function HMB, which is Himmelblau's function modified for maximization :

$$HMB(x, y) = 200 - (x^2 + y - 11)^2 - (x + y^2 - 7)^2$$

The parameters of HMB are represented by a binary chromosome of length thirty - the first fifteen bits are used to represent the value for *x*, the rest fifteen bits that for *y*. Each of the two parameters is in the range [-6,6].

The fitness function is defined as follows :

```
ParameterMap HMBmap(15,-6,6); // map a fifteen bits binary value to
                              // the range [-6,6]
class HMB
{
public:
    double operator()(const TGAObj * ind )
    {
      double x = HMBmap((const BinHaploid *) ind, 0);
      double y = HMBmap((const BinHaploid *) ind, 1);
      double a = x * x + y - 11;
      double b = x + y * y - 7;
      return 200 - a * a - b * b;
    }
};
```

The source code for the program is as follows :

```
1  void   main()
2  {
3    float     flMRate = 0.03;    // mutation rate
4    float     flXRate = 1;  // crossover rate
5    float     flGap      = 0.5;   // generation gap
6    unsigned uPopSize = 50; // population size
7    unsigned uGen = 100;    // number of generations to perform
8    //
9    // the GA used takes the following parameters
10   //
11   //   roulette wheel selection scheme
12   //   single-point crossover with crossover rate = 1.0
13   //   mutation rate   = 0.03
14   //   population size = 50
15   //   generation gap  = 0.5
16   //
17   //   offsprings are inserted into the population
18   //   in a random manner

19   //
20   //   creates the initial population of 50 instances
21   //   of class HMBind
22   //   the simplest population type (class Population) are
23   //   used in which old offsprings are replaced by new
24   //   offsprings in a random manner
25   //
26       int i;
27       PopulationBase<>* pPop = new SimplePopulation<HMB>();
28       for (i=0; i<uPopSize; i++)
29            pPop->add(new BinHaploid(30,true));
30       SelectionScheme<>* pSelect = new TournSelect<>();
31       Crossover* pXover = new MultiPtCrossover(2, flXRate);
32       GeneticAlgorithm<>   ga(pPop, pSelect, pXover,
33                                flMRate, flGap);

34       while (ga.trials() < uGen)
35              ga.evolve();

36       cout << “Selection performed : “ << ga.numSelected() << endl;
37       cout << “Crossover performed : “ << ga.numCrossed() << endl;
38       cout << “Best Individual : “ ;
39       ga.bestInd()->printOn(cout);
40       cout << endl;
41       cout << “Worst Individual : “ ;
42       ga.worstInd()->printOn(cout);
43       cout << endl;
44       cout << “Online performance : “ << ga.objOnLine() << endl;
45       cout << “Offline performance : “ << ga.objOffLine() << endl;
46  }
```

As show in lines 27-33, the **GeneticAlgorithm** object can be constructed using various combinations of population structures, selection schemes, and crossover schemes. For example, these lines can be replaced by the follow code segment :

```
PopulationBase<>* pPop = new ElitePopulation<HMB>();
for (i=0; i<uPopSize; i++)
     pPop->add(new BinHaploid(30,true));
SelectionScheme<>* pSelect = new RW_Select<>();
Crossover* pXover = new UniformPtCrossover(2, flXRate);
GeneticAlgorithm<>   ga(pPop, pSelect, pXover,
                        flMRate, flGap);
```

In addition, it is also possible to have a mixed population of binary haploids and binary diploids, You can replace line 28-29 with the follow code :

```
for (i=0; i<uPopSize; i++)
     if (i % 2)
         pPop->add(new BinHaploid(30,true));
     else
         pPop->add(new BinDiploid(30,true));
```