Algoritmos Genéticos en C++

Antonio M. Carpintero Sánchez
Dep. Computación e IA. Facultad de Informática
Universidad Politécnica de Madrid
email: pigmalio@usera.dia.fi.upm.es

1.- Introducción

El Algoritmo Genético (AG) [4] se ha destacado como técnica adecuada para abordar problemas de búsqueda y de optimización. Para esto, se apoya en los procesos genéticos de los organismos biológicos, que a lo largo de generaciones van desarrollándose de acuerdo con los principios de selección natural y la supervivencia del mejor adaptado. Ahora bien, a la hora de llevar a la práctica todas estas técnicas, el usuario, a menudo tiene que recurrir a implementaciones particulares para poder abordar un determinado problema. Este proceso de implementación, es a veces tan tedioso que aparta al usuario del verdadero objetivo de su trabajo, que es el de resolver determinado problema y no el de encontrar un error entre miles de líneas de código. Es por lo que la disponibilidad de herramientas que faciliten la aplicación del AG a problemas reales esta tomando sitio entre los usuarios de AG.

2.- Planteamiento

El propósito del diseño para obtener un sistema de soporte a los AGs es que sea lo más general posible y, por tanto, debe contemplar aquellas variaciones más importantes que puedan aparecer en las numerosas variantes del AG.

Así pues, se puede comenzar con un breve enunciado de la técnica empleada a la hora de abordar un problema con AGs, de forma que permita descubrir aquellas entidades relevantes en el proceso así como la información necesaria para su resolución.

Dado un determinado problema, lo primero que se hace es determinar las variables que lo caracterizan, así como la naturaleza de las mismas. Es decir, si se pretende optimizar, por ejemplo, la producción de una cadena de montaje se deben aislar aquellos parámetros susceptibles de ajuste y que afecten al objetivo (tiempos de las unidades procesadoras, velocidad de procesamiento, etc.). Dichos parámetros pueden ser tanto números reales, como enteros o sencillamente símbolos, componiendo lo que se denomina "*fenotipo*".

A continuación, se construye una función que permita, a partir de unos valores de dichas variables, determinar la bondad del fenotipo propuesto. Esta función se denomina "*fitness*" o "*función de bondad*" (conveniencia), y es la que determina la calidad de la solución, como por ejemplo, en el problema del viajante, sería la longitud de trayecto.

Ahora bien, ya se tiene modelado el problema a tratar, entonces es el momento del modelado genético. Aquí, se diseña una representación del fenotipo, denominada *genotipo*, y en cual cada variable del fenotipo estaría representada por, al menos, un *gen*, y los operadores genéticos de cruce y mutación así como una función de traducción que traduzca el genotipo al fenotipo deseado. El diseño del genotipo es una cuestión delicada, ya que por un lado hay que considerar la representatividad que presente, es decir, si se tiene un genotipo que puede tomar N valores distintos, pero de los cuales M corresponden a codificaciones incorrectas del fenotipo (en el problema de viajante serían rutas incorrectas, por ejemplo, que se pase dos veces por la misma ciudad), entonces se sufre una pérdida de rendimiento sustancial. Y, por otro lado, se debe de tener cuidado en el diseño de los operadores genéticos para que generen genotipos válidos y que cumplan su misión de búsqueda. Así pues, el diseño del genotipo se realiza en conjunción con el de los operadores genéticos, ya que estos son los encargados de manejar directamente dicha información con el fin de crear nuevos genotipos.

Así, una vez diseñados estos elementos, se está en posesión de los componentes necesarios para formar un *individuo*, el cual contará con su fenotipo, genotipo y operadores genéticos, y al que, además, se le añadirá el valor que tome la función de bondad con su Fenotipo y que determina la cualificación del individuo ante el problema.

Ahora, el paso siguiente es componer la *población* de individuos, que va a ser la unidad de trabajo del AG. Sobre esta población se aplicará el AG, para lo que es necesario implementar un nuevo operador llamado operador de selección que determine un par de individuos para la reproducción.

Como implementación del AG, se parte del más simple, al cual se le dotará de los componentes necesarios, o al menos se consideraran en el diseño, para soportar las extensiones más difundidas. El pseudocódigo del AG será:

```
// Inicializar las poblaciones.
     Poblacion P[CENSO],P_New[CENSO];
// Crear una población aleatoria y calcular su Fitness.
     Random(P);
     FitnessPoblacion(P);
// Bucle de generaciones.
     while (No_Converga) {
       for (j=0; j < CENSO/2; j++){
          // Selección de los congéneres.
          Ind_1 = Select(P);
          Ind_2 = Select(P);
          // Cruce.
          Cross(P[Ind_1],P[Ind_2],Hijo_1,Hijo_2);
          // Mutación de los hijos.
          Mutate(Hijo_1);
          Mutate(Hijo_2);
          // Cálculo del Fitness de los hijos.
          Fitness(Hijo_1);
          Fitness(Hijo_2);
          // Los hijos son individuos en la nueva
          población
          P_New[j]       = Hijo_1;
          P_New[2*j+1]   = Hijo_2;
       }
  // La nueva población pasa a ser la actual.
       P = P_New;
     }
```

3.- Diseño

El diseño orientado a objetos es un método de diseño que abarca el proceso de descomposición orientado a objetos y una notación para representar lógica y físicamente, además de modelos estáticos y dinámicos, el sistema bajo diseño.

En el apartado anterior, se ha descrito someramente la funcionalidad del sistema. Ahora, se pasará a la fase de diseño donde se aislarán los distintos objetos que aparecen en la descripción del sistema, así como los métodos asociados a cada uno y sus relaciones.

3.1.- Identificación las Clases y Objetos

Si se observa detenidamente la definición de objeto que siguiente [1] :

> *Un objeto tiene un estado, un comportamiento, e identidad; la estructura y comportamiento de objetos similares están definidos en su clase común; los términos instancia y objeto son intercambiables.*

Se puede apreciar en ella que, ante todo, un objeto debe mantener un estado y mostrar un comportamiento. Pues bien, siguiendo estas guías y repasando la descripción del sistema anteriormente mostrada, se pueden extraer las siguientes entidades con categoría de objeto:

- Fenotipo: Este objeto es el encargado de soportar las variables que definen el problema. En él se almacenarán y gestionarán las mismas con el fin de presentarlas en la forma adecuada a la función de bondad.

- Genotipo: En este objeto se soportarán los genes que representan el fenotipo correspondiente. Además, debe de dar acceso y operatibilidad a los operadores genéticos con los que estará estrechamente unido.

- Individuo: Este es el objeto que representa una solución particular del problema a tratar. Como miembros cuenta con los dos objetos anteriores: fenotipo y genotipo.

- Población. Este es el objeto que engloba a todos los anteriores. Básicamente, el objeto población está compuesto por un conjunto de Individuos sobre los que se desarrollará el AG.

Así pues la siguiente figura recoge las clases anteriores.

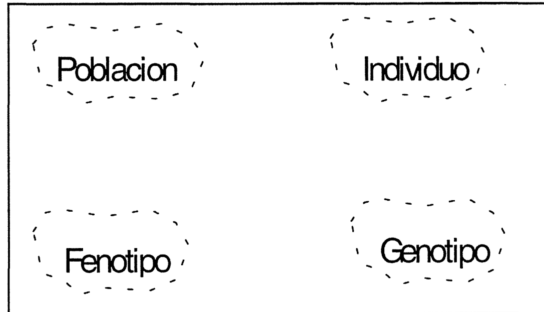


Figura 1.- Diagrama de clases

3.2.-Identificación de la Semántica de las Clases y los Objetos

En este paso, se va a dar significado a las clases y objetos obtenidos en la fase anterior. Para ello, se analiza cada clase desde el punto de vista del interface con sus compañeras de forma que se observa que cosas hace, y puede hacer, para las demás. De esta forma, se descubren cuales son los métodos y miembros con que debe contar cada clase.

