UNIVERSIDAD DE GUANAJUATO

DIVISIÓN DE INGENIERÍAS

CAMPUS – IRAPUATO SALAMANCA

Inteligencia Artificial

PROF. GARCÍA CAPULÍN CARLOS HUGO

PRÁCTICA 2

Algoritmo Genético Simple

Martínez Lona Verónica Montserrat

Martes 09 Mayo, 2017

Introducción:

La mecánica natural darwiniana de supervivencia de los seres mejores adaptados puede ser aplicada a la resolución de diferentes problemas de optimización en ingeniería y otros campos. En este trabajo se ha desarrollado un algoritmo genético simple para ser utilizado en la resolución de la ecuación siguiente:

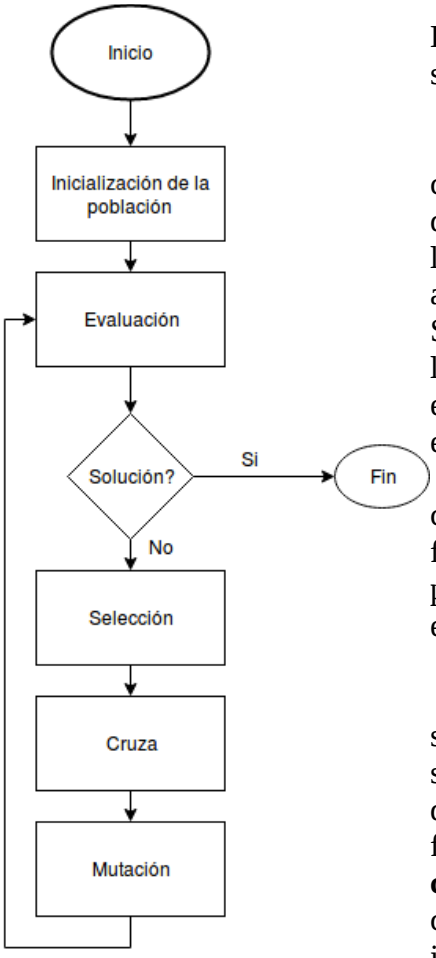
$$f(x)=10V+\sum_{i=0}^{10} ng(x_i)*\sin(|x_i|)$$

Mediante la asimilación de la estructuras a cromosomas representados por cadenas binarias y partiendo de una población generada aleatoriamente, se efectúan los procesos naturales de selección, cruce y mutación. Se evoluciona y así, generación tras generación, hacia el mejor adaptado; o sea, la estructura óptima.

Los Algoritmos Genéticos tratan de optimizar imitando el comportamiento natural; su aplicación en ingeniería se debe a los trabajos de Goldberg, y muy recientemente han aparecido algunas optimizaciones estructurales por este método. Sin embargo, su desarollo en el campo de la bioquímica se inició en la década de los 60 donde un grupo de biólogos utilizó los computadores digitales para realizar simulaciones genéticas; a partir de este momento empiezan a aparecer trabajos cercanos a los algoritmos genéticos; pero fue John Holland, quién en 1975 integró y elaboró sus dos fundamentos básicos: la capacidad de una representación simple (cadenas de bits) para mostrar estructuras complejas y la potencia de simples transformaciones para mejorar tales estructuras. Holland observó, a través de su teorema *Teorema Fundamental de los Algoritmos Geneticos* que con el conveniente control de la estructura, aparecen rápidas mejoras en las cadenas de bits como sucede en las poblaciones animales.

Para resolver un problema con un Algoritmo Genetico debemos concretar:

1. La representación cromosomal de la población.
2. La forma de crear una población inicial.
3. Una función de evaluación que interprete la acción del medio ambiente sobre cada uno de los individuos de la población.
4. Los operadores genéticos (cruce, mutación, etc.) que alteren la composición de los hijos.
5. La evaluación de los parámetros de control (tamaño de la población, probabilidad de reproducción, etc).

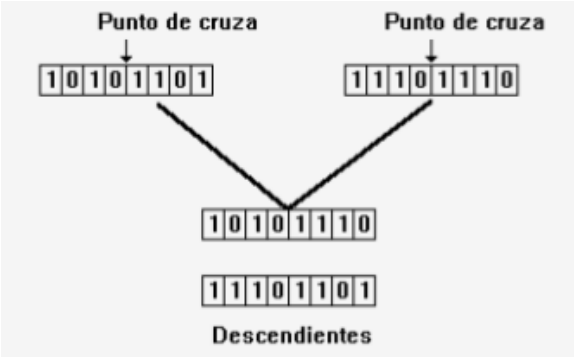


La operación de un algoritmo genético simple funciona de la siguiente forma:

Primero, se genera aleatoriamente la población inicial, que estará constituída por un conjunto de cromosomas, o cadenas de caracteres que representan las soluciones posibles del problema. A cada uno de los cromosomas de esta población se le aplicará la función de fitness a fin de saber qué tan buena es la solución que está codificando. Sabiendo el fitness de cada cromosoma, se procede a la selección de los que se cruzarán en la siguiente generación (presumiblemente, se escogerá a los "mejores"). El metodo de seleccion que se utilizo en este caso fue el de la Ruleta:

Metodo de la Ruleta: Este método es muy simple, y consiste en crear una ruleta en la que cada cromosoma tiene asignada una fracción proporcional a su fitness. Aleatoriamente se escoge una posición en la ruleta, lo que nos da como resultado un cromosoma elegido.

Una vez realizada la selección, se procede a la reproducción sexual o cruza de los individuos seleccionados. En esta etapa, los sobrevivientes intercambiarán material cromosómico y sus descendientes formarán la población de la siguiente generación. Las forma que se utilizó en este trabajo fue el **uso de un punto único de cruza** en el que se escoge de forma aleatoria sobre la longitud de la cadena que representa el cromosoma, y a partir de él se realiza el intercambio de material de los 2 individuos. Normalmente la cruza se maneja dentro de la implementación del algoritmo genético como un porcentaje que indica con qué frecuencia se efectuará. Esto significa que no todas las parejas de cromosomas se cruzarán, sino que habrán algunas que pasarán intactas a la siguiente generación.



