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 7

Definición del algoritmo genético

7.1 Introducción

La clase que permite crear una instancia de un algoritmo genético es `GAGeneticAlgorithm`. Es una clase abstracta y no puede ser instanciada. Para crear una instancia de un algoritmo genético hay que crear una instancia de las diferentes clases derivadas que representan los diferentes tipos de algoritmos genéticos que la librería GAlib ofrece. Estas clases son las siguientes:

- La clase derivada `GASimpleGA` representa el tipo de algoritmo genético más sencillo y lo ampliaremos en los siguientes apartados. Su característica principal es que crea una población totalmente nueva en cada generación.
- La clase `GADemeGA` representa un tipo de algoritmo genético con la particularidad de permitir la ejecución de varias poblaciones paralelas permitiendo la migración de individuos entre ellas.
- El tipo de algoritmo genético representado en la clase `GAIncrementalGA` tiene la particularidad de permitir un pequeño solapamiento de las poblaciones generadas (con individuos comunes) con un reemplazamiento personalizado por el usuario. En cada generación se producen varios hijos (normalmente 1 ó 2) que deben sustituir a otros individuos de la población.
- La clase derivada `GASteadyStateGA` representa un algoritmo que produce poblaciones solapadas, permitiendo especificar qué proporción de la población debe ser

sustituida en cada generación.

7.2 La clase GASimpleGA

Como comentamos anteriormente nos vamos a centrar en el algoritmo genético que está implementado en la clase `GASimpleGA`. Este algoritmo genético es el que se describe en [4]. Las poblaciones en este algoritmo no se superponen, es decir, se crea una población totalmente nueva en cada generación. El esquema básico del funcionamiento de este tipo de algoritmo es el siguiente:

```
1  función Algoritmo Genético
2       t = 0
3       inicializar P(t)
4       evaluar P(t)
5       mientras (no condición de parada) hacer
6                t = t + 1
7                seleccionar P(t) desde P(t-1)
8                cruzar P(t)
9                mutación P(t)
10               evaluar P(t)
```

De esta forma, al crear un algoritmo de este tipo usando la librería GAlib, se debe de especificar un individuo o una población de individuos. El algoritmo genético clonará el/los individuos que se hayan especificado para crear la población inicial. En cada generación el algoritmo crea una nueva población de individuos aplicando una selección de la población anterior, aplicando los operadores genéticos (cruce y mutación) para crear la población descendiente de la nueva población. El proceso continúa hasta que se cumple el criterio de terminación (este criterio se determina mediante el objeto `Terminator`).

La librería permite opcionalmente que este tipo de algoritmo sea elitista. El elitismo permite que el mejor individuo de cada generación pase automáticamente a formar parte de la siguiente generación. Por defecto, el elitismo está activado. Para desactivar el elitismo se utiliza el método `GABoolean elitist(GABoolean flag)` indicando como parámetro el valor `gaFalse`.

7.2.1 Constructor de clase

Para crear un algoritmo genético simple, la clase proporciona los tres constructores de clase siguientes:

- `GASimpleGA(const GAGenome &)`: Crea la población inicial de individuos a partir del individuo que se le pasa como parámetro.
- `GASimpleGA(const GAPopulation &)`: Crea la población inicial de individuos copiando la población que se pasa como parámetro.
- `GASimpleGA(const GASimpleGA &)`: Crea el algoritmo genético como copia del que se le pasa como parámetro.

7.2.2 Métodos principales

Además de los constructores de clase, el algoritmo genético dispone de una serie de métodos para establecer los diferentes parámetros del algoritmo. A continuación se detallan los más importantes:

- `void evolve(unsigned int seed=0)`: Este método inicia la evolución del algoritmo genético hasta que se cumpla el criterio de parada. Como parámetro se pasa una semilla para la generación de los valores aleatorios necesarios.
- `const GAStatistics& statistics() const`: El método devuelve una referencia al objeto de estadísticas del algoritmo genético. El objeto de estadísticas mantiene diversa información acerca de la ejecución del algoritmo y será comentado más adelante.
- `int generation() const`: Este método devuelve el número de generación actual.
- `int minimaxi() const / int minimaxi(int)`: Este método devuelve / define si el objetivo del algoritmo genético consiste en minimizar o maximizar. El valor a indicar cuando el problema sea de minimizar será de -1 y para maximizar el valor a indicar como parámetro es 1. Por defecto, el valor establecido es maximizar.

- `int nGenerations() const / int nGenerations(unsigned int)`: El método devuelve / establece el número de generaciones máxima que el algoritmo debe realizar.

- `float pMutation() const / float pMutation(float)`: Este método devuelve / define la probabilidad de mutación.

- `float pCrossover() const / float pCrossover(float)`: Este método devuelve / define la probabilidad de cruce.