Comenzando con el objeto Fenotipo, se puede ver que este es, básicamente, una entidad con capacidad de almacenamiento de las variables del problema a tratar, acceso y algunas operaciones sobre las mismas.

Igualmente, el comportamiento del objeto Genotipo se puede describir como el del Fenotipo. Va a ser una entidad que almacene variables y permita acceso a las mismas. Por tanto, se puede vislumbrar que ambos objetos son realmente instancias de una misma clase que, por las características funcionales requeridas, bien puede ser una clase *Vector*.

El objeto Individuo, como se ha dicho anteriormente, cuenta con un objeto Genotipo y otro Fenotipo, por tanto, inicialmente cuenta con estos miembros.

Finalmente, el objeto Población es aquel que cuenta con un número determinado de Individuos y que, además, soporta el engranaje que implementa el AG, por lo que van a ser las necesidades propias del AG las que determinen los métodos de la clase Población y del resto. Así

pues, si se sigue el pseudocódigo del AG anterior, el primer método que aparece es el constructor propio de la clase Población el cual toma como parámetro el número de individuos que componen la población (censo). A continuación, se observa que es necesario un método de inicialización aleatoria de los individuos de la población. Este método pertenece a la clase Individuo, ya que cualquier valor que se le de al Genotipo depende del problema y su codificación. El siguiente método que aparece es *FitnessPoblacion*, el cual se encarga de evaluar a cada Individuo de la Población. La ubicación de este método es más conflictiva, ya que hay que decidir que componente va a soportar la función de bondad del problema. Por un lado, se puede responsabilizar al objeto Población de la tarea, de forma que el método *FitnessPoblación* iría recorriendo los individuos evaluándolos con la función de bondad. Y, por otro, puede ser el Individuo el encargado de *autoevaluarse,* con lo que la función de bondad quedaría bajo su responsabilidad. Ambas alternativas son perfectamente válidas, aunque quizás sea esta última la que más se aproxime a la realidad y por tanto será la seleccionada. A continuación, aparece el método *Select* el cual se encarga de seleccionar un individuo de la población en función de la bondad del mismo. Este método necesita acceder a todos los Individuos de la Población y obtener de ellos su probabilidad de ser seleccionados. Ahora bien, un Individuo, por si solo no puede calcular su probabilidad de ser seleccionado ya que depende de la bondad del resto de los individuos, por tanto el individuo cuenta con un miembro *Probabilidad*, en el cual se almacena la probabilidad de ser almacenado, así como un método de acceso, *GetProbabilidad*, y otro de asignación, *SetProbabilidad.* El método que se encarga de asignar las probabilidades de cada individuo estará en la clase Población ya que es aquí donde se tiene una visión global de los individuos. Dicho método se llama *Normaliza* y, básicamente, lo que hace es recorrer los individuos y calcular la probabilidad de selección de cada uno en función de su bondad para posteriormente asignársela. Seguidamente, vienen los métodos de *Cruce* y *Mutación*, los cuales son responsabilidad de los Individuos, ya que sólo ellos saben como se tiene que cruzar y mutar. Como colofón, y dado el carácter generacional del AG, se puede definir un método, *OneGeneration*, que sea el encargado de implementar una generación completa del AG.

Evidentemente, a estos métodos hay que añadir el típico de asignación de objetos y copia.

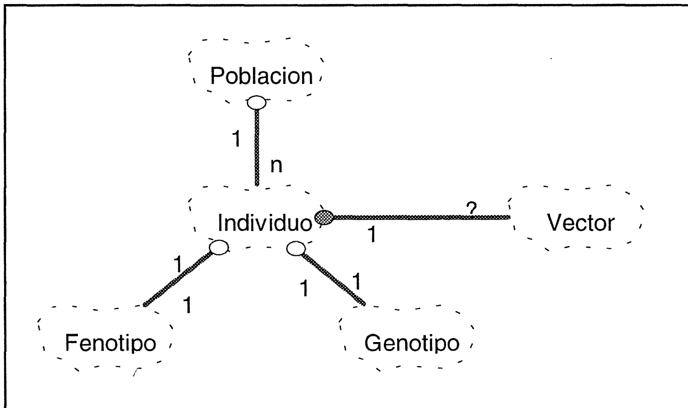


Figura 2.- Jerarquía de clases.

Como se muestra en la figura 2, una Población cuenta con *n* Individuos, cada uno de ellos tiene asociado un Fenotipo y un Genotipo, además la clase Individuo utiliza la clase Vector como contenedor del Fenotipo y Genotipo.

3.3.-Identificación de las Relaciones entre las Clases y Objetos

En esta fase, se identifican las interacciones entre los objetos y clases. Partiendo del razonamiento expuesto en la fase anterior, se obtiene las relaciones (herencia, pertenencia, instancia, etc.) entre los distintos objetos. Como actividades de esta fase, se distinguen:

1. La primera, es encontrar patrones entre las clases definidas, lo que puede causar una reestructuración del sistema de clases.

2. Decidir la visibilidad entre las clases, es decir, como una clase ve a otra o, en caso contrario, a cual de ellas no tiene que ver.

Para la búsqueda de patrones, se centrará la atención en el comportamiento de las clases Fenotipo y Genotipo que, como se ha dicho anteriormente, ambas pueden ser perfectamente instancias de una clase Vector. Esta clase Vector, es el clásico ejemplo de clase parametrizada o *template*, donde a partir de una plantilla de clase, y mediante uno o varios

parámetros, se puede describir el comportamiento de varias clases, como es el caso del Fenotipo y el Genotipo (figura 3).

En lo referente a la segunda actividad, cabe destacar que la clase Población tan sólo ve a los Individuos, y estos sólo a sus Fenotipos y sus Genotipos, soportados por la clase Vector. Esta decisión es coherente con la realidad. La Población da soporte al AG, el cual selecciona, evalúa, cruza y muta a los Individuos, sin importarle como lo hagan ya que son los propios Individuos los que se encargan de manejar sus Genotipos y Fenotipos para tales operaciones.

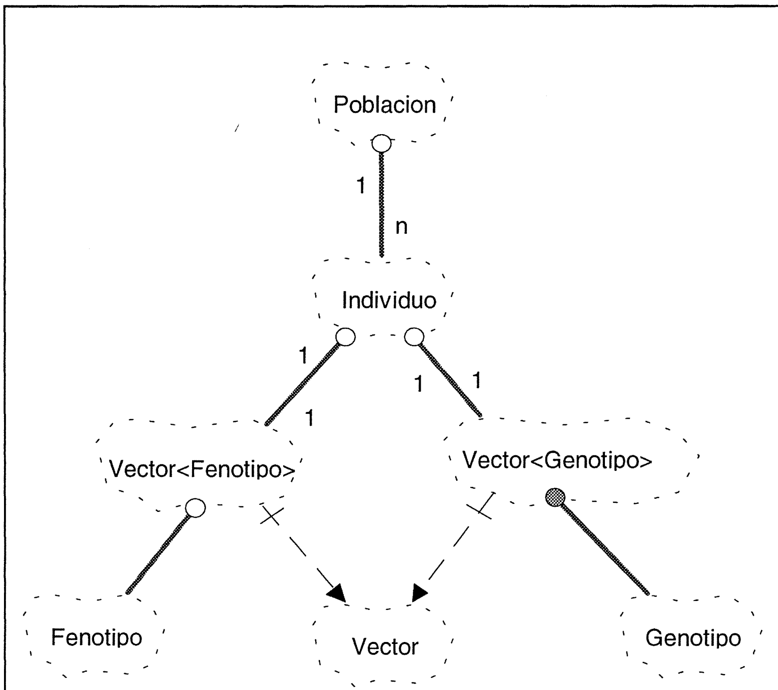


Figura 3.- Utilización del template Vector.

