Universidad de Murcia

Material didáctico para la asignatura Sistemas Inteligentes de 3º de Grado en Informática

AUTORES:

- José Manuel Cadenas Figueredo
- María del Carmen Garrido Carrera
- Raquel Martínez España
- Santiago Paredes Moreno

Capítulo 6

Clases GAGenome

6.1 Introducción

La elección de una buena representación de los individuos en un algoritmo genético es un factor clave para asegurar el éxito de la solución. La representación elegida debe representar soluciones factibles del problema. En caso de que la representación de un individuo pueda generar soluciones no factibles, la función objetivo debe ser diseñada para detectar aquellas soluciones que no sean factibles y penalizarlas de alguna manera.

La librería GAlib ofrece una serie de clases para representar los individuos. Todas estas clases heredan de la clase `GAGenome`. Esta es la clase principal a partir de la cual se puede elegir la estructura de datos que puede tener un cromosoma y no puede ser instanciada. Las clases que definen las diferentes representaciones que la librería GAlib ofrece se muestran en la Figura 6.1.

En general, podemos decir que los cuatro tipos principales de representación de los cromosomas son:

- `GAArrayGenome`: Esta clase se utiliza para representar un cromosoma mediante un array de objetos que puede representarse con o sin alelos.
- `GABinaryStringGenome`: Esta clase es utilizada para representar un cromosoma de forma binaria.
- `GAListGenome`: Esta clase es para crear un cromosoma donde es necesario tener cierto orden y la longitud de los cromosomas puede ser variable.

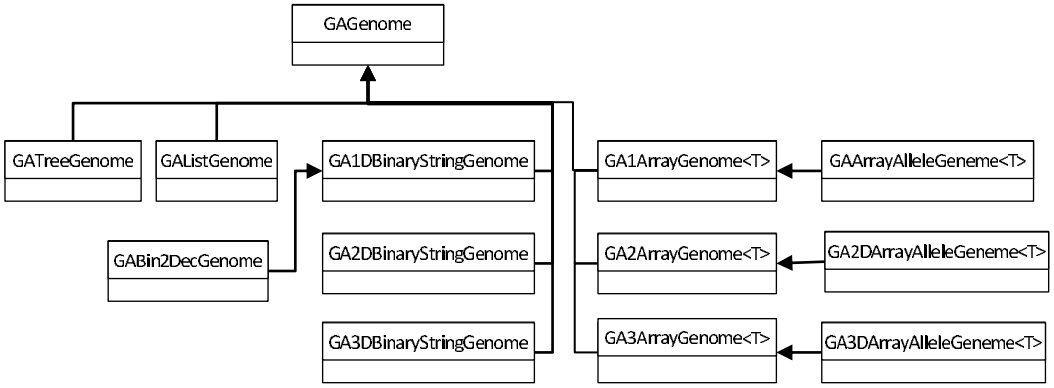

Figura 6.1: Clases para representar los cromosomas

- `GATreeGenome`: Esta clase es para representar un cromosoma utilizando la estructura de datos de un árbol.

En los siguientes apartados vamos a estudiar un poco más en detalle los métodos que tienen en común las diferentes clases que definen las representaciones de individuos disponibles en la librería GAlib y además vamos a estudiar en profundidad algunas de estas representaciones, en concreto, las clases `GABinaryStringGenome` y `GAArrayGenome` para dimensión 1.

6.2 Métodos principales

Sin tener en cuenta el tipo de representación elegida para resolver un problema, hay algunos métodos que son comunes a todas las representaciones. Entre ellos podemos destacar:

- `GAGenomeInitializer initializer() const` / `GAGenomeInitializer initializer(GAGenome::Initializer func)`: Este método devuelve/establece el operador de inicialización.

- `GAGenome::Mutator mutator() const` / `GAGenome::Mutator mutator (GAGenome::Mutator func)`: El método devuelve/establece el operador de mutación que se va a aplicar durante el algoritmo genético.

- `GAGenome::SexualCrossover crossover(GAGenome::SexualCrossover f) / GAGenome::AsexualCrossover crossover(GAGenome::AsexualCrossover f)`: Este método establece el operador genético de cruce para un individuo.