- `int populationSize() const / int populationSize(unsigned int)`: Este método devuelve / define el tamaño de la población. Dicho tamaño puede ser modificado durante la evolución del algoritmo genético.

- `GASelectionScheme& selector() const / GASelectionScheme& selector( const GASelectionScheme& s)`: Este método devuelve / define el operador de selección. Los esquemas posibles proporcionados por la librería como clases son:

  - `GARouletteWheel`: Se asigna una probabilidad de selección proporcional al valor del fitness del cromosoma.
  - `GATournamentSelector`: Escoge al individuo de mejor fitness de entre Nts individuos seleccionados aleatoriamente con reemplazamiento ($Nts = 2, 3, ...$).
  - `GAUniformSelector`: Todos los individuos de la población tienen la misma probabilidad de ser seleccionados.
  - `GARankSelector`: Selecciona al individuo con mejor fitness.

  El operador por defecto es `RouletteWheel`. Un ejemplo que cambiaría el operador de selección por defecto de un algoritmo es el siguiente:

```
1 GATournamentSelector s; // declaramos un objeto del tipo selección por
      torneo
2 ga.selector(s); // se lo asignamos al algoritmo genético ga
```

7.2.3 Terminación del algoritmo genético

Los métodos `GAGeneticAlgorithm::Terminator terminator()` y `GAGeneticAlgorithm:: Terminator terminator(GAGeneticAlgorithm ::Terminator)` devuelve / define el criterio de terminación del algoritmo genético. El criterio de terminación por defecto en un algoritmo genético del tipo `GASimpleGA` es el número de generaciones.

El usuario puede definir otros criterios de terminación y asignárselos al algoritmo genético usando el método `terminator` comentado anteriormente:

```
1    ...
2    GASimpleGA ga;
3    ga.terminator(FuncionTerminacion);
4    ...
5 GABoolean FuncionTerminacion(GAGeneticAlgorithm &){
6   // código que define el criterio de parada
7 }
```

donde podemos ver que la signatura de una función de terminación debe ser la mostrada en el código anterior: un parámetro de entrada que es el objeto algoritmo genético sobre el que se define la función de terminación y devuelve un valor GABoolean (GATrue o GAFalse) según se cumpla o no el criterio de parada respectivamente.

Por último, a modo de ejemplo mostramos a continuación la parte inicial del problema de minimizar una función. De esta forma veremos cómo se crea el algoritmo, cómo se establecen los parámetros y cómo el algoritmo comienza a evolucionar.

```
1  ...
2  int main(int argc, char **argv){
3
4  // Especificamos una semilla aleatoria.
5    GARandomSeed((unsigned int)atoi(argv[1]));
6
7  // Declaramos variables para los parámetros del GA e inicializamos algunos
       valores
8    int popsize  = 200;
9    int ngen     = 5000;
10   float pmut   = 0.01;
11   float pcross = 0.6;
12   int tam = 32;
13   GA1DBinaryStringGenome genome(tam, Objective ,NULL);
```

```
14    GASimpleGA ga(genome);
15    ga.minimaxi(-1); // minimizamos
16    ga.populationSize(popsize);
17    ga.nGenerations(ngen);
18    ga.pMutation(pmut);
19    ga.pCrossover(pcross);
20    ga.evolve();
21    ...
22  }
23  ...
```

7.3 Impresión de valores de la ejecución

El diseño de la solución de un problema utilizando un algoritmo genético conlleva el estudio y análisis de diversos parámetros. Un aspecto interesante que ofrece la librería GAlib es un conjunto de métodos que muestran diferentes estadísticas acerca de una ejecución del algoritmo genético. Este conjunto de métodos se encuentran en la clase `GAStatistics`. Esta clase se encuentra accesible desde todos los tipos de algoritmos genéticos, ya que todos los objetos de tipo algoritmo genético tienen una función llamada `statistics()` que devuelve un objeto de tipo `GAStatistics`. Por lo tanto, se puede consultar las estadísticas de las ejecuciones sea cual sea el tipo de algoritmo genético utilizado para buscar la solución a un problema planteado. Vamos a analizar un poco más en detalle dicha clase.

7.3.1 Objeto GAStatistics

El objeto `GAStatistics` contiene información sobre el estado actual de un algoritmo genético. Cada objeto del tipo algoritmo genético en la librería, tiene asociado un objeto `GAStatistics` al que se puede acceder a través del método `statistics()`.

Cuando un objeto de tipo algoritmo genético accede al método `statistics()` (`ga.statistics()`, siendo `ga` un objeto de tipo algoritmo genético), el método devuelve una referencia al objeto `GAStatistics`. Este objeto mantiene siempre actualizada la información como el mejor individuo, el peor, la media y la desviación típica del fitness, etc.