Ahora bien, uno de los propósitos del sistema era el de dotar al usuario de una herramienta que permitiese, por un lado, abordar de forma cómoda un determinado problema, y por otro, implementar nuevas variantes del AG con un mínimo esfuerzo. Para la primera consideración hay que tener en cuenta aquellos elementos del diseños que son dependientes del

problema en cuestión y que, claramente, sería la clase Individuo. En cambio, para la segunda consideración, depende del alcance de la variante del AG que se desee implementar se afectará a más o menos elementos. Ya que hay variantes que consisten en modificar el mecanismo de selección lo cual está dentro de Población, y otras que lo que hacen es añadir nuevos operadores con lo que se induce un cambio tanto en los Individuos como en el AG de la Población.

En la solución actual, si se desea implementar una AG que utilice números reales como Genotipo, es necesario construir una clase *Individuo_Real*, lo cual es inevitable, y modificar en la clase Población todas aquellas sentencias donde se haga referencia a algún individuo (miembros, variables locales, parámetros de métodos, etc.) y a continuación recompilar todos los fuentes.

Ahora bien, si se define un Individuo arquetipo, el cual presente a la clase Población un interfaz que más o menos compendie todo aquello que necesite de él para el desarrollo del AG, entonces se puede diseñar la clase Población de forma que maneje este tipo de Individuo Abstracto sin preocuparse de la verdadera implementación del mismo. De esta forma, la dependencia del problema por parte de la clase Población quedaría solucionado, ya que ésta sólo trabaja con Individuos Abstractos de los que requiere unos servicios que serían los que son dependientes del problema.

Esta consideración obliga a un cambio en el diseño, en donde se añadirá una nueva clase, *AbsIndiv*, que servirá como interface entre la Población y los Individuos. Esta nueva clase, será una *clase abstracta* que constará de un conjunto de métodos reclamados por el AG y que se espera que todos los individuos los tenga, por lo que serán estos los encargados de redefinirlos, ajustándolos al problema concreto. Para esto, la clase Individuo hereda de la clase AbsIndiv de forma que como mínimo, debe definir los métodos declarados en AbsIndiv además de aquellos que, sean necesarios por las particularidades del problema en cuestión.

Con lo que el diseño quedaría según la estructura de clases expuesta en la figura 4:

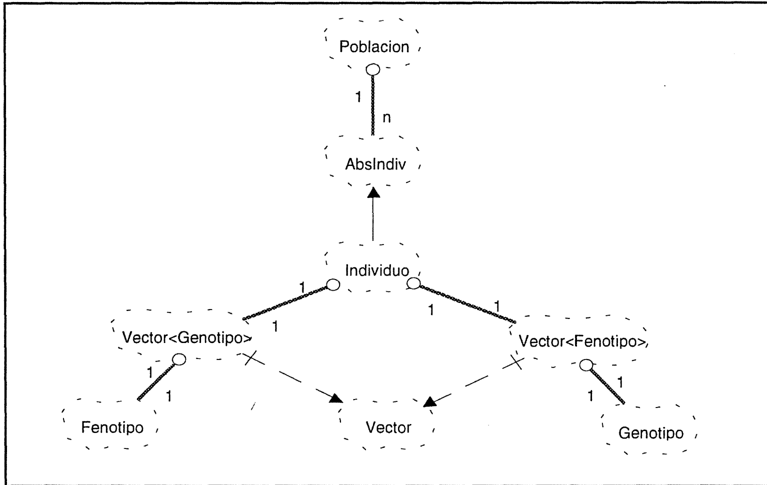


Figura 4.- Jerarquía de clases con la inclusión de la clase AbsIndiv.

En este nuevo diseño, la implementación de una nueva variante de AG que afectase, por ejemplo, al mecanismo de selección tan sólo habría que heredar de Población la nueva clase y reescribir el método nuevo de selección (Select) y el encargado de realizar una iteración del GA (OneGeneration) para que lo llamase. Así, por ejemplo, una versión de la variante Genitor que aparece en [2]; cuyo pseudocódigo corresponde a:

```
/* Inicializar la población. */
Poblacion P[CENSO];
/* Crear una población aleatoria y calcular su
   Fitness. */
Random(P);
FitnessPoblacion(P);
/* Bucle de generaciones. */
while (No_Converga) {
 /* Selección de los congéneres. */
 Ind_1 = Select(P);
 Ind_2 = Select(P);
 /* Cruce. */
```

```
        Cross(P[Ind_1],P[Ind_2],Hijo);
        /* Mutación del hijo. */
        Mutate(Hijo);
        /* Cálculo del Fitness del hijo. */
        Fitness(Hijo);
        /* Reemplazar un individuo de la población
              por el nuevo individuo. */
        Replacement(P,Hijo);
    }
```

En donde, en cada generación, tan sólo se reproduce una pareja y se reemplaza un sólo individuo de la población, se puede implementar reescribiendo el método *OneGeneration* y añadiendo el método *Replacement*. La clase *Genitor* se situaría en diseño según se muestra en la figura 5:

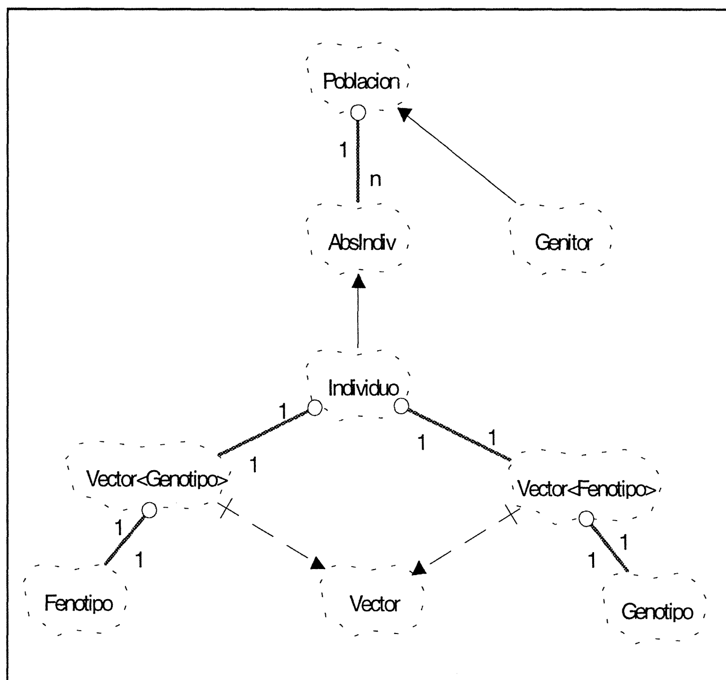
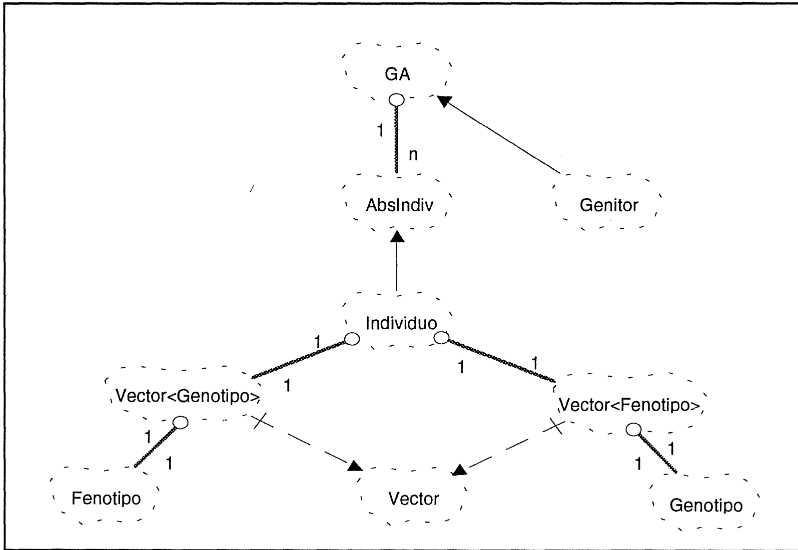


Figura 5.- Derivación de la clase Genitor.