- `void * userData() / void * userData(void * data)`: Cada individuo tiene la posibilidad de contener un puntero genérico a los datos que son necesarios asociar al mismo. Este método devuelve/establece dicho puntero. Sin embargo, hay que tener en cuenta que cuando se clona un individuo, los datos asociados no se clonan por lo que la referencia es la misma.

- `int length() const / int length(int l)`: Este método devuelve/establece la longitud del individuo.

Como acabamos de describir, los métodos `initializer`, `mutator` y `crossover` permiten cambiar el operador genético por defecto correspondiente. La librería GAlib asigna operadores por defecto para cada clase `GAGenome` y usando los métodos anteriores, estos valores por defecto pueden ser modificados por otros definidos en la librería o incluso por operadores definidos por el usuario.

Para definir un operador, el usuario debe de implementar dicho operador en una función con las siguientes signaturas:

- `void operadorInicio(GAGenome &g)` para el operador de inicio, donde el parámetro `g` es el genoma que se inicializa. Además, habrá que indicar que se quiere usar dicho operador usando el método `initializer`, `genoma.initializer(operadorInicio);`, donde `genoma` es un individuo del tipo que se usa en el problema que se está resolviendo.

- `int operadorMutacion(GAGenome &g,float pmut)` para el operador de mutación, donde `g` es el genoma a mutar y `pmut` es la probabilidad de mutación. Además habrá que indicar que se quiere usar dicho operador, `genoma.mutator(operadorMutacion);`.

- `int operadorCruce(const GAGenome& p1,const GAGenome& p2, GAGenome* c1,GAGenome* c2)` para el operador de cruce, donde `p1` y `p2` son los genomas a cruzar (progenitores) y `c1` y `c2`, los individuos generados. Además habrá que indicar que se quiere usar dicho operador, `genoma.crossover(operadorCruce);`.

6.3 Clase GA1DBinaryStringGenome

La clase `GA1DBinaryStringGenome` es una clase derivada de las clases `GABinaryString` y `GAGenome`. Esta clase representa al individuo como un string de ceros y unos. Los genes para este individuo son bits y los alelos para cada bit son 0 y 1.

6.3.1 Constructor de clase

Para crear un objeto de esta clase se debe usar uno de los siguientes constructores:

- `GA1DBinaryStringGenome(unsigned int x, GAGenome:: Evaluator objective = NULL, void *userData = NULL)`: Este constructor toma como argumentos la longitud del individuo, la función objetivo y se puede especificar datos asociados al individuo si se considera necesario. Si no se quieren especificar estos datos se debe de indicar el valor `NULL`.
- `GA1DBinaryStringGenome(const GA1DBinaryStringGenome&)`: Con este constructor se crea un individuo como copia del pasado como parámetro.

6.3.2 Operadores genéticos por defecto y disponibles

Los operadores genéticos disponibles en la librería para este tipo de individuos son los siguientes:

- `GA1DBinaryStringGenome::UniformInitializer`: Operador de inicio que inicializa el genoma siguiendo una distribución uniforme.
- `GA1DBinaryStringGenome::SetInitializer`: Operador de inicio que inicializa con unos.
- `GA1DBinaryStringGenome::UnsetInitializer`: Operador de inicio que inicializa con ceros.
- `GA1DBinaryStringGenome::FlipMutator`: Operador de mutación que cambia el valor de un elemento del genoma a otro de sus posibles valores.
- `GA1DBinaryStringGenome::UniformCrossover`:Operador de cruce que compara los genes de los padres y los intercambia con cierta probabilidad si son distintos.

- `GA1DBinaryStringGenome::EvenOddCrossover`: Operador de cruce en el que uno de los hijos hereda los genes pares de los padres y otro los impares.
- `GA1DBinaryStringGenome::OnePointCrossover`: Operador de cruce por un punto.
- `GA1DBinaryStringGenome::TwoPointCrossover`: Operador de cruce por dos puntos.