Vamos a detallar cuales son los métodos principales proporcionados por esta clase para mostrar los diferentes valores de un algoritmo genético en ejecución y que puede ser muy útil a la hora de ajustar los diferentes parámetros del mismo.

7.3.2 Métodos principales

Entre los métodos principales para imprimir información de la ejecución de un algoritmo genético encontramos los siguientes:

- `float bestEver() const`: Retorna el fitness del mejor individuo encontrado hasta el momento.
- `float maxEver() const`: Devuelve el fitness máximo desde la inicialización.
- `float minEver() const`: Devuelve el fitness mínimo desde la inicialización.
- `const GAGenome& bestIndividual(unsigned int n=0) const`: Devuelve una referencia al mejor individuo encontrado por el algoritmo genético.
- `int crossovers() const`: Devuelve el número de cruces que se han llevado a cabo desde la inicialización.
- `int generation() const`: Devuelve el número de generación actual.
- `int mutations() const`: Devuelve el número de mutaciones que se han llevado a cabo desde la inicialización de la población.

Un ejemplo del uso de estas estadísticas lo encontramos en la definición de una función de terminación para el problema de las N reinas, donde establecemos que el algoritmo genético terminará al alcanzar el fitness óptimo (0 en este problema) o el número de generaciones especificado:

```
1 // Definimos la función de terminación para el problema de las $N$ reinas
2
3 GABoolean Termina(GAGeneticAlgorithm & ga){
4     if ((ga.statistics().minEver()==0)||(ga.statistics().generation()==ga.
          nGenerations())) return gaTrue;
5     else return gaFalse;
6 }
```

Pero también podemos utilizar estas estadísticas para estudiar si un determinado operador tiene un funcionamiento correcto mirando el número de veces que el operador ha sido efectivo. Un ejemplo de uso para este fin sería el siguiente:

```
1 int umbral = popsize/1000;
2
3 // Definimos una función de determina si el número de cruces realizados es
      menor que cierto umbral, predefinido anteriormente.
4
5 void OperadorCruceCorrecto(GAGeneticAlgorithm &ga){
6           if ((ga.statistics().crossovers() < umbral))
7              cout << "El operador de cruce realiza pocas operaciones" <<
                   endl;
8 }
```