Una última consideración es que, a la vista de las posibilidades de expansión en cuanto al soporte de variedades de AG, sería conveniente que el nombre de la clase base de la que herede cuente con un nombre representativo y ya que es el soporte de la versión tradicional del AG, se ha creído conveniente renombrarla como clase GA.

Por tanto, el diseño final resultante de esta fase puede resumirse como aparece en la figura 6:



Figigura 6.- Jerarquía final.

3.4.- Desarrollo de las Clases y Objetos

A continuación, se muestra el código final de los distintos objetos que aparecen en el diseño anterior, y se comentarán aquellos puntos oscuros que puedan aparecer, aunque por la sencillez de la codificación no será necesario.

Para su implementación, se ha seguido el criterio de la portabilidad entre plataformas, evitando las particularidades propias de cada compilador. Así, todo el código mostrado ha sido probado tanto en PC como en workstation bajo Unix.

Comenzando por las alturas de la jerarquía, el código de soporte de la clase GA es:

```
#ifndef _SGA_H_
#define _SGA_H_

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <float.h>
#include <math.h>
#include "ctes.h"
#include "vector.h"
#include "random.h"
#include "absindv.h"

#define ASCENDENTE      1
#define DESCENDENTE     2

class GA {

protected:
    int            Censo;          // Censo of GA.
    float          MinFitness;     // Minimo fitness.
    float          MaxFitness;     // Maximo fitness
    long           Generation;     // Generacion actual
    AbsIndiv       **Poblacion;    // Puntero a la poblacion.
    AbsIndiv       **PobTmp;       // Población Temporal.
    float          ProbMutation;   // Probabilidad de
   Mutacion.
    float          ProbCross;      // Probabilidad de
   Cruzamiento.

    // Ordena un vector de punteros a individuos.
    void           SortAbsIndiv(AbsIndiv **P, int ORDEN =
ASCENDENTE);

  public:
```

```
// Constructores.
GA ( AbsIndiv *indiv, int NumIndiv ) ;
// Constructor de copia.
GA ( const GA & P );
// Destructor.
virtual    ~GA();

// Retorna el censo de la poblacion.
int             GetCenso(void) {return Censo;}
// Visualiza la poblacion.
void            Show(void);
// Retorna el indice del individuo con mayor fitness.
int             GetMaxFitness(void) const;
// Retorna el indice del individuo con menor fitness.
int             GetMinFitness(void) const;
// Inicializa aleatoriamente individuos desde iStart hasta
iEnd.
void            Random(int iStart = 0,int iEnd=0);
// Selecciona un individuo que sea distinto de "i"
int             Select(int i = -1);
// Realiza una iteracion del GA.
void            OneGeneration(void);
// Ordena la poblacion segun ORDEN.
void            SortPob(int ORDEN = ASCENDENTE);
// Ordena la poblacion temporal segun ORDEN.
void            SortPobTmp(int ORDEN = ASCENDENTE);
// Normaliza las probabilidades de la poblacion temporal.
void            NormalizeTmp(void);
// Normaliza las probabilidades de la poblacion.
void            NormalizePob(void);
// Normaliza las probabilidades de un vector de punteros a
individuos.
void            Normalize(AbsIndiv **);
// Reinicializa las poblaciones.
void            Reset(void);
// Retorna el numero de generacion actual.
long            GetGeneration(void) const {return
Generation;};
// Calcula los fenotipos de los individuos.
```

```
    void             SetPhenotype(void);
    // Calcula los fitness de los individuos.
    void             SetFitness(void);
    // Sustituye los individuos [iStart..iEnd] de la poblacion
   por los
    // de la poblacion temporal [iStart..iEnd].
    void             Replace(int iStart , int iEnd);
    // Asigna la probabilidad de mutacion.
    void             SetProbMutation(float P) { ProbMutation =
   P;}
    // Asigna la probabilidad de cruce.
    void             SetProbCross(float P) { ProbCross = P;}
    // Intercambia dos punteros a individuos.
    void             Swap(AbsIndiv* &Ind1,AbsIndiv* &Ind2);

    // Operadores.
    // Operador de asignacion.
    GA&              operator=(const GA& V);
    // Retorna una referencia al individuo i.
    AbsIndiv&        operator[] (int i);

};

#endif
```