Además de la selección y la cruza, existe otro operador llamado mutación, el cual realiza un cambio a uno de los genes de un cromosoma elegido aleatoriamente. Cuando se usa una representación binaria, el gen seleccionado se sustituye por su complemento (un cero cambia en uno y viceversa). Este operador permite la introducción de nuevo material cromosómico en la población, tal y como sucede con sus equivalentes biológicos. Al igual que la cruza, la mutación se maneja como un porcentaje que indica con qué frecuencia se efectuará, aunque se distingue de la primera por ocurrir mucho más esporádicamente.

Finalmente, para detener el algoritmo normalmente se usan 2 criterios principales de detención : correr el algoritmo genético durante un número máximo de generaciones o detenerlo cuando la población se haya estabilizado (i.e. cuando todos o la mayoría de los individuos tengan el mismo fitness).

¿Qué Ventajas y Desventajas tienen con respecto a otras técnicas de búsqueda?

- No necesitan conocimientos específicos sobre el problema que intentan resolver.
- Operan de forma simultánea con varias soluciones, en vez de trabajar de forma secuencial como las técnicas tradicionales.
- Cuando se usan para problemas de optimización -maximizar una función objetivo- resultan menos afectados por los máximos locales (falsas soluciones) que las técnicas tradicionales.
- Resulta sumamente fácil ejecutarlos en las modernas arquitecturas masivas en paralelo.
- Usan operadores probabilísticos, en vez de los típicos operadores determinísticos de las otras técnicas.
- Pueden tardar mucho en converger, o no converger en absoluto, dependiendo en cierta medida de los parámetros que se utilicen -tamaño de la población, número de generaciones, etc.-.
- Pueden converger prematuramente debido a una serie de problemas de diversa índole.

Desarrollo:

Para la resolución de la ecuación

$$f(x)=10V+\sum_{i=0}^{10} ng(x_i)*\sin(|x_i|)$$

se utilizaron los parámetros dados por la teoría.

- Número de genes, que es igual al número de variables de la ecuación
  - const uint NUMGENS = 10;
- Los límites de x:
  - $x\in[-500,500]$
- Por lo tanto, el número de bits por gen:
  - const uint BITGEN = 9;
  - *Por la naturaleza del problema y para un fácil re-uso del algortimo todos los genes tienen la misma longitud*
- El número de cromosomas a probar es:
  - 10, 20, 30, 50, 100, 200

Las estructuras que sirven para el manejo de la población de soluciones son las siguientes:

- **POPULATION**
  - Maneja los cromosomas que conforman las posibles soluciones de la población. También guarda el infex del mejor cromosa histórico y el fitness del mejor cromosoma histórico.

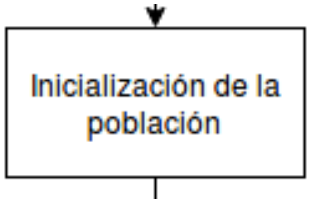
```
typedef struct {
   CHROMOSOME* chromosomes; // Arreglo de cromosomas que conforman la ecuacion
   double bestChromosome;   // Index del cromosoma con mejor fitness
} POPULATION;
```

- **CHROMOSOME**
  - Almacena la información de un cromosoma, maneja los genes que este contiene. También almacena el fitness del cromosoma.

```
typedef struct {
  //unsigned int numGens; // Numero de genes
  GEN* genes;          // Conjunto de los genes que conforma el cromosoma
  double chromFit;     // Fitness del gen
  double bestFit;      // Mejor fitness historico
} CHROMOSOME;
```

- **GEN**
  - Almacena la información del gen (la cadena de bits que lo conforma). De igual modo, almacena el fenotipo del gen.

```
typedef struct {
  char* gen;           // Arreglo que contiene al gen, bit por bit.
  double fenotype;     // Guarda el fenotipo, lo que es el valor numerico del gen completo
}GEN;
```



Las funciones que se encargan de la inicialización de la población son las siguientes:

- **Inicializar gen** (*void Gen_Init(GEN *pGen);)*
  - Inicializa cada bit del gen. Le asigna al gen un valor aleatorio (1 o 0) bit por bit.

INPUT	OUTPUT
Un gen	void

- **Inicializar chromosoma** *(void Chromosome_Init(CHROMOSOME *pChromosome);)*
  - Inicializa un cromosoma y su conjunto de genes. Tambien se le asigna un fitness igual a 0.

INPUT	OUTPUT
Un cromosoma	void

- **Inicializar población** *(POPULATION* Population_Init(void);)*
  - Inicializa una poblacion y sus cromosamas. La variable de bestChromosome es igualada a 0.

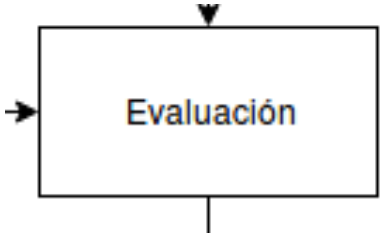
INPUT	OUTPUT
Una población	Una población inicializada.