Por defecto, esta representación de los individuos tiene los siguientes operadores genéticos:

- El operador de inicio por defecto es `GA1DBinaryStringGenome::UniformInitializer`.
- El operador de mutación por defecto es `GA1DBinaryStringGenome::FlipMutator`.
- El operador de cruce por defecto es `GA1DBinaryStringGenome:: OnePointCrossover`.

Si se quisiera cambiar el operador por defecto, usaremos los métodos `initializer`, `mutator` y `crossover` comentados en la Sección 6.2. Por ejemplo, si quisiéramos cambiar el operador de cruce a `GA1DBinaryStringGenome ::TwoPointCross over`, usaríamos el siguiente código:

```
1 genome.crossover(GA1DBinaryStringGenome::TwoPointCrossover);
```

Algunos de los métodos más comunes de esta representación de individuos son los siguientes:

- `void copy(const GA1DBinaryStringGenome & original,unsigned int dest, unsigned int src,unsigned int length)`: copia desde la posición `src`, `length` bits desde el individuo `original` al individuo al que se aplica el método.
- `short gene(unsigned int x=0) const /short gene(unsigned int, short value)`: devuelve/establece el valor de un gen.

6.3.3 Representación para el problema de minimizar una función con restricciones

Para resolver este problema donde debemos encontrar los valores X_1 y X_2 que minimizan la función $Y = X_1^2 - 2 \cdot X_1 + X_2^2 + 2$, usaremos un individuo del tipo `GA1DBinaryString Genome` de tamaño 32 bits. Usaremos 16 bits para obtener el valor de X_1 y otros 16 bits para obtener el valor de X_2. Cuanto mayor sea la cantidad de bits asignados a cada variable mayor será la precisión de las mismas.

Dado que el mayor valor decimal que se puede representar con 16 bits es 65535, a partir de los 16 primeros bits del invididuo obtenemos el valor de X_1 (obtenemos el fenotipo desde el genotipo) como:

$$0 + \left(\frac{decimalPrimeros16Bits}{65535} \right) \times (5 - 0)$$

donde $decimalPrimeros16Bits$ es el valor decimal representado en los 16 primeros bits del individuo y mediante la anterior expresión es trasladado al intervalo $[0, 5]$ al que debe pertenecer X_1. Lo mismo haremos para obtener el valor de X_2 pero en este caso a partir de los últimos 16 bits del individuo:

$$0 + \left(\frac{decimalUltimos16Bits}{65535} \right) \times (5 - 0)$$

De esta forma la función que obtendría el fenotipo a partir del genotipo de un individuo, es decir, el valor de X_1 y X_2, sería:

```
1  float * decodificar(GAGenome & g){
2
3    GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &)g;
4
5    float * vect = new float[2];
6
7  // del gen 0 al 15 representa el valor de X1
8    float parte1 = 0;
9    for(int i=0; i<16; i++)
10           parte1+=(genome.gene(i)* pow(2.0,(float)15-i));
11   float X1 = 0 + (parte1/65535.0) * (5-0);
12
13 // del gen 16 al 31 representa el valor de X2
14   float parte2 = 0;
```

```
15    for(int i=16; i<32; i++)
16            parte2+=(genome.gene(i)* pow(2.0,(float)31-i));
17    float X2 = 0 + (parte2/65535.0) * (5-0);
18
19    vect[0] = X1;
20    vect[1] = X2;
21
22    return vect;
23  }
```

El código que crearía un individuo con esta estructura es el que se muestra a continuación:

```
1  int main(int argc, char **argv){
2
3    cout << "Este programa encuentra el valor minimo en la funcion\n";
4    cout << "  y = x1^2 - 2x^1 + x2^2 + 2\n";
5    cout << "con las restricciones\n";
6    cout << "      0 <= x1 <= 5\n";
7    cout << "      0 <= x2 <= 5\n";
8    cout << "\n\n";
9    cout.flush();
10
11   ...
12
13 // El genoma contiene 16 bits para representar el valor de X1 y
14 // 16 bits para representar el valor de X2
15
16    int tam = 32;
17
18 // del gen 0 al 15 representa el valor de X1
19 // del gen 16 al 31 representa el valor de X2
20
21 // Creamos el individuo que es usado por el GA para clonarlo y crear una
       población de genomas.
22
23   GA1DBinaryStringGenome genome(tam, Objective,NULL);
24
25   ...
26 }
```

Como se observa en el ejemplo, el individuo se crea mediante el primer constructor

comentado, indicando el tamaño que en este caso es un tamaño fijo de 32 bits de longitud, la función objetivo y, como en este caso no es necesario añadir datos al individuo, el último parámetro es `NULL`.

