

Introducción a la Computación Evolutiva

Tarea No. 2

Instructor: Dr. Carlos Artemio Coello Coello

Alumno: Ángel Alonso Galarza Chávez

23 de mayo de 2025

Algoritmo genetico

Un Algoritmo Genético es una técnica de optimización inspirada en los principios de la evolución biológica. Basada en el proceso del Neo-Darwinismo que sigue los mecanismos como la reproducción, cruza y mutación para evolucionar una población a través de generaciones, donde una población contiene individuos que representan las soluciones [1].

Un algoritmo genético consta de las siguientes etapas:

1. **Inicialización:** Se genera una población inicial de individuos (soluciones) de forma aleatoria.
2. **Evaluación:** Cada individuo es evaluado mediante una función de aptitud (*fitness*) que mide la calidad de la solución que representa.
3. **Selección:** Se eligen individuos basándose en su aptitud para participar en la reproducción. Los individuos con mayor aptitud tienen mayor probabilidad de ser seleccionados.
4. **Cruza:** Los individuos seleccionados se combinan para formar nuevos individuos, intercambiando partes de su estructura genética.
5. **Mutación:** Se introducen cambios aleatorios en algunos individuos para mantener la diversidad genética de la población.

Esta técnica permite evolucionar las soluciones de individuos por medio de generaciones.

El problema se presentan a continuación en la ecuación 1 donde se requiere de minimizar la siguiente función.

$$f(x_1, x_2, x_3, x_4) = 10(x_2 - x_1^2)^2 + (1 - x_1)^2 + 90(x_4 - x_3^2)^2 + (1 - x_3)^2 + 10(x_2 + x_4 - 2)^2 + 0,1(x_2 - x_4)^2 \quad (1)$$

Para $j = 1, \dots, 4$:

$$-20,0 \leq x_j \leq 20,0$$

Código

En esta sección se presenta el código fuente correspondiente a la implementación de un algoritmo genético. Cada componente del algoritmo desde la representación de los individuos hasta las funciones como decodificación, evaluación (función objetivo), selección, cruce y mutación está descrito mediante funciones en lenguaje C.

Decodificación

La función *decode* transforma una cadena binaria (cromosoma) en un valor decimal dentro de un rango especificado mediante una transformación lineal. Este valor se interpreta como una variable del problema. La función *decode_all_vars* aplica esta transformación a cada una de las cuatro variables codificadas en el cromosoma del individuo.

Mapeo binario–decimal

Para cada variable $x_j \in [-20, 20]$ para $j \in [1, 4]$ se requiere una precisión de 10^{-10} . El número mínimo de bits n que permite representar cualquier valor del intervalo con dicha precisión se obtiene con

$$n = \left\lceil \log_2((\max - \min) \cdot 10^{10}) \right\rceil = \left\lceil \log_2(40 \times 10^{10}) \right\rceil = \lceil 38,576 \rceil = 39 \text{ bits.}$$

Como la función objetivo posee 4 variables, la longitud total del cromosoma es

$$codesize = n \cdot k = 39 \times 4 = 156 \text{ bits.}$$

La codificación de binario a decimal se puede emplear de la siguiente manera:

$$x = \min + (\max - \min) \cdot \frac{I}{2^n - 1}$$

y luego se codifica I como una cadena binaria de n bits que se escribe en orden de bit menos significativo primero.

```

1  /* Decode the chromosome string into its corresponding decimal value */
2  double decode(unsigned *chrom, int start, int length, double min, double
    max) {
3      double accum = 0.0;
4      for (int j = 0; j < length; j++) {
5          if (chrom[start + j] == 1) {
6              accum += pow(2.0, (double)j);
7          }
8      }
9
10     // AJUSTE DE LOS VALORES AL RANGO [-20, 20] USANDO LA ECUACION DE LA
        RECTA
11     double val = min + (max - min) * (accum / (pow(2.0, (double)length) -
        1.0));
12     return val;
13 }
14
15 // FUNCION AUXILIAR PARA DECODIFICAR LAS 4 VARIABLES
16 void decode_all_vars(struct individual *ind, double min, double max) {
17     ind->x1 = decode(ind->chrom, 0 * VAR_LENGTH, VAR_LENGTH, min, max);
18     ind->x2 = decode(ind->chrom, 1 * VAR_LENGTH, VAR_LENGTH, min, max);
19     ind->x3 = decode(ind->chrom, 2 * VAR_LENGTH, VAR_LENGTH, min, max);
20     ind->x4 = decode(ind->chrom, 3 * VAR_LENGTH, VAR_LENGTH, min, max);
21 }

```

Estructura del individuo

La representación de cada individuo en el algoritmo genético se implementa mediante la siguiente estructura en C.

```

1  struct individual {
2      unsigned *chrom; /*      chromosome string for the individual */
3      double x1;      /*      value of the decoded string */
4      double x2;      /*      value of the decoded string */
5      double x3;      /*      value of the decoded string */
6      double x4;      /*      value of the decoded string */
7      double fitness; /*      fitness of the individual */
8      int xsite1;     /*      crossover site 1 at mating */
9      int xsite2;     /*      crossover site 2 at mating */
10     int placemut;    /*      mutation place */
11     int mutation;    /*      mutation indicator */
12     int parent[2];   /*      who the parents of offspring were */
13 };

```

Función Objetivo

La función *objfunc* evalúa la calidad (fitness) de cada individuo mediante una fórmula basada en varias combinaciones cuadráticas de las variables decodificadas. El objetivo es minimizar,

es por eso que se realiza una transformación del resultado a través de una relación inversa para calcular la aptitud. Con esta operación, el individuo que consiga un valor mínimo de la función objetivo tendrá un fitness mas alto.

```

1 // FUNCION OBJETIVO ->
2 void objfunc(struct individual *ind) {
3
4     double term1 = pow(10.0 * (ind->x2 - ind->x1 * ind->x1), 2);
5     double term2 = pow(1.0 - ind->x1, 2);
6     double term3 = 90.0 * pow(ind->x4 - ind->x3 * ind->x3, 2);
7     double term4 = pow(1.0 - ind->x3, 2);
8     double term5 = 10.0 * pow(ind->x2 + ind->x4 - 2.0, 2);
9     double term6 = 0.1 * pow(ind->x2 - ind->x4, 2);
10    ind->fitness = 0.0;
11
12    // MINIMIZAR
13    ind->fitness = 1.0 / (1.0 + term1 + term2 + term3 + term4 + term5 +
14                        term6);
15 }

```

Selección

La etapa de selección consiste en elegir a los individuos de la población actual para la etapa de reproducción. Esta selección se basa en una función de aptitud (fitness) que evalúa la calidad de cada individuo.

La selección consiste en elegir los individuos que participarán en la reproducción. Se emplea el algoritmo de la ruleta [1], donde cada individuo tiene una probabilidad de ser seleccionado proporcional a su aptitud relativa. Se calcula el valor esperado de cada individuo con base en la media de aptitud, y se selecciona uno aleatoriamente según su contribución acumulada.

Algorithm 1 Algoritmo de la Ruleta

Dada la distribución acumulativa de probabilidad a

Suponiendo que se desea seleccionar λ individuos para el *mating pool*

$current_member \leftarrow 1$

while $current_member \leq \lambda$ **do**

 Elegir un valor aleatorio r uniformemente de $[0, 1]$

$i \leftarrow 1$

while $a_i < r$ **do**

$i \leftarrow i + 1$

end while

$mating_pool[current_member] \leftarrow parents[i]$

$current_member \leftarrow current_member + 1$

end while

Como primer paso, debemos de calcular la sumatoria total de la aptitud de la poblacion *sum_fitness* y su media *med*, despues se hace la division de $avg_fit[i] += ind[i].fitness / med$ para obtener el valor esperado de cada inividuo.