Una población inicializada luce así:

```
tsukimo@MadHattersDomain:~/Docum
 Individuo   1
        Gen   1 :  110111011
        Gen   2 :  000110111
        Gen   3 :  100010001
        Gen   4 :  000100000
        Gen   5 :  001111000
        Gen   6 :  000000100
        Gen   7 :  100111011
        Gen   8 :  100100100
        Gen   9 :  001111010
        Gen  10 :  110111101
chromosome's Fitness: 0.000000
-----
```

```
 Individuo   2
        Gen   1 :  011001010
        Gen   2 :  100111010
        Gen   3 :  010001110
        Gen   4 :  011010100
        Gen   5 :  100111000
        Gen   6 :  111001001
        Gen   7 :  011101000
        Gen   8 :  100011110
        Gen   9 :  011010000
        Gen  10 :  111111111
chromosome's Fitness: 0.000000
-----
```

```
 Individuo   3
        Gen   1 :  010111000
        Gen   2 :  110110100
        Gen   3 :  010101001
        Gen   4 :  110100011
        Gen   5 :  111000000
        Gen   6 :  101100110
        Gen   7 :  101011011
        Gen   8 :  000110100
        Gen   9 :  011101011
        Gen  10 :  011101001
chromosome's Fitness: 0.000000
-----
```

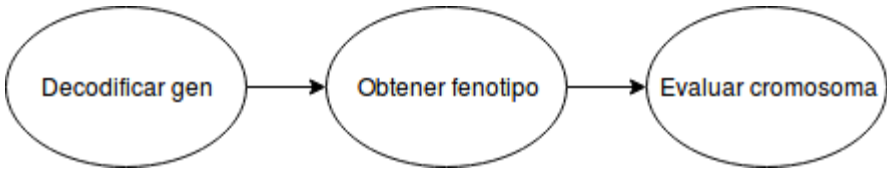


La evaluación se hace sobre la siguiente equación

$$f(x) = 10V + \sum_{i=0}^{10} ng(x_i)\sin(|x_i|)$$

$$V = 4189.829101$$

El proceso de evaluación es el siguiente:



Las funciones que realizan la evaluación son las siguientes:

- **Decodificar gen** *(int Decode_Gen(GEN* pGen);)*
    - Calcula el valor numérico de cada gen, conviertiendo el conjunto binario a (entero) decimal.

INPUT	OUTPUT
Gen a decodificar	Genitpo calculado

- **Obtener el fenotipodouble** *(Get_Fenotype(GEN* pGen, int binMax, int range);)*
  - Calcula el valor numérico de punto flotante que guarda cada gen, mediante la fórmula

$$fenotype = \frac{decodedGen}{binMax} \times range + lowerLimit$$

En donde:

decodedGen: El valor decodificado del gen

binMax: El valor máximo binario que se puede almacenar en un gen (cuando todos sus bits son 1)

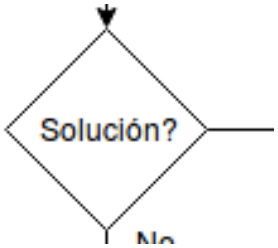
rango: El rango en el que se encuentran los valores del gen, dependen del problema.

LowerLimit: Límite inferior de los valores que puede tomar el gen.

INPUT	OUTPUT
Gen del cual se obtendrá el fenotipo	Valor numérico calculado
binMax	
Rango	

- **Evaluar individuo** *(void Evaluate_Chromosome(CHROMOSOME* pChromosome, int binMax, int range);)*
  - Evalúa el cromosoma en la ecuación.

INPUT	OUTPUT
Gen a evaluar	void
binMax	
Rango	



Para detener el algoritmo y encontrar una solución óptima de definió una varible de error

const double ERROR = 0.1;

El error se evalúa sobre el mejor cromosoma, y para encontrarlo se utilizan las siguientes funciones:

- **Actualizar mejores** *(void UpdateBest(POPULATION *pPopulation))*
  - Actualiza los mejores fitness de cada cromosoma y de toda la poblacion. En realidad sólo se encarga de llamar las funciones que se explicarán a continuación, para actualizar el mejor fintess de cada cromosma y el mejor fitness global de la población.

INPUT	OUTPUT
La población	void

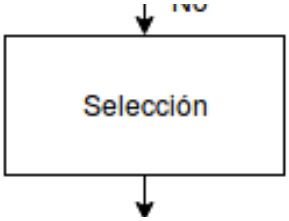
- **Actualizar mejores fitness de cada crosoma** *(void UpdateChromosomeBest(POPULATION *pPopulation))*
  - Actualiza el mejor fitness histórico de cada cromosoma.

INPUT	OUTPUT
La población	void

- **Actualizar mejores fitness de cada la población**

  *(void UpdatePopulationBest(POPULATION *pPopulation))*
  - Actualiza el mejor fitness histórico de la población.

INPUT	OUTPUT
La población	void



Las funciones que sirven para la implementación son los siguientes:

- **Calcula fitness total** *(double Total_Fit(POPULATION *pPopulation))*
    - Suma los fitness de cada uno de los cromosomas.

INPUT	OUTPUT
La población	Valor del fitness total

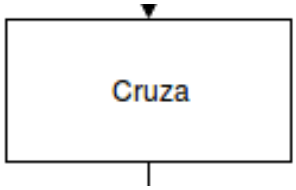
- **Calcular la probabilidad de selección de cada cromosoma**

  (ARRAY Selection_Probability(POPULATION* pPopulation, int totalFit))*
    - Divide el fitness de cada cromosoma entre el fitness total.

INPUT	OUTPUT
La población	ARRAY con la probabilidad de selección e index de cada cromosoma
El fitness total	

- **Método de la ruleta** *(int* Roulette_Metod(POPULATION *pPopulation))*
    - Cada cromosoma tiene una probabilidad de seleccion, se acomodan de menor a mayor)en este caso mediante el método de quick sort), simulando una ruleta. Se obtiene un numero random que apunta a una sección específica de la ruleta, donde se encuentra un cromosoma, el index de este cromosoma en guardado en un arreglo en el cual se van armando pares de cromosomas para la cruza.