La implementación de esta clase corresponde al siguiente fichero:

```
#include "ga.h"

//  AbsIndiv *indiv : Puntero a un individuo representativo de
la población
//  int       NumIndiv: Censo de la población.
 GA::GA(AbsIndiv *indiv, int NumIndiv){
    // Validación de los parámetros de entrada.
    assert(NumIndiv > 0);
    assert(indiv != NULL);

    // Reserva de memoria.
    Poblacion   = new AbsIndiv *[Censo = NumIndiv];
```

```
    assert(Poblacion != 0);
    PobTmp      = new AbsIndiv *[Censo];
    assert(PobTmp != 0);

    // Creación de los individuos a partir del individuo
   "indiv"
    for ( int i = 0; i < Censo; i++ ) {
        Poblacion[i] = indiv->New();
        assert(Poblacion[i] != NULL);
        PobTmp[i]    = indiv->New();
        assert(PobTmp[i] != NULL);
    }

    // Inicicialización con valores por defecto.
    MaxFitness = -FLT_MAX;
    MinFitness =  FLT_MAX;
    Generation =  0;
    ProbMutation = 0.001;
    ProbCross    = 0.9;
}

void GA::Reset(void) {
    // Inicialización aleatoria
    Random(0,Censo);
    SetFitness();
    NormalizePob();

    MaxFitness = -FLT_MAX;
    MinFitness =  FLT_MAX;
    Generation =  0;
}

// Constructor de copia.
GA::GA( const GA & P){
    Censo = P.Censo;
    Poblacion = new AbsIndiv *[Censo];
    assert(Poblacion != 0);

    for ( int i = 0; i < Censo; i++) {
```

```
        Poblacion[i] = P.Poblacion[i]->New();
        assert(Poblacion[i] != 0);
        PobTmp[i] = P.PobTmp[i]->New();
        assert(PobTmp[i] != 0);
    }

    MaxFitness = P.MaxFitness;
    MinFitness = P.MinFitness;
    Generation = P.Generation;
    ProbMutation = P.ProbMutation;
    ProbCross    = P.ProbCross;

}

// Destructor.
GA::~GA(){
    for (int i = 0; i < Censo; i++) {
        delete  Poblacion[i];
        delete  PobTmp[i];
    }
    delete[] Poblacion;
    delete[] PobTmp;
}

AbsIndiv& GA::operator[] (int i) {
    assert(i < Censo && i >= 0);
    return *Poblacion[i];
}

void           GA::Show(void){
    for (int i = 0; i < Censo; i++)
        Poblacion[i]->Show();
}

// Retorna el indice de la criatura que tiene mayor fitness.
int  GA::GetMaxFitness(void) const {
    float          Max = -FLT_MAX;
    float          Aux;
```

```
    int             C = 0;
    for (int i = 0; i < Censo; i++)
        if ((Aux = Poblacion[i]->GetFitness()) > Max) {
            Max = Aux;
            C = i;
        }
    return C;
}

int  GA::GetMinFitness(void) const {
    float           Min = FLT_MAX;
    float           Aux;
    int             C = 0;

    for (int i = 0; i < Censo; i++)
        if ((Aux = Poblacion[i]->GetFitness()) <= Min) {
            Min = Aux;
            C = i;
        }
    return C;
}

void GA::NormalizeTmp(void){
    Normalize(PobTmp);
}

void GA::NormalizePob(void) {
    Normalize(Poblacion);
}

// Asigna las probabilidades de seleccion.
// Tiene mayor probabilidad aquel que tenga menor Fitness.
void GA::Normalize(AbsIndiv **Pob) {
    float factor =  0.0, aux;
    float maxF   = -FLT_MAX,
          minF   =  FLT_MAX,
          Cte    =  0.0;
    int   i;
```

```
    // Cálculo del máximo y el mínimo.
    for ( i = 0; i < Censo; i++) {
        aux = Pob[i]->GetFitness();
        if (aux > maxF)
            maxF = aux;
        if (aux < minF)
            minF = aux;
    }

    factor = 0.0;
    if ( fabs(maxF - minF) > 1.0E-4 ) {
        Cte = maxF-minF;
        for ( i = 0 ; i < Censo; i++)
            factor += Cte - (Pob[i]->GetFitness()-minF);
    } else {
        Cte     = 0.0;
        minF    = 0.0;
        for ( i = 0 ; i < Censo; i++)
            factor +=  fabs(Pob[i]->GetFitness());
        // Si todos los individuos tienen un fitness muy muy
          pequeño.
        if ( factor < 1.0E-8 ) {
            factor = 1.0;
            Cte    = 1.0/(float)Censo;
        }
    }

    for (i = 0 ; i < Censo; i++) {
        aux = Pob[i]->GetFitness();
        Pob[i]->SetProbability( fabs((Cte-(aux-minF))/factor));
    }

}

void GA::SortPob(int ORDEN) {
    SortAbsIndiv(Poblacion,ORDEN);
}
```

```
inline void GA::SortPobTmp(int ORDEN) {
    SortAbsIndiv(PobTmp,ORDEN);
}

// Ordena la poblacion por su fitness.
void GA::SortAbsIndiv(AbsIndiv **P, int ORDEN ) {
    int i,j;
    AbsIndiv *Aux;
    for ( i = 1; i < Censo; i++)
        for ( j = Censo-1; j >= i; j--)
            if ( ORDEN == ASCENDENTE ) {
              if (  P[j-1]->GetFitness()> P[j]->GetFitness()) {
                    Aux        = P[j-1];
                    P[j-1] = P[j];
                    P[j]   = Aux;
                }
            } else {
               if ( P[j-1]->GetFitness()< P[j]->GetFitness()) {
                    Aux        = P[j-1];
                    P[j-1] = P[j];
                    P[j]   = Aux;
                }
            }
    return;
}

#define MAX_INTERATIONS         5
int GA::Select(int k) {
        float   fProb   = 0.0;
        float   fSum    = 0.0;
        int     i       = 0,
                j;
        int     iter    = 0;

         // Probability != 0.0
         do {
             i = ::Random()%Censo;
             while ( (fProb = FRandom()) == 0.0);
```

```
            fSum  = Poblacion[i]->GetProbability();
            j = 0;
            while ( j < Censo &&  fSum < fProb ) {
               i=(++i%Censo);
               fSum += Poblacion[i]->GetProbability();
               j++;
            }
            if ( i >= Censo )
               i = ::Random()%Censo;
            if ( iter++ > MAX_INTERATIONS) {
               i = ::Random()%Censo;
            }
           } while ( i == k);
return (i);
}

void    GA::OneGeneration(void) {
    int i,P1,P2;

    for ( i = 0; i < Censo; i++) {
        // Seleccion de los congeneres.
        P1 = Select();
        P2 = Select(P1); // Selecciona uno que sea distinto a
P1.
        // ¨ Hay cruce ?
        if (FRandom() <= ProbCross) {
            // Seleccion aleatoria de quien es el padre y quien
            // la madre.
            if ( FRandom() <= 0.5 )
                    Poblacion[P1]->Cross(Poblacion[P2],
                  PobTmp[i]);
            else
                    Poblacion[P2]->Cross(Poblacion[P1],
                  PobTmp[i]);
        } else {
            *PobTmp[i] = *Poblacion[P1];
        }
        // Mutacion del descendiente.
        PobTmp[i]->Mutate(ProbMutation);
```

```
        // Calculo del fenotipo.
        PobTmp[i]->SetPhenotype();

        // Evaluación del nuevo individuo.
        PobTmp[i]->SetFitness();
    }

    // Normalizacion de las probabilidades de seleccion de la
    // poblacion temporal.
    NormalizeTmp();

    // Reemplazo de toda la poblacion.
    Replace(0,Censo);

    // Nueva generacion.
    Generation++;

    return ;
}

GA& GA::operator=( const GA& P){
    int i;
    if ( this == &P) {
        printf("GA<AbsIndiv>::operator= : Asignacion del mismo
           objeto.\n");
        return *this;
    }

    if ( Censo != P.Censo ) {
        for ( i = 0; i < Censo; i++) {
            delete Poblacion[i];
            delete PobTmp[i];
        }
        delete[] Poblacion;
        delete[] Poblacion;
        Censo = P.Censo;
        Poblacion = new AbsIndiv*[Censo];
        assert(Poblacion != NULL);
```

```
        PobTmp     = new AbsIndiv*[Censo];
        assert(PobTmp != NULL);

        for ( i = 0; i < Censo; i++) {
            Poblacion[i] = P.Poblacion[i]->New();
            assert(Poblacion[i] != 0);
            PobTmp[i] = P.PobTmp[i]->New();
            assert(PobTmp[i] != 0);
        }

    } else {
        for ( i = 0 ; i < Censo; i++)
            *Poblacion[i] = *P.Poblacion[i];
            *PobTmp[i] = *P.PobTmp[i];
    }

    MaxFitness = P.MaxFitness;
    MinFitness = P.MinFitness;
    Generation = P.Generation;
    ProbMutation = P.ProbMutation;
    ProbCross    = P.ProbCross;

    return (*this);
}

void    GA::SetFitness(void) {
    for ( int i = 0 ; i < Censo; i++)
        Poblacion[i]->SetFitness();
}

// Generate random indiv from iStart to iEnd indi.
void    GA::Random(int iStart, int iEnd) {
        assert(iStart <= Censo );
        assert(iEnd <= Censo);
        iEnd = (iEnd ? iEnd:Censo);
        for ( int i = iStart ; i < iEnd; i++) {
            Poblacion[i]->Random();
            Poblacion[i]->SetPhenotype();
```

```
        }
}

void    GA::SetPhenotype(void) {
        for ( int i = 0; i < Censo; i++)
            Poblacion[i]->SetPhenotype();
}

void   GA::Swap(AbsIndiv*&Ind1,AbsIndiv* &Ind2) {
  AbsIndiv*       Aux;
  Aux    = Ind1;
  Ind1   = Ind2;
  Ind2   = Aux;
}

// INTERCAMBIA los individuos de la GA Src a partir de iStart.
void   GA::Replace(int iStart ,int iEnd ) {
        assert(iStart <= iEnd);
        assert(iEnd <= Censo);
        for ( int i = iStart; i < iEnd; i++)
           Swap(Poblacion[i],PobTmp[i]);
        return;
}
```

Como comentario, cabe resaltar el mecanismo de construcción que tiene la clase GA. El constructor de GA tiene como argumento un puntero a un individuo AbsIndiv y el censo de la población deseada. Asi pues, para crear una población, como se verá posteriormente, basta con definir un individuo representante para que, a partir de él, se genere el resto de la población.

Una vez definida la clase GA, la siguiente clase en la jerarquia es la clase abstracta AbsIndiv, cuya definición es la siguiente:

```
#ifndef _ABSINDV_H_
#define _ABSINDV_H_

class AbsIndiv {
    public:
```

```
        // Inicializa aleatoriamente al individuo.
        virtual void    Random() =0;

        // Visualiza al individuo.
        virtual void    Show()  =0;

        // Retorna el fitness.
        virtual float   GetFitness() =0;

        // Calcula el fitness del individuo.
        virtual void    SetFitness()  =0;

        // Calcula el fenotipo a partir de genotipo.
        virtual void    SetPhenotype()  =0;

        // Asigna la probabilidad.
        virtual void    SetProbability(float p) =0;

        // Retorna la probabilidad de seleccion del individuo.
        virtual float   GetProbability()  =0;

        // Cruza al individuo (madre) con el padre (*Padre)
         para dar
        // un descendiente (*Hijo).
        virtual void    Cross(AbsIndiv *Padre , AbsIndiv
         *Hijo)=0;

        // Muta al individuo con probabilidad de mutacion p.
        virtual void    Mutate(float p) =0;

        // Crea un nuevo individuo del mismo tipo que ,l,
         retornando
        // un puntero al mismo.
        virtual AbsIndiv*   New()  =0;

};

#endif
```

Esta es pues la clase de la que van a derivar todos los individuos que se traten en el sistema, por lo que la definición sirve de plantilla de qué métodos debe contar cualquier individuo. Como se puede ver, la clase AbsIndiv continene métodos virtuales puros, lo que la hace una clase abstracta, es decir, no pueden instanciarse objetos de dicha clase directamente, sino que tiene que realizarse utilizando herencia.