7.4 Código completo para la optimización de una función

Una vez analizados los distintos elementos necesarios para la resolución de este problema usando un algoritmo genético implementado mediante la librería GAlib, mostramos el código completo del mismo:

```
1
2 /* ----------------------------------------------------------------
3   Optimización de una función
4
5   Programa que ilustra cómo usar un GA para encontrar el valor mínimo de
        una función continua en dos variables. Usa un genoma string binario
6
7 ---------------------------------------------------------------- */
8
9 #include <ga/GASimpleGA.h>  // usaremos un algoritmo genético simple
10 #include <ga/GA1DBinStrGenome.h> // y el genoma string binario de 1
       dimensión
11 #include <ga/std_stream.h>
12 #include <iostream>
13 #include <math.h>
14
15 using namespace std;
16
```

```
17 float Objective(GAGenome &);  // La función objetivo la definimos más
       adelante
18
19 float * decodificar(GAGenome &); // Función que obtiene el fenotipo de un
       genoma
20
21 int main(int argc, char **argv){
22
23   cout << "Este programa encuentra el valor minimo en la funcion\n";
24   cout << "  y = x1^2 - 2x^1 + x2^2 + 2\n";
25   cout << "con las restricciones\n";
26   cout << "      0 <= x1 <= 5\n";
27   cout << "      0 <= x2 <= 5\n";
28   cout << "\n\n";
29   cout.flush();
30
31 // Especificamos una semilla aleatoria.
32
33   GARandomSeed((unsigned int)atoi(argv[1]));
34
35 // Declaramos variables para los parámetros del GA e inicializamos algunos
       valores
36
37   int popsize  = 200;
38   int ngen     = 5000;
39   float pmut   = 0.01;
40   float pcross = 0.6;
41
42 // El genoma contiene 16 bits para representar el valor de X1 y 16 bits
       para representar el valor de X2
43
44    int tam = 32;
45
46 // del gen 0 al 15 representa el valor de X1
47 // del gen 16 al 31 representa el valor de X2
48
49 // Ahora creamos el GA y lo ejecutamos. Primero creamos el genoma que es
       usado por el GA para clonarlo y crear una población de genomas.
50
51   GA1DBinaryStringGenome genome(tam, Objective,NULL);
52
53 // Una vez creado el genoma, creamos el algoritmo genético e inicializamos
       sus parámetros - tamaño de la población, número de generaciones,
       probabilidad de mutación, y probabilidad de cruce. Finalmente, le
```

```
    indicamos que evolucione.
54
55   GASimpleGA ga(genome);
56   ga.minimaxi(-1); // minimizamos
57   ga.populationSize(popsize);
58   ga.nGenerations(ngen);
59   ga.pMutation(pmut);
60   ga.pCrossover(pcross);
61   ga.evolve();
62
63 // recuperamos el mejor individuo y obtenemos su fenotipo
64
65   GA1DBinaryStringGenome & mejor = (GA1DBinaryStringGenome &)ga.statistics
         ().bestIndividual();
66   float * vect = decodificar(mejor);
67
68 // Imprimimos el mejor genoma encontrado por el GA, su fitness y los
     valores X1 y X2
69
70   cout << "El GA encuentra:\n" << ga.statistics().bestIndividual() << "\n\n
         ";
71   cout << "Mejor valor fitness es y = " << ga.statistics().minEver() << "\n
         \n";
72   cout << "El punto encontrado es <" << vect[0] << "," << vect[1] << ">" <<
          endl;
73
74   delete [] vect;
75
76   system("pause");
77   return 0;
78 }
79
80 // decodifica el genoma
81
82 float * decodificar(GAGenome & g){
83
84   GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &)g;
85
86   float * vect = new float[2];
87
88 // del gen 0 al 15 representa el valor de X1
89   float parte1 = 0;
90   for(int i=0; i<16; i++)
91           parte1+=(genome.gene(i)* pow(2.0,(float)15-i));
```

```
92    float X1 = 0 + (parte1/65535.0) * (5-0);
93
94  // del gen 16 al 31 representa el valor de X2
95    float parte2 = 0;
96    for(int i=16; i<32; i++)
97            parte2+=(genome.gene(i)* pow(2.0,(float)31-i));
98    float X2 = 0 + (parte2/65535.0) * (5-0);
99
100   vect[0] = X1;
101   vect[1] = X2;
102
103   return vect;
104 }
105
106 // Definimos la función objetivo -- minimizar y = X1*X1 -2*X1 + X2*X2 + 2
107
108 float Objective(GAGenome& g) {
109   GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &)g;
110
111   // devolvemos el valor de Y
112
113   float * vect, X1, X2;
114
115   vect = decodificar(genome);
116   X1 = vect[0];
117   X2 = vect[1];
118
119   delete [] vect;
120
121   float Y =(X1*X1) - (2*X1) + (X2*X2) + 2;
122
123   return Y;
124 }
```

Cuando ejecutamos el programa obtenemos una salida como la mostrada en la Figura 7.1.

7.5 Código completo del ejemplo de las N reinas

El código completo que resuelve mediante un algoritmo genético el problema de las N reinas es el siguiente:

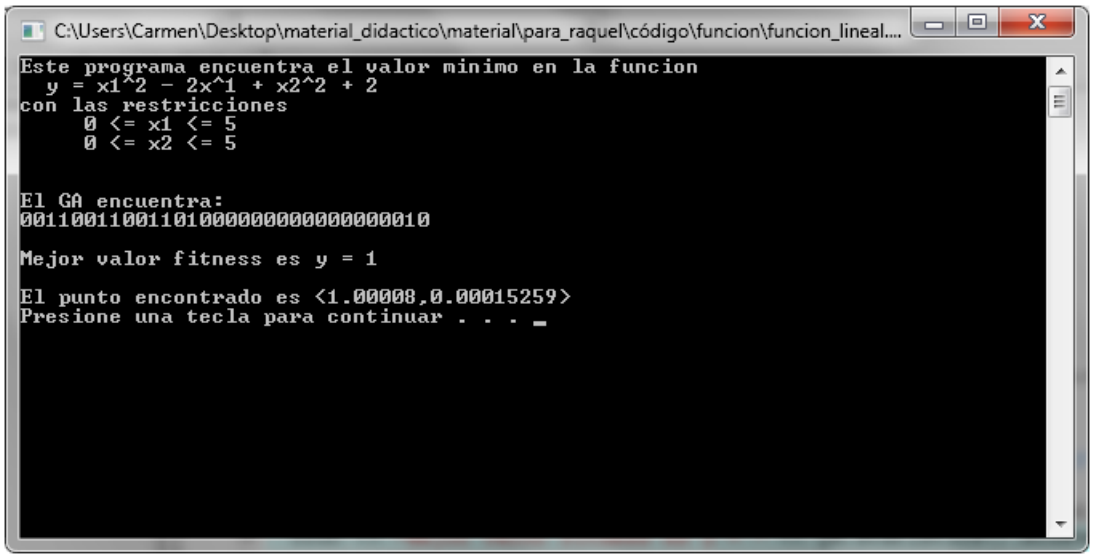

Figura 7.1: Salida del problema de optimización de una función

```
1  /* ------------ PROBLEMA DE LAS N REINAS ------------ */
2
3  #include <ga/GASimpleGA.h> //  Algoritmo Genético simple
4  #include <ga/GA1DArrayGenome.h> // genoma --> array de enteros (dim. 1)
5  #include <iostream>
6  #include <fstream>
7  using namespace std;
8
9  float Objective(GAGenome &); // Función objetivo --> definida más adelante
10 GABoolean Termina(GAGeneticAlgorithm &); // Función de terminación -->
       definida más adelante
11
12 int main(int argc, char **argv){
13
14 int nreinas = 8;
15
16 cout << "Problema de las " << nreinas << " reinas \n\n";
17 cout.flush();
18
19 // Establecemos una semilla para el generador de números aleatorios
20
21 GARandomSeed((unsigned int)atoi(argv[1]));
22
```

```
23 // Declaramos variables para los parámetros del GA y las inicializamos
24
25 int popsize = 100;
26 int ngen = 10000;
27 float pmut = 0.1;
28 float pcross = 0.9;
29
30 // Conjunto enumerado de alelos --> valores posibles de cada gen del genoma
31
32 GAAlleleSet<int> alelos;
33 for(int i = 0; i < nreinas; i++) alelos.add(i);
34
35 // Creamos el genoma
36
37 GA1DArrayAlleleGenome<int> genome(nreinas,alelos,Objective,NULL);
38
39 // Creamos el algoritmo genético
40
41 GASimpleGA ga(genome);
42
43 // Inicializamos - minimizar la función objetivo, tamaño de la población, n
       . generaciones, prob. mutación, prob. cruce, función de terminación y
       le indicamos que evolucione.
44
45 ga.minimaxi(-1);
46 ga.populationSize(popsize);
47 ga.nGenerations(ngen);
48 ga.pMutation(pmut);
49 ga.pCrossover(pcross);
50 ga.terminator(Termina);
51 ga.evolve();
52
53 // Imprimimos el mejor individuo que encuentra el GA y su valor fitness
54
55 cout << "El GA encuentra:\n" << ga.statistics().bestIndividual() << "\n";
56 cout << "Mejor valor fitness es " << ga.statistics().minEver() << "\n";
57
58 system("pause");
59
60 return 0;
61 }
62
63 // Definimos la función objetivo.
64
```

```
65 float Objective(GAGenome& g) {
66   GA1DArrayAlleleGenome<int> & genome = (GA1DArrayAlleleGenome<int> &)g;
67   float jaques = 0;
68   int c,f;
69
70   // si repetido en vector es un jaque de misma fila
71
72   for(int i = 0; i < genome.length(); i++)
73           for(int j = i+1; j < genome.length(); j++)
74                   if (genome.gene(i)==genome.gene(j)) jaques++;
75
76   // si diagonal también es jaque
77
78   for(int en_est = 0; en_est < genome.length(); en_est++){
79
80     // diagonal derecha abajo
81     c = en_est+1;
82     f = genome.gene(en_est)+1;
83     while ((c < genome.length()) && (f < genome.length())){
84             if (genome.gene(c) == f) jaques++;
85             c++;
86             f++;
87     }
88     // diagonal derecha arriba
89     c = en_est + 1;
90     f = genome.gene(en_est) - 1;
91     while ((c < genome.length()) && (f >= 0)){
92             if (genome.gene(c) == f) jaques++;
93             c++;
94             f--;
95     }
96   }
97   return jaques;
98 }
99
100 // Definimos la función de terminación para que termine cuando alcance el
      número de generaciones o el fitness 0
101
102 GABoolean Termina(GAGeneticAlgorithm &ga){
103           if ((ga.statistics().minEver() == 0) || (ga.statistics().
                 generation() == ga.nGenerations())) return gaTrue;
104           else return gaFalse;
105 }
```

Cuando ejecutamos el programa obtenemos una salida como la mostrada en la Figura 7.2.