INPUT	OUTPUT
La población	Arreglo de enteros con las parejas que se cruzarán



- **Cruza población** *(POPULATION* Cross_Population(POPULATION *pPopulation, int *selPair))*
    - La cruza se hace con **un sólo punto de cruza**, que ya ha sido explicada en la introducción. El punto de cruza se escoge aleatoriamente para cada par de cromosomas. La función sobreescribe nuestra población original, por lo que debemos guardar en la nueva poblaicón los valores de mejor cromosoma y mejor fitness.

INPUT	OUTPUT
La población	Nueva generación
El vector de parejas	

- **Mutar poblacion** *(void Muta_Population(CHROMOSOME *pChromosome);)*
    - Dependiendo de la probabilidad, cambia aleatoriamente el valor de un bit por su complemento.

INPUT	OUTPUT
La población	void

OTRAS FUNCIONES

- **Funcion de comparacion** *(int cmpfunc (const void * a, const void * b))*
  - Es para implementar correctamente la función qsort de la libreria stdlib. Hace comparaciones entre el valor que guarda un elemento del array.
- **Mostrar población** *(void Show_Population(POPULATION *pPopulation))*
  - Imprime la poblacion completa y otros datos de interes. Imprime el gen en binario y el fitness del cromosoma.

INPUT	OUTPUT
La población	void

- **Liberar población** *(void Free_Population(POPULATION *pPopulation))*
  - Libera el espacio de memoria que ocupa la población.