A continuación, se mostrará la clase Vector que, como se ha comentado anteriormente, es una clase parametrizada o template, ya que se empleará posteriormente en la definición de los individuos.

```
#ifndef _VECTOR__H_
#define _VECTOR__H_
#include <stdio.h>
#include <string.h>
#include <assert.h>

#define MIN(a,b) (a<b?a:b)

template < class T >class Vector {
    T              *Ptr;
    int             NumOfItems;
 public:
       Vector(int L);
       Vector(const Vector <T> &V);
       ~Vector();
       Vector <T> &operator = (const Vector <T> &V);
       T & operator[] (int i);
       T   at(int i) const;
       int  Length(void) const { return NumOfItems; }
       void             ReSize(int NewLength);
};

template <class T>Vector <T> ::Vector(int L)
{
    assert(L > 0);
    Ptr = new T[NumOfItems = L];
    assert(Ptr != 0);
```

```
}

template <class T>Vector <T> ::Vector(const Vector <T> &V)
{
    NumOfItems = V.NumOfItems;
    Ptr        = new T[NumOfItems];
    memcpy(Ptr, V.Ptr, sizeof(T) * NumOfItems);
}

template <class T>Vector <T> ::~Vector()
{
    delete[] Ptr;
    NumOfItems = 0;
}

template <class T>Vector <T> &Vector <T> ::operator = (const
  Vector <T> &V) {
    // Evitar auto asignaciones P = P.
    if (this == &V)
        return *this;
    if (NumOfItems != V.NumOfItems) {
        delete[] Ptr;
        Ptr = new T[NumOfItems = V.NumOfItems];
        assert(Ptr != 0);
    }
    memcpy(Ptr, V.Ptr, sizeof(T) * NumOfItems);
    return *this;
}

template <class T>T & Vector <T> ::operator[] (int i) {
    assert(i < NumOfItems);
    return Ptr[i];
}
template <class T> T Vector <T> ::at(int i) const {
    assert(i < NumOfItems);
    return Ptr[i];
}

template <class T>void Vector <T> ::ReSize(int NewLengt)
```

```
{
    int             Total = MIN(NewLengt, (int) NumOfItems);
    T              *AuxPtr;

    assert(NewLengt > 0);
    AuxPtr = new T[NewLengt];
    assert(AuxPtr != 0);
    for (int i = 0; i < Total; i++)
        AuxPtr[i] = Ptr[i];
    delete[] Ptr;
    Ptr = AuxPtr;
}
```

Hasta este momento, se ha desarrollado la parte del diseño que era independiente del problema a tratar. Ahora, se presentarán varios ejemplos de individuos:

El primer individuo que se presenta realiza la codificación tradicional binaria:

```
#ifndef _BINARY_H_
#define _BINARY_H_

#include "ctes.h"
#include "vector.h"
#include "bitvec.h"
#include "absindv.h"


class Binary: public AbsIndiv  {
private:
    // Convierte Genotype en un vector de unsigned long.
    Vector < unsigned long > GentoL(void);
    // Convierte un trozo del genotipo en unsigned logn.
    unsigned long   BVtoL(int Start, int L);
public:
    Vector <char>     Genotype;       // Genotipo
    Vector <float>    Phenotype;      // Fenotipo
    float             Fitness;        // Fitness del individuo.
```

```
  float          Probability;    // Probabilidad de ser
seleccionado
  float         (*Function)(Binary& indv);
  int            LengthOfGen;    // Numero de bits por gen.
  int            NumberOfGens;    // Numero de genes..

  // Constructores.
  Binary(int NumGens, float (*fFunc)(Binary &),
         int LengGen = DEF_LEN_GEN, float Prob = 1.0);
  Binary(const Binary & C);

  // Operadores Geneticos
  void            Mutate(float fProb = P_MUTATION);
  void            Cross(AbsIndiv *C1, AbsIndiv *Hijo1);

  // Inicializacion aleatoria.
  void            Random(void);
  // Retorna el genotipo.
  Vector <char>       GetGenotype(void) {return Genotype;}
  // Retorna el fenotipo.
  Vector <float>      GetPhenotype(void) {return Phenotype;}

  // Visualiza el individuo.
  void            Show(void);
  // Calcula el fenotipo.
  void            SetPhenotype(void);
  // Retorna el fitness.
  float           GetFitness(void)  {return Fitness;}
  // Retorna la probabilidad de ser seleccionado.
  float          GetProbability(void)  {return Probability;}
  // Asigna la probabilidad de ser seleccionado.
  void            SetProbability(float P) {Probability = P;}
  // Calcula el fitness del individuo.
  void            SetFitness(void);

  // Operador de asignacion.
  Binary & operator = (const Binary & C);

  // Crea un nuevo individuo a partir de ,l retornando
```

```
    // un puntero al mismo.
    AbsIndiv*       New(void) ;

};
#endif
```

Cuya implementación será:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <math.h>
#include "binary.h"
#include "random.h"

Binary::Binary(int NumGens, float (*fFunc)(Binary &),int
LengGen, float Prob):
Genotype(NumGens*LengGen),
Phenotype(NumGens),
Fitness(0.0),
Probability(Prob),
Function(fFunc),
LengthOfGen(LengGen),
NumberOfGens(NumGens)
{}

Binary::Binary(const Binary & C):
Genotype(C.Genotype),
Phenotype(C.Phenotype),
Fitness(C.Fitness),
Probability(C.Probability),
Function(C.Function),
LengthOfGen(C.LengthOfGen),
NumberOfGens(C.NumberOfGens)
{}

Binary & Binary::operator = (const Binary & C) {
    Genotype    = C.Genotype;
```

```
        Phenotype    = C.Phenotype;
        Probability = C.Probability;
        Fitness      = C.Fitness;
        Function     = C.Function;
        LengthOfGen = C.LengthOfGen;
        NumberOfGens= C.NumberOfGens;

        return *this;
    }

    void            Binary::Show(void){
        int i;
        printf("* Binary: Fit[%g]
Prob[%6.5f]\n",Fitness,Probability);
        printf("   Genotype : ");
        for ( i = 0 ; i < Genotype.Length(); i++) {
            printf("%d",Genotype[i]);
        }
        printf("\n");
        printf("   Phenotype: ");
        for ( i = 0 ; i < NumberOfGens; i++) {
            printf(" %g ",Phenotype[i]);
        }
    }

    void            Binary::Random(void) {
        for ( int i = 0 ; i < Genotype.Length(); i++)
            if (FRandom() <= 0.5 )
                Genotype[i] = 0;
            else
                Genotype[i] = 1;
    }

    // Normaliza los valores del genotipo al fenotipo en [0,1].
    void            Binary::SetPhenotype(void){
        // Valor de los genes pasados a unsigned long.
        Vector < unsigned long >VL = GentoL();
        int iNbits = LengthOfGen;
        // Máximo valor representable con iNbits.
```

```
    float fMax = (float)pow((double)2.0,(double)(iNbits))-1.0;

    for (int i = 0; i < NumberOfGens; i++)
        Phenotype[i] = (float)VL[i]/fMax;
}

void            Binary::SetFitness(void) {
        Fitness = (*Function)(*this);
}