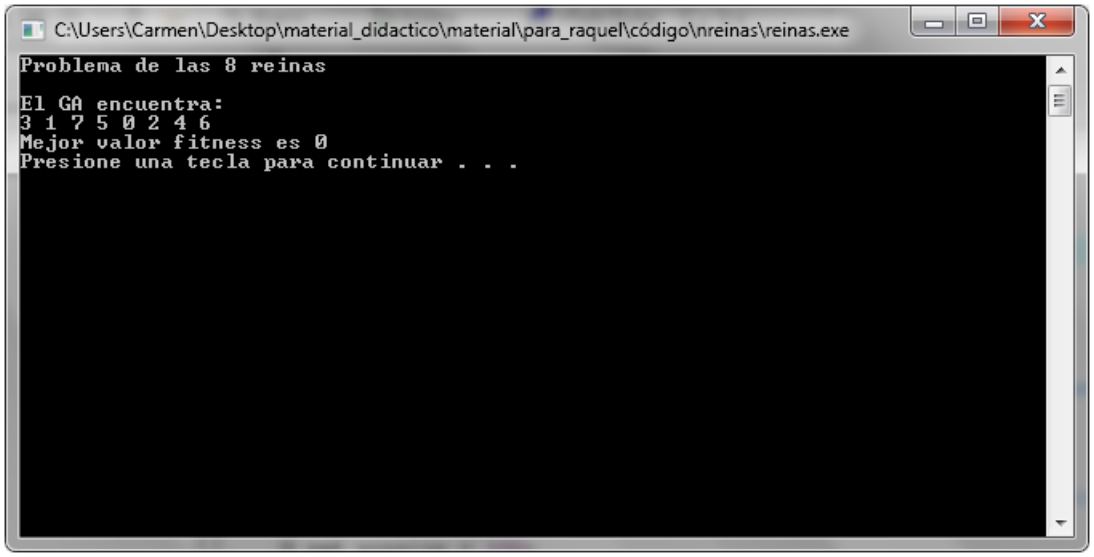

Figura 7.2: Salida del problema de las N reinas

7.6 Resolviendo el problema de la mochila 0-1

Por último, abordamos la resolución del problema de la mochila 0-1.

Para resolver este problema, usaremos un individuo formado por un string binario de dimensión 1 y de tamaño igual al número de objetos disponibles en el problema y susceptibles de ser introducidos en la mochila. De esta forma, el gen i tomará el valor 1 cuando el objeto i está dentro de la mochila y tomará valor 0 cuando el objeto i no está en la mochila.

El fitness asignado a un individuo será la suma de los beneficios de los objetos introducidos en la mochila en dicha solución (con gen a 1 en la representación). Por lo tanto, el algoritmo genético tratará de encontrar el individuo con mayor fitness (mayor beneficio) que no sobrepase la capacidad o peso máximo de la mochila determinado por el problema. Es debido a esta última restricción (la capacidad o peso máximo de la mochila) por lo que no podremos usar los operadores genéticos proporcionados por la librería y tendremos que definir unos nuevos. Los operadores definidos (inicio, cruce