INPUT	OUTPUT
La población	void

Codigo

```
/*
   Simple Genetic Algorith
   Artficial Inteligent, DICIS. 2017.
   April 2017

   Funcion F6

   Galvan Hernandez Armando
   Martinez Lona Veronica Montserrat

*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

// Variable perteneciente a la ecuacion
#define V 4189.829101

typedef unsigned int uint;

// Number of genes == number of variables of the equation
const uint NUMGENS = 10;
// Gen lenght (for simplicity and re-use all the gens has the same length )
const uint BITGEN = 9; // Para poder almacenar el numero 500
// Number of indiduals in the populations(has to be pair)
// 10, 30, 50, 100, 200
const uint NUMCHROMOSOME = 200;
// Numero maximo de iteraciones
const uint NUMMAXIT = 500;
// Search space limits
const int LOWERLIM = -500;
const int UPPERLIM = 500;
//Probabilidad de mutacion 0.001 - 0.009
const double PMUTA = 0.008;
//Probabilidad de cruza
const double PCROSS = 0.8;
const double ERROR = 0.1;

typedef struct {
   char* gen;            // Arreglo que contiene al gen, bit por bit.
   // unsigned int bitgen;  // Longitud del gen.
   //double fenotype;       // Guarda el fenotipo, lo que es el valor numerico del gen completo
}GEN;

typedef struct {
   //unsigned int numGens; // Numero de genes
   GEN* genes;           // Conjunto de los genes que conforma el cromosoma
   double chromFit;      // Fitness del gen
   double bestFit;       // Mejor fitness historico
} CHROMOSOME;

//Estructura POPULATION, es para el manejo de la poblacion de cromosomas
typedef struct {
   CHROMOSOME* chromosomes; // Arreglo de cromosomas que conforman la ecuacion
   int bestChromosome;   // Index del cromosoma con mejor fitness
   // double bestGFit;       // Mejor fitness historico
} POPULATION;

// Estructura ARRAY, es auxiliar en la funcion Roulette_Metod.
// Guarda el valor del index (index) y el valor de la probabilidad de seleccion (value)
typedef struct {
   uint index;    // Index del cromosoma
   double value;  // Valor de probabilidad de seleccion
} ARRAY;

//_____________________________________ Init
// Inicializar gen
void Gen_Init(GEN *pGen);
```

```
// Inicializar chromosoma
void Chromosome_Init(CHROMOSOME *pChromosome);
// Inicalizar poblacion
POPULATION* Population_Init(void);
//_____________________________________ Evaluation
// Decodificar gen
int Decode_Gen(GEN* pGen);
// Obtener el fenotipo
double Get_Fenotype(GEN* pGen, int binMax, int range);
// Evaluar individuo
void Evaluate_Chromosome(CHROMOSOME* pChromosome, int binMax, int range);
//_____________________________________ Selection
// Compute the totalFit
double Total_Fit(POPULATION *pPopulation);
// seleccion probability
ARRAY* Selection_Probability(POPULATION* pPopulation, int totalFit);
// For quickSort function
int cmpfunc (const void * a, const void * b);
// Roulette Method
int* Roulette_Metod(POPULATION *pPopulation);
//_____________________________________ Evolucion
// Operador cruza
POPULATION* Cross_Population(POPULATION *pPopulation, int *selPair);
// Mutación
void Muta_Population(CHROMOSOME *pChromosome);
//_____________________________________ Actualizar mejores Es solucion?
// Actualizar mejores
void UpdateBest(POPULATION *pPopulation);
//Actualizar mejor fit del cromosoma
void UpdateChromosomeBest(POPULATION *pPopulation);
// Actualizar mejor cromosoma de la poblacion
void UpdatePopulationBest(POPULATION *pPopulation);
//_____________________________________ Finish
// Show poblacion
void Show_Population(POPULATION *pPopulation);
// Liberar poblacion
void Free_Population(POPULATION *pPopulation);

int main(void)
{
   srand(time(NULL));
   uint i, binMax, range, it;
   int bestChromosome;
   double bestFitness;
   int *selPair;
   POPULATION *pPopulation;

   // Inicalizamos poblacion
   pPopulation = Population_Init();

   binMax = 2 << (BITGEN - 1);
   range = UPPERLIM - LOWERLIM;

   for(it = 0; it < NUMMAXIT; it++)
   {
      // Evaluamos la poblacion
      for(i = 0; i < NUMCHROMOSOME; i++)
      {
         Evaluate_Chromosome(&(pPopulation -> chromosomes[i]), binMax, range);
      }
      UpdateBest(pPopulation);
      //Show_Population(pPopulation);

      // Show bestChromosome
      bestChromosome = pPopulation -> bestChromosome;
      bestFitness    = pPopulation -> chromosomes[bestChromosome].bestFit;

      // if(bestFitness == 0) printf("No ideal solution found\n");
      //else
      printf("bestChromosome : %d , bestFitness : %f\n", bestChromosome, bestFitness);
      printf("----------------------------Iteracion %d\n", it + 1);
      if(fabs(bestFitness - 10 * V) <= ERROR) break;
```

```
      // Seleccion
      selPair = Roulette_Metod(pPopulation);
      //Curza
      pPopulation = Cross_Population(pPopulation, selPair);
      // Mutacion
      for(i = 0; i < NUMCHROMOSOME; i++)
      {
         Muta_Population(&(pPopulation -> chromosomes[i]));
      }
   }

   Free_Population(pPopulation);

   return 0;
}


/*
   Inicializa cada bit del gen.
   Le asigna al gen un valor aleatorio (1 o 0) bit por bit.
   Inputs:
      Un gen vacio.
   Output:
      void
*/
void Gen_Init(GEN *pGen)
{
   uint i;

   pGen -> gen = (char*) malloc (BITGEN*sizeof(char));

   // Para cada bit del gen
   for(i = 0; i < BITGEN; i++)
   {
      // Alatoriamente asigna 1 o 0
      pGen -> gen[i] = (char)round((double) rand() / RAND_MAX);
   }

   return;
}

/*
   Inicializa el cromosoma
   Inicializa un cromosoma y su conjunto de genes .
   Tambien se le asigna un fitness igual a 0.
   Inputs:
      Un cromosoma
   Outputs:
      void
*/
void Chromosome_Init(CHROMOSOME *pChromosome)
{
   uint i;

   // El fitness se inicializa en 0
   pChromosome -> chromFit = 0;
   pChromosome -> bestFit = 0;

   //Se crea el conjunto de genes
   pChromosome -> genes = (GEN*) malloc (NUMGENS * sizeof(GEN));

   // Para cada gen
   for(i = 0; i < NUMGENS; i++)
   {
      //Se inicializa cada gen del cromosoma
      Gen_Init(&(pChromosome -> genes[i]));
   }

   return;
}

/*
   Inicializa la poblacion.
```

```
   Inicializa una poblacion y sus cromosamas.
   La variable de bestChromosome es igualada a 0.

   Inputs:
      Una poblacion vacia
   Outputs
      Una poblacion inicializada
*/
POPULATION* Population_Init(void)
{
   uint i;
   POPULATION* pPopulation;

   // Se reserva la memoria para la poblacion
   pPopulation = (POPULATION*) malloc (sizeof(POPULATION));