6.4 Clase GA1DArrayGenome<T>

Este tipo de representación de individuos consiste en un array genérico de una dimensión de tamaño ampliable. La clase `GA1DArrayGenome<T>` es una clase plantilla que deriva de la clase principal `GAGenome`. Cada elemento del array representa un gen y el valor de cada gen es determinado por el tipo <T> que se especifique. Por ejemplo, si se especifica `<int>`, cada gen representará un número entero, si por el contrario se especifica `<double>` cada gen representará un valor real.

6.4.1 Constructor de clase

Para construir un individuo utilizando este tipo de representación, la clase dispone de los siguientes constructores:

- `GA1DArrayGenome(unsigned int length, GAGenome::Evaluator objective = NULL, void * userData = NULL)`: Los argumentos del constructor son la longitud del array, la función objetivo y los datos adicionales. Si algunos de los dos últimos argumentos no se quieren indicar se deben de poner a NULL.
- `GA1DArrayGenome(const GA1DArrayGenome<T> &)`: Este constructor crea un individuo como copia del pasado como parámetro.

6.4.2 Operadores genéticos por defecto y disponibles

Los operadores genéticos que se encuentran disponibles para ser utilizados por este tipo de representación son los siguientes:

- `GA1DArrayGenome<>::SwapMutator`: Operador de mutación que intercambia dos genes del individuo.
- `GA1DArrayGenome<>::UniformCrossover`: Operador de cruce que compara los genes de los padres y los intercambia con cierta probabilidad si son distintos.

- `GA1DArrayGenome<>::EvenOddCrossover`: Operador de cruce en el que uno de los hijos hereda los genes pares de los padres y otro los impares.
- `GA1DArrayGenome<>::OnePointCrossover`: Operador de cruce por un punto.
- `GA1DArrayGenome<>::TwoPointCrossover`: Operador de cruce por dos puntos.
- `GA1DArrayGenome<>::PartialMatchCrossover`: Operador de cruce en el que dados dos puntos, se intercambia la parte intermedia de los dos padres.
- `GA1DArrayGenome<>::OrderCrossover`: Operador de cruce usado cuando los genomas son listas ordenadas. Se selecciona un punto de cruce, y cada hijo toma la primera parte de un padre y la segunda parte del hijo se ordena según el orden del otro padre.

Los operadores genéticos establecidos por defecto para esta representación son:

- Para inicializar el individuo se utiliza por defecto el operador `GAGenome::NoInitializer`.
- El operador de mutación establecido por defecto es `GA1DArrayGenome<>::SwapMutator`.
- El operador de cruce definido por defecto es `GA1DArrayGenome<>::OnePointCrossover`.

Por último, comentar que algunos de los métodos que pueden ser utilizados con más frecuencia cuando se hace uso de este tipo de representación de individuos son los siguientes:

- `void copy(const GA1DArrayGenome<T>& original,unsigned int dest, unsigned int src,unsigned int length)`: copia desde la posición `src`, `length` bits desde el individuo `original` al individuo al que se aplica el método.
- `T & gene(unsigned int x=0) const / T & gene(unsigned int, const T& value) const`: que devuelve/establece el valor de un gen.

6.5 Clase GA1DArrayAlleleGenome<T>

La clase `GA1DArrayAlleleGenome<T>` es derivada de la clase comentada anteriormente, `GA1DArrayGenome<T>`. Comparte el mismo comportamiento y añade la característica de asociar a los genes del individuo un conjunto de alelos. El valor que puede tomar cada elemento en un individuo con este tipo de representación depende del conjunto de alelos que define los posibles valores.