y mutación) deberán contemplar que los nuevos individuos obtenidos no sobrepasan la capacidad máxima de la mochila ya que en este caso los individuos no son válidos. Por lo tanto, los operadores que definimos proporcionarán siempre individuos que respetan la restricción.

```
1  /* ----------------------------------------------------------------
2
3    Los datos del problema se leen de un fichero
4
5    Se definen el inicializador, el cruce y la mutación para que no generen
6    individuos no válidos
7
8    Se ejecuta mochila fichero_datos semilla
9
10 ----------------------------------------------------------------*/
11
12 #include <ga/GASimpleGA.h>  // usaremos un algoritmo genético simple
13 #include <ga/GA1DBinStrGenome.h> // y el genoma string binario de 1
       dimensión
14 #include <ga/std_stream.h>
15 #include <iostream>
16 #include <fstream>
17 using namespace std;
18
19 float Objective(GAGenome &);  // La función objetivo la definimos más
       adelante
20
21 void miInicio(GAGenome &); // Definimos un operador de inicio
22
23 int miMutacion(GAGenome &,float); // Definimos un operador de mutación
24
25 int miCruce(const GAGenome&,const GAGenome &,GAGenome*,GAGenome*); //
       Definimos un operador de cruce
26
27 struct datos{
28           int capacidad; // capacidad de la mochila
29           int * pebe[2]; // pebe[0][i] contiene el peso del objeto i y pebe
                 [1][i] contiene el beneficio del objeto i
30 };
31
32 int main(int argc, char **argv)
33 {
34   cout << "Mochila 0/1 \n\n";
```

```
35   cout << "Este programa trata de resolver el problema de la mochila\n\n";
36   cout.flush();
37
38 // Especificamos una semilla aleatoria.
39
40   GARandomSeed((unsigned int)atoi(argv[2]));
41
42 // Declaramos variables para los parametros del GA e inicializamos algunos
       valores
43
44   int popsize  = 100;
45   int ngen     = 2000;
46   float pmut   = 0.001;
47   float pcross = 0.9;
48
49 // Leemos datos del problema
50
51    struct datos D;
52    int tam;
53
54    ifstream f(argv[1]);
55
56    f>>D.capacidad; // se lee la capacidad de la mochila
57    f>>tam; // se lee el número de objetos
58
59    D.pebe[0]=new int[tam];
60    D.pebe[1]=new int[tam];
61
62    for(int kk=0;kk<tam;kk++) {
63            f>>D.pebe[1][kk];    // se leen beneficios
64            f>>D.pebe[0][kk];    // se leen pesos
65            }
66
67    f.close();
68
69 // Ahora creamos el GA y lo ejecutamos. Primero creamos el genoma que es
       usado por el GA para clonarlo y crear una población de genomas.
70
71 // Especificamos la función de inicio, de cruce y de mutación que queremos
       utilizar.
72
73   GA1DBinaryStringGenome genome(tam, Objective,(void *)&D);
74   genome.initializer(miInicio);
75   genome.crossover(miCruce);
```

```
76    genome.mutator(miMutacion);
77
78  // Una vez creado el genoma, creamos el algoritmo genético e inicializamos
        sus parámetros: tamaño de la población, número de generaciones,
        probabilidad de mutación, y probabilidad de cruce. Finalmente, le
        indicamos que evolucione.
79
80    GASimpleGA ga(genome);
81    ga.populationSize(popsize);
82    ga.nGenerations(ngen);
83    ga.pMutation(pmut);
84    ga.pCrossover(pcross);
85    ga.evolve();
86
87  // calculamos la capacidad consumida por la mejor solución encontrada
88
89    float capa=0;
90
91    GA1DBinaryStringGenome & mejor = (GA1DBinaryStringGenome &)ga.statistics
          ().bestIndividual();
92
93    for(int k=0;k<mejor.length();k++)
94            capa+=(mejor.gene(k)*D.pebe[0][k]);
95
96  // Imprimimos el mejor genoma encontrado por el GA, su fitness y capacidad
        usada
97
98    cout << "El GA encuentra:\n" << ga.statistics().bestIndividual() << "\n\n
          ";
99    cout << "Mejor valor fitness es " << ga.statistics().maxEver() << "\n\n";
100   cout << "Capacidad consumida " << capa << "\n\n";
101
102   system("pause");
103   return 0;
104 }
105
106 // Definimos la función objetivo
107
108 float Objective(GAGenome& g) {
109   GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &)g;
110   struct datos *D1=(struct datos *)genome.userData(); // recuperamos la
          estructura de datos
111   float valor=0.0;
112
```

```
113   // devolvemos el beneficio
114
115   for(int i=0; i<genome.length(); i++)
116           valor+=(genome.gene(i)*D1->pebe[1][i]);
117
118   return valor;
119 }
120
121 // Definimos una función de inicio que genere individuos válidos
122
123 void miInicio(GAGenome &g){
124       GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &)g;
125       struct datos *D1=(struct datos *)genome.userData();
126       float valor=0.0;
127
128       for(int i=0;i<genome.length();i++){
129               valor+=D1->pebe[0][i];
130
131               // comprobamos si meter el objeto i en la mochila sobrepasa la
                       capacidad si la sobrepasa, el objeto no se puede incluir;
                       si no se sobrepasa, se dedide aleatoriamente si se
                      incluye
132
133               if (valor>D1->capacidad){
134                                       genome.gene(i,0);
135                                       valor-=D1->pebe[0][i];
136               }
137               else {
138                       genome.gene(i,GARandomInt());
139                       if (!(genome.gene(i))) valor-=D1->pebe[0][i];
140               }
141       }
142 }
143
144 // Definimos un operador de mutación que genere individuos válidos
145
146 // Parámetros de entrada: genoma y probabilidad de mutación
147
148 // Valor de salida: número de mutaciones
149
150 int miMutacion(GAGenome &g,float pmut){
151     GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &)g;
152     struct datos *D1=(struct datos *)genome.userData();
153     float count=0.0;
```

```
154     int nmut=0;
155
156     if (pmut<=0.0) return 0;
157
158     // calculamos el peso actual de la mochila
159
160     for(int i=0; i<genome.length(); i++)
161           count+=(genome.gene(i)*D1->pebe[0][i]);
162
163     // si un gen debe mutarse, si es un objeto que se extrae no se
            comprueba nada, si es un objeto que se incluye se comprueba que no
            sobrepase la capacidad de la mochila. Si la sobrepasa el gen no se
            muta
164
165     for(int i=0; i<genome.length(); i++){
166             if (GAFlipCoin(pmut)) {
167                nmut++;
168                if (genome.gene(i)) genome.gene(i,0);
169                else {
170                     count+=D1->pebe[0][i];
171                     if(count>D1->capacidad) {
172                                             count-=D1->pebe[0][i];
173                                             nmut--;
174                                             }
175                     else genome.gene(i,1);
176                }
177             }
178     }
179     return nmut;
180 }
181
182 // Definimos el operador de cruce para que genere individuos válidos
183
184 // Parámetros de entrada: los dos genomas padres y los dos hijos
185
186 // Valor de salida: el numero de hijos que genera
187
188 int miCruce(const GAGenome& p1,const GAGenome & p2,GAGenome* c1,GAGenome*
        c2){
189     GA1DBinaryStringGenome & m = (GA1DBinaryStringGenome &)p1;
190     GA1DBinaryStringGenome & p = (GA1DBinaryStringGenome &)p2;
191     struct datos *D1=(struct datos *)m.userData();
192     int n=0;
193
```

```
194     int punto1=GARandomInt(0,m.length()); // generamos el punto de cruce
195     int punto2=m.length()-punto1;
196
197     // el hijo 1 hereda la primera parte del padre 1 y la segunda parte del
            padre 2. Comprobamos el peso del hijo generado y cuando un objeto
        hace que se sobrepase la capacidad de la mochila, se extrae.
198
199     if (c1){
200             GA1DBinaryStringGenome & h1= (GA1DBinaryStringGenome &)*c1;
201             float count=0.0;
202             h1.copy(m,0,0,punto1);
203             h1.copy(p,punto1,punto1,punto2);
204             for(int i=0;i<h1.length();i++){
205                     count+=(h1.gene(i)*D1->pebe[0][i]);
206                     if (count>D1->capacidad) {
207                         h1.gene(i,0);
208                         count-=D1->pebe[0][i];
209                     }
210             }
211             n++;
212     }
213
214     // el hijo 2 hereda la primera parte del padre 2 y la segunda parte del
            padre 1. Comprobamos el peso del hijo generado y cuando un objeto
        hace que se sobrepase la capacidad de la mochila, se extrae.
215
216     if (c2){
217             GA1DBinaryStringGenome & h2= (GA1DBinaryStringGenome &)*c2;
218             float count=0.0;
219             h2.copy(p,0,0,punto1);
220             h2.copy(m,punto1,punto1,punto2);
221             for(int i=0;i<h2.length();i++){
222                     count+=(h2.gene(i)*D1->pebe[0][i]);
223                     if (count>D1->capacidad) {
224                        h2.gene(i,0);
225                        count-=D1->pebe[0][i];
226                     }
227             }
228             n++;
229     }
230     return n;
231 }
```