   // Por default, el mejor cromosoma es el 0
   pPopulation -> bestChromosome = 0;

   // Se reserva el espacio para el arreglo de cromosomas
   pPopulation -> chromosomes = (CHROMOSOME*) malloc (NUMCHROMOSOME * sizeof(CHROMOSOME));

   for(i = 0; i < NUMCHROMOSOME; i++)
   {
      // Se inicaliza cada cromosoma de la poblacion
      Chromosome_Init(&(pPopulation -> chromosomes[i]));
   }

   return pPopulation;
}

/*
   Decodificar el gen
   Calcula el valor numerico de cada gen, conviertiendo el conjunto binario a decimal

   Inputs:
      El gen a decodificar
   Outputs:
      El valor numerico calculado
*/
int Decode_Gen(GEN* pGen)
{
   int i;
   uint base = 1;
   uint number = 0;

   // Desde el bit menos significativo hasta el mas significativo
   // Por eso va "al reves" el conteo
   for(i = BITGEN - 1; i >= 0; i --, base *= 2)
   {
      number += pGen -> gen[i] * base;
   }

   return number;
}

/*
   Obtiene el fenotipo.
   Calcula el valor numerico de punto flotante que guarda cada gen
   Inputs:
      El gen del cual se obtendra el fenotipo.
      binMax: el numero maximo que se puede tener con ese numero de bits
      range: el rango donde se encuenra la solucion
   Outputs:
      El valor numerico calculado
*/
double Get_Fenotype(GEN* pGen, int binMax, int range)
{
   uint decodedGen;
   double fenotype;

   decodedGen = Decode_Gen(pGen);
   fenotype = (double)decodedGen / binMax * range + LOWERLIM;
```

```
   return fenotype;
}

/*
   Evalua el cromosoma en la ecuacion
   Fx = 10V + sum[-xi * sin(sqrt(abs(xi)))]
   Inputs:
      El cromosoma a evaluar
      binMax: el numero maximo que se puede tener con ese numero de bits
      range: el rango donde se encuenra la solucion
   Outputs:
      void
*/
void Evaluate_Chromosome(CHROMOSOME* pChromosome, int binMax, int range)
{
   uint i;
   double x;
   double F = 0;

   //Para cada gen
   for(i = 0; i < NUMGENS; i++)
   {
      x = Get_Fenotype(&(pChromosome -> genes[i]), binMax, range);
      // Si el fenotipo de un se sale de los limites asignados entonces el cromosoma no es una solucion
      if(x > UPPERLIM || x < LOWERLIM)
      {
         pChromosome -> chromFit = 0;
         return;
      }
      // Si el fenotipo tiene un valor correcto
      else
      {
         F += -1 * x * sin(sqrt(fabs(x)));;
      }
   }

   pChromosome -> chromFit = 10 * V + F;

   return;
}

/*
   Calcula el fitness total.
   Suma los fitness de cada uno de los cromosomas.
   Inputs:
      La poblacion
   Outputs:
      Valor del fitness total
*/
double Total_Fit(POPULATION *pPopulation)
{
   uint i;
   double totalFit = 0;

   //Por cada cromosoma
   for(i = 0; i < NUMCHROMOSOME; i++)
   {
      totalFit += pPopulation -> chromosomes[i].chromFit;
   }

   return totalFit;

}

/*
   Obtiene la probabilidad de seleccion de cada cromosoma
   Divide el fitness de cada cromosma entre el fitness total
   Inputs:
      La poblacion
      El fitness total
   Outputs:
      Un arreglo del tipo ARRAY, que contiene el valor de la probabilidad de seleccion y el indice del comosoma
```

```
*/
ARRAY* Selection_Probability(POPULATION* pPopulation, int totalFit)
{
   uint i;
   ARRAY *pSelProb;

   pSelProb  = (ARRAY*)  malloc (NUMCHROMOSOME * sizeof(ARRAY));

   for(i = 0; i < NUMCHROMOSOME; i++)
   {
      pSelProb[i].value = pPopulation -> chromosomes[i].chromFit / totalFit;
      pSelProb[i].index = i;
   }

   return pSelProb;
}

/*
   Funcion de comparacion.
   Regresa el elemento del tipo ARRAY cuyo .value sea mayor.
   Es para poder implementar la Funcion qsort de la libreria stdlib
   Inputs:
      Dos elemtos del arreglo
   Outputs:
      un valor entero
*/
int cmpfunc (const void * a, const void * b)
{
   const ARRAY *x;
   const ARRAY *y;
   x = a;
   y = b;

   if (x -> value < y -> value) return -1;
   else if (x -> value > y -> value) return 1;
   return 0;
}

/*
   Aplica el metodo de la ruleta
   Cada cromosoma tiene una probabilidad de seleccion, se acomodan de menor a mayor,
   simulando una ruleta. Se obtiene un numero random que apunta a una sección
   específica de la ruleta, donde se encuentra un cromosoma, el index de este
   cromosoma en guardado en un arreglo en el cual se van armando pares de
   cromosomas para la cruza.
   Inputs:
      La poblacion
   Outputs:
      Un arreglo tipo entero con los pares de cromosomas que se cruzaran
*/
int* Roulette_Metod(POPULATION *pPopulation)
{
   uint i, j;
   double rNum, sumAux;
   ARRAY *pSelProb;
   int *pSelPair; //Guarda los indices de los cromosomas que seran pareja
   pSelPair = (int*) malloc (NUMCHROMOSOME * sizeof(int));

   // Se calcula la probabilida de seleccion de cada cromosoma de la poblacion
   pSelProb = Selection_Probability(pPopulation, Total_Fit(pPopulation));
   // Se ordenan de menor a mayor la probabilidad de seleccion de cada cromosoma
   qsort(pSelProb, NUMCHROMOSOME, sizeof(ARRAY), cmpfunc);

   for(i = 0; i < NUMCHROMOSOME; i++)
   {
      // Valor random que nos dice que parte de la ruleta se debe escoger
      rNum = (double) rand() / RAND_MAX;
      for(j = 0, sumAux = 0; j < NUMCHROMOSOME; j++)
      {
         //Va acumulando las probabilidades para avanzar de posicion en la ruleta
         sumAux += pSelProb[j].value;
         // Cuando se alcanza la posicion elegida en la ruleta
         if(sumAux > rNum)
```

```
        {
          // Se guarda el index del cromosoma para formar parejas
          pSelPair[i] = pSelProb[j].index;
          break;
        }
      }
    }

    // Libera la memoria del arreglo de probabilidades
    free(pSelProb);

    return pSelPair;
}

/*
//-------------------------------------------------------Cruza poblacion
Hace la cruza con 1 punto de cruza. (Intercambia bits entre dos cromosomas)
    Inputs:
      Poblacion
      Vector de parejas
    Output:
      Nueva generacion
*/
POPULATION* Cross_Population(POPULATION *pPopulation, int *selPair){
        int i, j, k, nCouples, nBits, bitCounter, crossPoint;
        double vCross; // Valor de cruza, determina si la pareja se cruzara o no
        POPULATION *newPopulation; // Guarda la nueva generacion

    // Numero de parejas, es la mitad del numero de cromosomas
        nCouples = NUMCHROMOSOME * 0.5;
    // Numero de bits totales del cromosoma
        nBits = NUMGENS * BITGEN;

    // Creamos e inicializamos la nueva poblacion
        newPopulation = Population_Init();

    // Para cada pareja