// Mutacion.
void            Binary::Mutate(float fProb) {
    for ( int i = 0 ; i < Genotype.Length(); i++)
        if (FRandom() <= fProb )
            if (Genotype[i]==1)
                Genotype[i] = 0;
            else
                Genotype[i] = 1;
}

// Cruce uniforme.
void        Binary::Cross(AbsIndiv* C1, AbsIndiv* Hijo1) {
    Binary          *C,*Hijo;

    // IMPORTANTE: Hay que realizar un casting a la clase
  Binary para
    // que el compilador "vea" los métodos propios de la clase
  Binary.
    C    = (Binary *)C1;
    Hijo = (Binary *)Hijo1;

    for ( int i = 0 ; i < Genotype.Length(); i++)
        if ( FRandom() < 0.5 )
            Hijo->Genotype[i] = C->Genotype[i];
        else
            Hijo->Genotype[i] = Genotype[i];
}

// Considera que los bits de menor peso estan a la izquierda,
```

```
// es decir al reves de la representacion natural:
//
//           136
//       01248624..
//       __________
//       1011011010
unsigned long   Binary::BVtoL(int Start, int L) {
    unsigned long   Aux = 0,
                    Acum = 1;
assert(Start + L <= Genotype.Length());
    for (int j = Start, i = 0; i < L; j++, i++, Acum *= 2)
        if (Genotype[j]==1)
            Aux += Acum;
    return Aux;
}

// Retorna un vector con el valor de los genes traducidos a
// unsigned long.
Vector < unsigned long >Binary::GentoL(void){
    Vector < unsigned long >Aux(NumberOfGens);
    for (int j = 0, i = 0; i < NumberOfGens; i++, j +=
         LengthOfGen)
        Aux[i] = BVtoL(j, LengthOfGen);
    return Aux;
}

AbsIndiv*             Binary::New(void) {
    Binary *pBin = new Binary(NumberOfGens,
Function,LengthOfGen);
    assert (pBin != NULL);
    return pBin;
}
```

Como comentario, cabe resaltar que el constructor de esta clase, además del número de genes y la función de bondad, necesita conocer la longitud de los genes. Este último parámetro lo que determina es la resolución de la representación y dependerá del problema a tratar. En esta clase el operador de cruce que se ha implementado es el denominado cruce uniforme [3].

Otro ejemplo de individuo es la clase Real, la cual implementa una codificación con números reales donde, tanto el Genotipo como el Fenotipo, son dos vectores de números reales con la diferencia que el Fenotipo siempre va a estar normalizado entre cero y uno.

```
#ifndef _REAL_C_H_
#define _REAL_C_H_
#include "ctes.h"
#include "vector.h"
#include "absindv.h"

class Real: public AbsIndiv {
    float           Fitness;        // Fitness del individuo.
    float           Probability;    // Probabilidad de
                                       seleccion
    float           MaxV;           // Maximo valor.
    float           MinV;           // Minimo valor.

    float        (*Function)(Real& I); // Funcion de bondad.

 public:

    // Genotipo.
    Vector < float > Genotype;
    // Fenotipo.
    Vector < float > Phenotype;

    // Constructores.
    Real(int NumGens, float (*fFunc)(Real&),
                      float maxV=100.0,
                      float minV = -100.0 ,
                      float Prob = 1.0);
    Real(const Real& C);

    // Operadores Geneticos.
    void        Mutate(float fProb = P_MUTATION);
    void        Cross(AbsIndiv*C, AbsIndiv *Hijo);
```

```
        // Inicializacion aleatoria.
        void            Random(void);

        // Creacion de un nuevo individuo.
        AbsIndiv*           New(void);

        // Operador de asignacion.
        Real & operator = (const Real & C);

        // Retorna el fenotipo.
        Vector < float >GetPhenotype(void) { return Phenotype; }

        // Visualiza un individuo.
        void            Show(void);

        // Calcula el fenotipo a partir del genotipo.
        void            SetPhenotype(void);

        // Retorna el fitness del individuo.
        float           GetFitness(void) { return Fitness; }

        // Retorna la probabilidad de seleccion.
        float           GetProbability(void) { return Probability;
}

        // Asigna la probabilidad de seleccion.
        void            SetProbability(float P) { Probability = P;
}

        // Calcula el fitness del individuo.
        void            SetFitness(void);

    };
    #endif
```

La implementación para la clase Real es:

```
    #include <stdio.h>
```

```
#include <stdlib.h>
#include <limits.h>
#include <math.h>
#include <float.h>
#include "random.h"
#include "error.h"
#include "real.h"

// NumGens: Numero de variables reales (genes).
// fFunc: Funcion de bondad.
// maxV:  Valor maximo que puede tomar cualquiera de los
  genes.
// minV:  Valor minimo que puede tomar cualquiera de los
  genes.
// Prob:  Probabilidad inicial de seleccion.
Real::Real(int NumGens, float (*fFunc) (Real&), float maxV,
float minV, float Prob):
Genotype(NumGens),
Phenotype(NumGens)
{
    if (maxV <= minV)
        Error("[Real::Real] Invalid <MaxV>=<%g>
<MinV>=<%g>\n", maxV, minV);
    Fitness = FLT_MAX;
    Probability = Prob;
    MaxV = maxV;
    MinV = minV;
    Function = fFunc;
}

Real::Real(const Real & C):
Genotype(C.Genotype),Phenotype(C.Phenotype){
    Genotype  = C.Genotype;
    Phenotype = C.Phenotype;
    Probability = C.Probability;
    Fitness = C.Fitness;
    MaxV = C.MaxV;
    MinV = C.MinV;
    Function = C.Function;
}
```

```
Real & Real::operator = (const Real & C){
    Genotype = C.Genotype;
    Phenotype = C.Phenotype;
    Probability = C.Probability;
    Fitness = C.Fitness;
    MaxV = C.MaxV;
    MinV = C.MinV;
    Function = C.Function;
    return *this;
}

void            Real::Random(void){
    int             L = Genotype.Length();
    float           f = MaxV - MinV;

    for (int i = 0; i < L; i++)
        Genotype[i] = f * FRandom() - MinV;
}

void            Real::Show(void){
    int i ;
    printf("* Real: Fit[%g] Prob[%g]\n", Fitness,
  Probability);
    printf("  Genotype : ");
    for (i = 0; i< Genotype.Length();i++ )
        printf("<%g>", Genotype[i]);
    printf("  Phenotype: ");
    for (i = 0; i< Phenotype.Length();i++ )
        printf("<%g>", Phenotype[i]);
    printf("\n");
}

// Calcula el fenotipo normalizando el genotipo en el
  intervalo [0,1].
void            Real::SetPhenotype(void){
    float           fMax = MaxV - MinV;
    int             L = Genotype.Length();
```

```
        for (int i = 0; i < L; i++)
            Phenotype[i] = (Genotype[i] - MinV) / fMax;
    }

    // La mutacion consiste en asignar un valor aleatorio del
    // intervalo [MinV,MaxV].
    void            Real::Mutate(float fProb){
        int             L = Genotype.Length();
        float           f = MaxV - MinV;

        for (int i = 0; i < L; i++)
            if (FRandom() <= fProb)
                Genotype[i] = f * FRandom() + MinV;
    }

    // El cruce consiste en generar un hijo cuyo genotipo es un
      vector resultante
    // de la suma ponderada aleatoria de los genotipos de los
      padres.
    void            Real::Cross(AbsIndiv* C1, AbsIndiv* Hijo1){
        int             L = Genotype.Length();
        float           s1,s2;

        // IMPORTANTE: Hay que realizar un casting a la clase Real
             para
        // que el compilador "vea" los métodos propios de la clase
             Real.
        Real *C,*Hijo;
        C=(Real *)C1;
        Hijo=(Real *)Hijo1;

        for (int i = 0; i < L; i++) {
            s1 = FRandom() < 0.5 ? 1.0:-1.0;
            s2 = FRandom() < 0.5 ? 1.0:-1.0;
            Hijo->Genotype[i] = (s1*Genotype[i] + s2*C-
>Genotype[i]) / (2.5 - FRandom());
        }
    }