Cuando ejecutamos el programa usando como datos los mostrados en la Tabla 7.1, obtenemos una salida como la mostrada en la Figura 7.3.

5
10
0 1
1 1
2 1
3 1
4 1
5 1
6 1
7 1
8 1
9 1
capacidad
número de objetos
beneficio y peso de cada objeto

Tabla 7.1: Datos para una instancia de la mochila 0-1

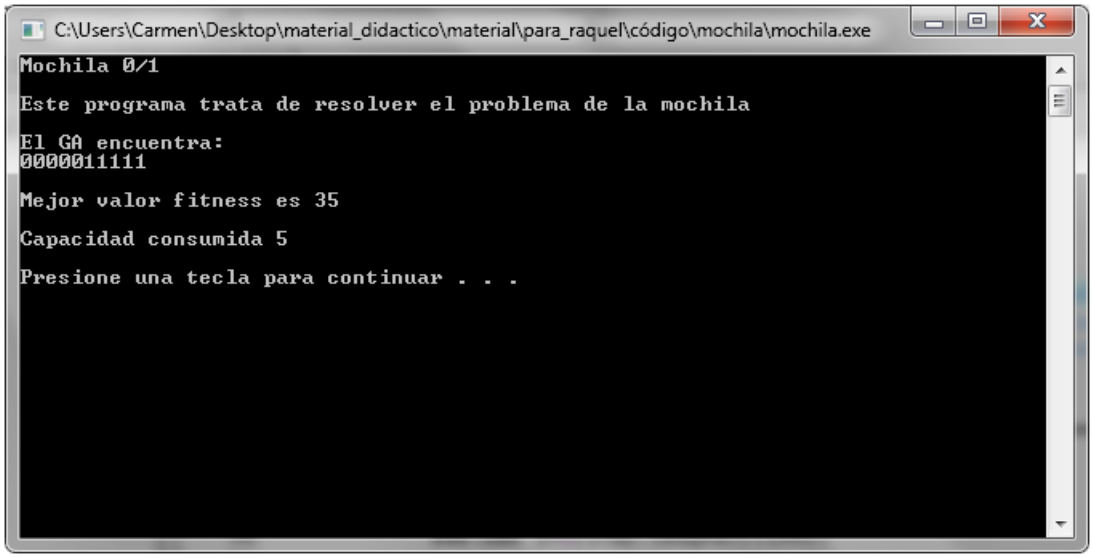

Figura 7.3: Salida del problema de la mochila 0-1