Si se crea un individuo con un único conjunto de alelos, el individuo tendrá la longitud que especifique el usuario y el conjunto de alelos será utilizado como posibles valores para cada gen. Si se crea el individuo utilizando un array de conjuntos de alelos, el individuo tendrá una longitud igual al número de conjuntos de alelos del array y a cada gen se le asignará un valor del conjunto de alelos correspondiente.

6.5.1 Constructor de clase

Para crear un individuo utilizando esta representación, la clase ofrece los siguientes constructores:

- `GA1DArrayAlleleGenome(unsigned int length, const GAAlleleSet <T>& alleleset, GAGenome::Evaluator objective = NULL, void * userData = NULL)`: Este constructor toma como parámetro la longitud del individuo, un conjunto de alelos, la función objetivo y los datos adicionales.

- `GA1DArrayAlleleGenome(const GAAlleleSetArray<T>& allelesets, GAGenome::Evaluator objective = NULL, void * userData = NULL)`: El constructor toma como argumentos el array de conjuntos de alelos, la función objetivo y los datos adicionales. En este constructor no se especifica la longitud del individuo.

- `GA1DArrayAlleleGenome(const GA1DArrayAlleleGenome<T>&)`: Este método crea un individuo como copia del individuo que se pasa como parámetro.

6.5.2 Definición de los alelos: Clase GAAlleleSet<T>

Antes de crear un individuo de este tipo, es necesario definir los conjuntos de alelos que definirán los valores de sus genes. Para ello, creamos una instancia de la clase

`GAAlleleSet<T>` y añadiremos los distintos valores al conjunto usando el método siguiente:

- `T add(const T& allele)`: El método añade el alelo `allele` al conjunto.

6.5.3 Operadores genéticos por defecto y disponibles

Al utilizar esta representación, los operadores genéticos que hay disponibles, además de los heredados de la clase `GA1DArrayGenome<T>` son:

- `GA1DArrayAlleleGenome<>::UniformInitializer`: Operador de inicio que inicializa el genoma siguiendo una distribución uniforme.
- `GA1DArrayAlleleGenome<>::OrderedInitializer`: Operador de inicio que tiene en cuenta que el genoma es una lista ordenada.
- `GA1DArrayAlleleGenome<>::FlipMutator`: Operador de mutación que cambia el valor de un elemento del genoma a otro de sus posibles valores.

Los operadores por defecto son:

- Como operador de inicialización el operador `GA1DArrayAlleleGenome<>::UniformInitializer`.
- El operador de mutación establecido por defecto es `GA1DArrayAlleleGenome<>::FlipMutator`.
- Como operador de cruce, el establecido por defecto es `GA1DArrayGenome<>::OnePointCrossover`.

6.5.4 Representación para el problema de las N reinas

Usaremos este tipo de representación para resolver el problema de las N reinas. Para ello supondremos que cada gen del individuo representa una columna del tablero de ajedrez y su valor indica la fila en la que se encuentra situada la reina de esa columna (cualquier solución al problema no tendrá más de una reina por columna). El valor por lo tanto de cada gen estará limitado al número de filas del tablero y por lo tanto podemos construir un conjunto de alelos formado por los valores enteros comprendidos

en el intervalo [0,numeroFilas-1]. Por lo tanto, el individuo será un array de 1 dimensión de valores enteros con una longitud igual al número de columnas del tablero y donde cada gen toma valores del conjunto de alelos comentado.

```
1  int main(int argc, char **argv)
2  {
3
4      int nreinas = 8;
5
6      cout << "Problema de las " << nreinas << " reinas \n\n";
7      cout.flush();
8
9      ...
10
11     // Conjunto enumerado de alelos --> valores posibles de cada gen del
           genoma
12
13     GAAlleleSet<int> alelos;
14     for(int i=0;i<nreinas;i++) alelos.add(i);
15
16     // Creamos el individuo pasándole como argumentos la longitud, el
           conjunto de alelos y la referencia a la función objetivo
17
18     GA1DArrayAlleleGenome<int> genome(nreinas,alelos,Objective,NULL);
19
20     ...
21 }
```

6.6 Función Objetivo/Fitness

Como ya hemos comentado, la librería GAlib nos ofrece un conjunto de métodos y clases para poder resolver un problema mediante un algoritmo genético. Utilizando dichos métodos y clases se puede implementar un algoritmo genético casi completo. La única función que como mínimo debe definir el usuario es la función objetivo o función fitness. Esta función obtiene una medida que indica la bondad de un individuo frente al resto. Dicha medida es la utilizada por el operador de selección para discriminar entre individuos.