    // Evalua la bondad del individuo.
```

```
void            Real::SetFitness(void){
    Fitness = (*Function) (*this);
}

// Retorna un puntero a un nuevo individuo Real.
AbsIndiv*            Real::New(void){
    Real *pReal = new
Real(Genotype.Length(),Function,MaxV,MinV);
    assert (pReal != NULL);
    return pReal;
}
```

Ahora, se va a mostrar como se implementa una variante del AG, como es la clase Genitor anteriormente descrita:

```
#ifndef _GENITOR_H_
#define _GENITOR_H_

#include "absindv.h"
#include "ga.h"

class Genitor: public GA {

    public:
        // Constructor.
        Genitor(AbsIndiv *indiv, int NumIndiv ):GA(indiv,
         NumIndiv ) {}
        // Implementa una generacion del Genitor.
        void OneGeneration(void);
        // Reemplaza el individuo n de la poblacion temporal
         por el
        // peor individuo (mayor fitness) de poblacion.
        void Replacement(int n);
};

#endif
```

Y la correspondiente implementación:

```
#include "genitor.h"

void Genitor::OneGeneration(void){
    int P1,P2;

    P1 = Select();
    P2 = Select(P1);
    if (FRandom() <= ProbCross) {
        // ¿ Quien es el Padre y quien la Madre...?
        if ( FRandom() <= 0.5 )
            Poblacion[P1]->Cross(Poblacion[P2], PobTmp[0]);
        else
            Poblacion[P2]->Cross(Poblacion[P1], PobTmp[0]);
    } else {
            *PobTmp[0] = *Poblacion[P1];
    }

    PobTmp[0]->Mutate(ProbMutation);
    PobTmp[0]->SetPhenotype();

    // Evaluación del nuevo individuo.
    PobTmp[0]->SetFitness();

    Replacement(0);

    Generation++;
    return ;
}

// Reemplaza el individuo n de la poblacion temporal por el
// peor individuo (mayor fitness) de poblacion.
void Genitor::Replacement(int n) {
    int i = GetMaxFitness();
    Swap(PobTmp[n],Poblacion[i]);
}
```

Nótese la reutilización de gran parte del código de la clase base GA y la sencillez con la que se puede derivar una nueva variante, modificando una cantidad mínima de código.

4.- Ejemplo de Utilización

Ahora, se va desarrollar un pequeño ejemplo en el que se va a mostrar como se puede abordar la optimización de una función utilizando tanto poblaciones con GA como con Genitor y con los dos tipos de individuos descritos anteriormente.

Para esto, el proceso comienza con la especificación de la función a minimizar, la cual se dotará del interface adecuado para incorporarla en los distintos individuos. A continuación, se pasa a declarar dos individuos, uno de la clase Real y otro de la clase Binary, los cuales van a ser la semilla de sus correspondientes poblaciones. El siguiente paso es declarar ambas poblaciones, asignando por ejemplo a la población de GA el individuo de tipo Real y a la de Genitor el individuo de tipo Binary con sus correspondientes censos. Y, una vez en este punto, se comienza con el bucle de iteracción hasta que se alcanze un determinado número de generaciones.

El código que refleja todo este proceso sería el siguiente:

```
#ifndef _BORLANDC_
#ifdef _DOS_
#include <gppconio.h>
#endif
#endif

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "random.h"
#include "binary.h"
#include "ga.h"
#include "real.h"
#include "genitor.h"
```

```
// Número de individuos de las poblaciones.
#define CENSO           50

// Intervalo de búsqueda.
#define A          -30.0
#define B          50.0

// Macro de expansión de los fenotipos a un valor válido
  dentro del área
// [A,B]x[A,B]
#define EXPAND(a) ((a)*((float)B-(float)A)+(float)A)

// Número de variables de la función de optimización y que en
  este
// ejemplo se corresponde con el número de genes de los
  individuos.
#define NUM_VARIABLES  2

// Función a MINIMIZAR:
//    Esta toma como parametro un vector con dos elementos de
  tipo float (x,y)
//    y retorna la siguiente expresion:
//       100*(x^2-y^2)^2+(1-x)^2
//    cuyo valor optimo está en el punto (1,1) con un valor de
        cero
float            fdj2(float *p)
{
   float            sum,
                    sum1;

    sum = (p[0] * p[0] - p[1] * p[1]);
    sum *= sum;
    sum1 = 1.0 - p[0];
    sum1 *= sum1;
    sum = ((float) 100.0) * sum + sum1;
    return (sum);
}

// Función de bondad para un individuo de la clase Real.
float            FDJ2_R(Real & I)
```

```
{
    float           p[NUM_VARIABLES];
    float aux;

    p[0] = EXPAND(I.Phenotype[0]);
    p[1] = EXPAND(I.Phenotype[1]);
    aux = fdj2(p);

    return (aux);
}

// Función de bondad para un individuo de la clase Binary.
float           FDJ2(Binary & I)
{
    float           p[NUM_VARIABLES];

    p[0] = EXPAND(I.Phenotype[0]);
    p[1] = EXPAND(I.Phenotype[1]);
    return (fdj2(p));
}

main(int nargs, char *args[])
{
    FSrandom();
    SRandom((int) time(NULL));

    Real            RealIndiv(NUM_VARIABLES, FDJ2_R, B, A);
    Binary          BinIndiv(NUM_VARIABLES,FDJ2);

    // Población con GA tradicional y con individuos Real.
    GA      PobGA_Real(&RealIndiv,CENSO);

    // Población con Genitor y con individuos Binary.
    Genitor PobGenitor_Bin(&BinIndiv, CENSO);

    int             i,
                    iters,
                    s;
```

```
        // Número de generaciones.
        if (nargs > 1)
            iters = atoi(args[1]);
        else
            iters = 10;

        // Frecuencia de visualización.
        if (nargs > 2)
            s = atoi(args[2]);
        else
            s = 2;
        // Inicialización aleatoria de las poblaciones.
        PobGA_Real.Reset();
        PobGenitor_Bin.Reset();

        // Asignación de las probabilidades de mutación
        PobGenitor_Bin.SetProbMutation(0.09);
        PobGA_Real.SetProbMutation(0.09);

        // Bucle principal.
        printf("\t          Real        Binary        \n");
        for (i = 0; i < iters; i++) {
            // Ejecución de una iteración.
            PobGA_Real.OneGeneration();
            PobGenitor_Bin.OneGeneration();

            // ¿ Hay que mostrala ?
            if (!(i % s))
                // Visualización de la generación, menor fitness e
                    indice del
                // mejor individuo de cada población.
                printf("\t[%6d] %g[%d] | %g[%d] \n", i,

PobGA_Real[PobGA_Real.GetMinFitness()].GetFitness(),
                      PobGA_Real.GetMinFitness(),

PobGenitor_Bin[PobGenitor_Bin.GetMinFitness()].GetFitness(),
                      PobGenitor_Bin.GetMinFitness()
```

```
                    );
        }

        printf("___________________________________\n");

        // Visualización de los mejores individuos.
        PobGA_Real[PobGA_Real.GetMinFitness()].Show();
        PobGenitor_Bin[PobGenitor_Bin.GetMinFitness()].Show();

    return (0);
    }
```

Referencias

[1] Booch, G., (1991)."Object Oriented Design With Applications", The Benjamin/Cummings Publishing Company Inc.

[2] Whitley, D., Kauth, J. (1988). “Genitor: A different Genetic Algotihm”. Technical Report CS-88-101, Colorado State University.

[3] Syswerda, G. “Uniform Crossover in Genetic Algrithms”. Proceedings of the Third Int´l Conference on Genetic Algorithms and ther Applications”, 2-9, H. D. Schaffer, ed., Morgan Kaufmann, June 1989.

[4] Holland, J. H. (1975). “Adaptation in Natural and Artificial Systems”. University of Michigan Press.