        for(i = 0; i < nCouples; i++){
            vCross = (double)rand()/RAND_MAX;
      // Si el valor de cruza es menor a la probabilidad de cruza hay cruza
            if(vCross <= PCROSS){
                bitCounter = 0;
        // El punto de cruza es aleatorio py diferente para cada par de parejas
                crossPoint = (int)(((double)rand() / RAND_MAX * (nBits - 1)) + 1);
        // Para cada gen
                for(j = 0; j < NUMGENS; j++){
          // Para cada bit
                    for (k = 0; k < BITGEN; k++){
                        if(bitCounter < crossPoint){ //Copia de forma normal hasta cierto punto
                            newPopulation->chromosomes[2*i].genes[j].gen[k] = pPopulation-
>chromosomes[selPair[2*i]].genes[j].gen[k];
                            newPopulation->chromosomes[(2*i)+1].genes[j].gen[k] = pPopulation-
>chromosomes[selPair[(2*i)+1]].genes[j].gen[k];
                        }
                        else{ // Hace la cruza (copia cruzada)
                            newPopulation->chromosomes[2*i].genes[j].gen[k] = pPopulation-
>chromosomes[selPair[(2*i)+1]].genes[j].gen[k];
                            newPopulation->chromosomes[(2*i)+1].genes[j].gen[k] = pPopulation-
>chromosomes[selPair[2*i]].genes[j].gen[k];
                        }
             // Para saber si llegamos al punto de cruza
                        bitCounter++;
                    }
                }
            }
            else{       //No hubo cruza, se pasan los padres igual.
        // Para cada gen
                for(j = 0; j < NUMGENS; j++){
          // Para cada bit
                    for (k = 0; k < BITGEN; k++){
                        newPopulation->chromosomes[2*i].genes[j].gen[k] = pPopulation-
>chromosomes[selPair[2*i]].genes[j].gen[k];
```

```
                                                        newPopulation->chromosomes[(2*i)+1].genes[j].gen[k] = pPopulation-
>chromosomes[selPair[(2*i)+1]].genes[j].gen[k];
                                                }
                                        }
                                }
                        }

   free(selPair);

   // Como se sobreescribe nuestra poblacion original es necesario guardar estos valores manualmente
   newPopulation -> bestChromosome = pPopulation -> bestChromosome;

   for(i = 0; i < NUMCHROMOSOME; i++)
   {
      newPopulation -> chromosomes[i].bestFit = pPopulation -> chromosomes[i].bestFit;
   }

                Free_Population(pPopulation);//Libera poblacion anterior

                return newPopulation;
}

/*
//-------------------------------------------------------Mutar poblacion
   Cambia el valor de un bit por su complemento.

   Inputs:
      Poblacion
   Output:
      void
*/
void Muta_Population(CHROMOSOME *pChromosome){
                int i, j;
                double vMuta; // Valor de mutacion, define si habra o no muta

   // Para cada gen
                for(i = 0; i < NUMGENS; i++){
      // Para cada bit
                        for(j = 0; j < BITGEN; j++){
                                vMuta = (double)rand()/RAND_MAX;
         // Si el valor de muta es menor a la probabilidad de muta hay mutacion
                                if(vMuta <= PMUTA){
            // Calculamos el complemento mediante modulos
                                        pChromosome -> genes[i].gen[j] = (pChromosome -> genes[i].gen[j] + 1) % 2;
                                }
                        }
                }

   return;
}

/*
   Actualiza los mejores fitness de cada cromosoma y de toda la poblacion.
   Inputs:
      La poblacion
   Outputs:
      void
*/
void UpdateBest(POPULATION *pPopulation)
{
   // Actualiza los mejores fitness de los cromosomas
   UpdateChromosomeBest(pPopulation);
   // Actualiza el mejor cromosoma de la poblacion
   UpdatePopulationBest(pPopulation);

   return;
}

/*
   Actualiza los mejores fitness de cada cromosoma
   Inputs:
      La poblacion
   Outputs:
```

```
      void
*/
void UpdateChromosomeBest(POPULATION *pPopulation)
{
 unsigned int i;

 //Para todos los cromosomas
 for(i = 0; i < NUMCHROMOSOME; i++)
 {
  // Si el fitness de actual es mejor que el del historico de ese cromosoma
  if(fabs(pPopulation -> chromosomes[i].chromFit - 10 * V) <= fabs(pPopulation -> chromosomes[i].bestFit - 10 * V))//ERROR)
  {
   pPopulation -> chromosomes[i].bestFit = pPopulation -> chromosomes[i].chromFit;
  }
 }

 return;
}

/*
   Actualiza los mejores fitness de toda la poblacion.
   Inputs:
      La poblacion
   Outputs:
      void
*/
void UpdatePopulationBest(POPULATION *pPopulation)
{
 unsigned int i;
 //Peso del mejor cromosoma de la poblacion
 float best = pPopulation -> chromosomes[pPopulation -> bestChromosome].bestFit;

 for(i = 0; i < NUMCHROMOSOME; i++)
 {
  // Si el mejor fitness actual del cromosoma es mejor que el del mejor de la poblacion
  if(fabs(pPopulation -> chromosomes[i].bestFit - 10 * V) < fabs(best - 10 * V))
  {
   //Se actualiza el id y peso del mejor
   pPopulation -> bestChromosome = i;
   best = pPopulation -> chromosomes[i].chromFit;
  }
 }

 return;
}

/*
   Imprime la poblacion completa y otros datos de interes.
   Imprime el gen en binario y el fitness del cromosoma.
   Inputs:
      La poblacion
   Outputs:
      void
*/
void Show_Population(POPULATION *pPopulation)
{
   uint i, j, k;

   //Para cada cromosoma
   for(i = 0; i < NUMCHROMOSOME; i++)
   {
      printf(" Individuo %3d", i + 1);
      //Para cada gen
      for(j = 0; j < NUMGENS; j++)
      {
         printf("\n\tGen %3d :  ", j + 1);
         //Para cada bit
         for(k = 0; k < BITGEN; k++)
         {
            printf("%d",
               pPopulation -> chromosomes[i].genes[j].gen[k]);
         }
      }
```

```
      printf("\nchromosome's Fitness: %f", pPopulation -> chromosomes[i].chromFit);
      printf("\n-----\n");
   }

   return;
}

/*
   Libera el espacio de memoria que ocupa la poblacion
   Inputs:
      La poblacion
   Outputs:
      void
*/
void Free_Population(POPULATION *pPopulation)
{
   uint i, j;

   for(i = 0; i < NUMCHROMOSOME; i++)
   {
      for(j = 0; j < NUMGENS; j++)
      {
         free(pPopulation -> chromosomes[i].genes[j].gen);
      }
   }

   for(i = 0; i < NUMCHROMOSOME; i++)
   {
      free(pPopulation -> chromosomes[i].genes);
   }

   free(pPopulation -> chromosomes);
   free(pPopulation);

   return;
}
```