La función objetivo que el usuario debe definir toma como argumento la referencia del individuo a evaluar y devuelve un valor que indica cómo de bueno o malo es dicho

individuo. La signatura de esta función debe ser la siguiente:

```
1 float Objective(GAGenome &) {
2    // Aquí se debe de detallar el código de la función objetivo
3 }
```

Como se puede ver en este código, la función recibe como argumento un individuo genérico y por lo tanto para poder trabajar con dicho individuo dentro de la función, será necesario llevar a cabo el casting al tipo concreto de individuo que se esté utilizando. Por ejemplo, si el individuo es del tipo binario, la función comenzaría de la siguiente forma:

```
1 float Objective(GAGenome & g) {
2   // Se realiza el casting correspondiente
3   GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &)g;
4   ...
5 }
```

6.6.1 Ejemplos de funciones objetivo

Para clarificar la definición de esta función objetivo vamos a mostrar una función objetivo posible para los dos primeros problemas planteados: minizar una función y N reinas.

Función objetivo para el problema de minimizar una función

Para el problema de minimizar la función y usando la representación del individuo comentada en la Sección 6.3.3, la medida de la bondad o fitness de un individuo es simplemente el valor Y obtenido al aplicar el punto $X1$ y $X2$ representado en el individuo a la función que estamos minimizando. Sabemos que analíticamente, el valor óptimo lo proporcionará una solución con fitness 1, es decir, $Y = 1$.

El método `float * decodificar(GAGenome & g)` es el descrito en la Sección 6.3.3. Como comentamos, este método se encarga de obtener el fenotipo del individuo `g`, es decir, los valores decimales $X1$ y $X2$ representados a partir de dicho individuo.

```
1 float Objective(GAGenome & g) {
2   // Se realiza el casting correspondiente
3   GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &)g;
```

```
4
5   // Se devuelve el valor de Y
6   float * vect, X1, X2;
7   vect = decodificar(genome);
8   X1 = vect[0];
9   X2 = vect[1];
10  delete [] vect;
11  float Y =(X1*X1) - (2*X1) + (X2*X2) + 2;
12  return Y;
13 }
```

Función objetivo para el problema de las N reinas

Como segundo ejemplo vamos a mostrar una función objetivo para el problema de las N reinas, partiendo de la representación comentada en la Sección 6.5.4. En este problema, una posible medida de la bondad de una solución o un cromosoma es el número de jaques que se producen en la misma. Por lo tanto, el valor óptimo lo proporcionará una solución con fitness 0, es decir, con un número de jaques igual a 0.

```
1  float Objective(GAGenome & g) {
2    // Se realiza el casting correspondiente
3    GA1DArrayAlleleGenome<int> & genome = (GA1DArrayAlleleGenome<int> &)g;
4    float jaques=0;
5    int c,f;
6
7    //Si hay un repetido en vector es un jaque de misma fila
8    for(int i=0; i<genome.length(); i++)
9            for(int j=i+1;j<genome.length();j++)
10                   if (genome.gene(i)==genome.gene(j)) jaques++;
11
12   //Si diagonal también es jaque
13   for(int en_est=0;en_est<genome.length();en_est++){
14
15     //Diagonal derecha abajo
16     c=en_est+1;
17     f=genome.gene(en_est)+1;
18     while ((c<genome.length()) && (f<genome.length())){
19             if (genome.gene(c)==f) jaques++;
20             c++;
21             f++;
22     }
```

```
23      //Diagonal derecha arriba
24      c=en_est+1;
25      f=genome.gene(en_est)-1;
26      while ((c<genome.length()) && (f>=0)){
27              if (genome.gene(c)==f) jaques++;
28              c++;
29              f--;
30      }
31    }
32
33    return jaques;
34  }
```