Pruebas y resultados

La siguiente tabla muestra los resultados de las pruebas que se hicieron, todas ellas con un error igual a 0.1. El número de cromosomas fue sacado de la teoría de los documentos proporcionados por el profesor[1].

El número de generaciones que se muestran es la mediana (en datos discretos) de los resultados de correr 10 veces el algoritmo bajo cada uno de los parámetros, se hizo de este modo para tener un número más estables, pues a veces las iteraciones eran realmente bajas y ptras veces podian llegas hasta el valor máximo de iteraciones.

Número de cromosomas	Probabilidad de cruza	Probabilidad de mutación	Número de generaciones
10	0,3	0,0	500
		0,003	401
		0,005	312
		0,008	362
	0,5	0,0	500
		0,003	119
		0,005	340
		0,008	301
	0,8	0,0	500
		0,003	144
		0,005	421
		0,008	425
30	0,3	0,0	500
		0,003	465
		0,005	414
		0,008	278
	0,5	0,0	500
		0,003	500
		0,005	327
		0,008	251
	0,8	0,0	500
		0,003	227
		0,005	47
		0,008	254
50	0,3	0,0	500

		0,003	138
		0,005	238
		0,008	128
	0,5	0,0	500
		0,003	360
		0,005	131
		0,008	78
	0,8	0,0	500
		0,003	162
		0,005	136
		0,008	195
100	0,3	0,0	500
		0,003	257
		0,005	165
		0,008	96
	0,5	0,0	500
		0,003	302
		0,005	245
		0,008	103
	0,8	0,0	108
		0,003	54
		0,005	120
		0,008	93
200	0,3	0,0	500
		0,003	107
		0,005	45
		0,008	78
	0,5	0,0	192
		0,003	109
		0,005	49
		0,008	80
	0,8	0,0	187
		0,003	61
		0,005	22
		0,008	47

En la tabla se puede observar fácilmente que entre más cromosomas el número de generaciones necesarias para llegar a un resultado aceptable va decreciendo.

Por mencionar un caso particular, teniendo 10 cromosomas, con una probabilidad de cruza de 0,3 el mejor caso lleva 312 generaciones; en cambio con 200 cromosomas con 0,3 el mejor caso lleva 45 generaciones. Lo cual es un cambio muy radical, pudiendo concluir que entre más cromosomas podemos tener una mejor solución en menos generaciones, sin embargo los cálculos son más tardados por lo que en realidad tardamos más tiempo en obtener el resultado, a pesar de ser menos generaciones. Por lo tanto, debemos escoger entre menos generaciones o menor tiempo, aunque los resultados y la decisión dependen mucho del problema a implementar y del hardware donde se implementará.

Se puede observar también que la probabilidad de mutación influye mucho en el resultado que arroja el algoritmo. A pesar de que la mutación tiene una probabilidad muy baja de ocurrencia, ayuda bastante a que el algoritmo converga, de hecho, si no hay probabilidad de mutación es muy poco probable que se llegue a una solución. Llegamos a pensar que la mutación no influía mucho en el resultado, pero en la etapa de pruebas nos dimos cuenta que en realidad influye y bastante.

En nuestro caso particular, el punto medio entre tiempo y generaciones se logra bastante bien con 50 cromosomas y con una probabilidad de cruza de 0,8 y con una probabilidad de mutación de 0,005. Cabe destacar que estas cifras no serán igual siempre, dependen mucho de la implementación, incluso del SO y del hardware donde se corre el algoritmo.

Conclusión

El algoritmo genético simple, a diferencia del PSO a pesar de empezar con una población aleatoria llega rápidamente a un rango solución y ahí busca la mejor, convergiendo de una manera más rápida. En cambio el PSO, después de un determinado número de iteraciones algo elevado (arriba de 200) y con un gran número de partículas tendía a explotar rápidamente, por lo que en la implementación se debían hacer varios ajustes para regresarlo al espacio de búsqueda deseado.

De igual forma, a pesar de que la teoría es algo más compleja, considero que la implementación es mucho más sencilla e intuitiva, de hecho resulta en funciones bastante simples. Claro está que es un algoritmo genético sencillo, pero funciona bastante bien (en nuestro caso particular funcionó mucho mejor que el PSO).

Es importante remarcar que no se puede decir que hay “mejores algoritmos”, cada algoritmo tiene sus ventajas y desventajas (del AG se han mencionado en la introducción), además que se aplican en situaciones diferentes y con ajustes diferentes para cada implementación. Puedo decir que los algoritmos genéticos, por las lecturas que realicé, se utilizan cuando se desconoce por completo la solución o sus límites. En este caso conocíamos ambos, porque es una implementación menos sofisticada pero que ilustra bien la escencia de un algoritmo genético.

Algo sorprendente de este tipo de algoritmos es que, a pesar de que dependen mucho de probabilidades y de números aleatorios, los resultados que arroja son bastantes buenos. Claro está que cada vez el algitmo va convergiendo porque va evolucionando hacia mejores soluciones, pero incluso la cruza y la mutación son aleatorias.

Como se puede observar de lo que llevamos del curso es que no hay una forma teórica de saber los mejores parámetros, todo siempre se obtiene de manera práctica y depende de cada implementación en particular, por lo que el periodo de prueba y experimentación resulta especialmente importante.

Referencias y bibliografía

[1] “An experimental study of benchmarking functions for Genetic Algorithms”, Jason G. Dgalakis, Konstantinos G. Margaritis.

[2] “Notas de Reconocimiento de Patrones”, Camarena Ibarrola, José Antonio. Universidad Michoacana de San Nicolás de Hidalgo, 2009.

[3] “Proceeding of the Fourth International Conference on Genetis Algorithms”, M.F, Bramlette. 1991

[4] “Un algoritmo genético simple para la optimización de estructuras planas articuladas”, Galante, Miguel. Revista Internacional de Métodos Numéricos para Cálculo y Diseño en Ingeniería, 1993.