En la funcion de *ruleta* se pasan los parametros de *sum_fitness* y el arreglo de los valores esperados de cada inividuo de la poblacion *avg_fit* y seleccionando un numero pseudoaleatorio *r* del rango $r \in [0,0, popsize]$ sumara los valores esperados de cada inividuo hasta que supere o iguale a *r*, de esta manera se selecciona el inividuo y retorna su indice.

```

1 // FUNCION PARA CALCULAR LA MEDIA DEL FITNESS DE LA POBLACION
2 double f_med(struct individual *ind, double *avg_fit) {
3     double med = 0.0;
4     double sum_fitness = 0.0;
5
6     for (int i = 0; i < popsize; i++)
7         avg_fit[i] = 0.0;
8
9     for (int i = 0; i < popsize; i++) {
10         sum_fitness += ind[i].fitness;
11     }
12     med = sum_fitness / popsize;
13
14     for (int i = 0; i < popsize; i++) {
15         avg_fit[i] += ind[i].fitness / med;
16     }
17
18     return sum_fitness;
19 }
20
21 // FUNCION DE SELECCION POR RULETA
22 int ruleta(double sumfitness, double *avg_fit) {
23     float r = rndreal(0.0, popsize);
24     double accum = 0.0;
25     int i = 0;
26     do {
27         accum += avg_fit[i];
28         i++;
29     } while (accum < r && i < popsize);
30
31     return i - 1;
32 }

```

Cruza

La crusa, o recombinación, permite generar nuevos individuos combinando partes del código genético de dos padres. En este caso, se implementa la crusa de un punto: se elige aleatoriamente un punto de corte en el cromosoma, y se intercambian las secciones de los padres para formar los hijos. Este proceso permite explorar nuevas combinaciones en el espacio de

soluciones [1].

A continuación se hizo la implementación de la cruce de un punto, el funcionamiento es elegir un numero j en un rango de $j \in [1, L - 1]$ donde L es la longitud del cromosoma del individuo, y se divide a ambos padres en el punto j para crear a los hijos, intercambiando la parte izquierda y derecha de los padres a sus hijos (con eso se obtiene a hijos con diferentes partes de los padres).

```
1 /* Cross 2 parent strings, place in 2 child strings */
2 // FUNCION DE CRUZA MODIFICADA PARA HACER LA CRUZA DE UN PUNTO
3 void crossover(unsigned *parent1, unsigned *parent2, unsigned *child1,
4               unsigned *child2, int *jcross) {
5
6     int j = 0;
7     if (flip(Pc)) {
8         *jcross = rnd(1, codesize - 1);
9         ncross++;
10
11         // PARTE 1 DEL PADRE_1 AL HIJO_1 Y PARTE 1 DEL PADRE_2 AL HIJO_2
12         for (j = 0; j < *jcross; j++) {
13             child1[j] = parent1[j];
14             child2[j] = parent2[j];
15         }
16
17         // PARTE 2 DEL PADRE_2 AL HIJO_1 Y PARTE 2 DEL PADRE_1 AL HIJO_2
18         for (j = *jcross; j < codesize; j++) {
19             child1[j] = parent2[j];
20             child2[j] = parent1[j];
21         }
22     } else {
23         // SE COPIAN LOS VALORES DE LOS PADRES A LOS HIJOS SIN
24         // MODIFICACIONES
25         for (j = 0; j < codesize; j++) {
26             child1[j] = parent1[j];
27             child2[j] = parent2[j];
28         }
29         *jcross = 0;
30     }
```

Mutación

La mutación introduce variaciones aleatorias con una baja probabilidad en los individuos, modificando uno o más genes. El propósito es mantener la diversidad en la población, evitando la convergencia prematura a soluciones locales [1].

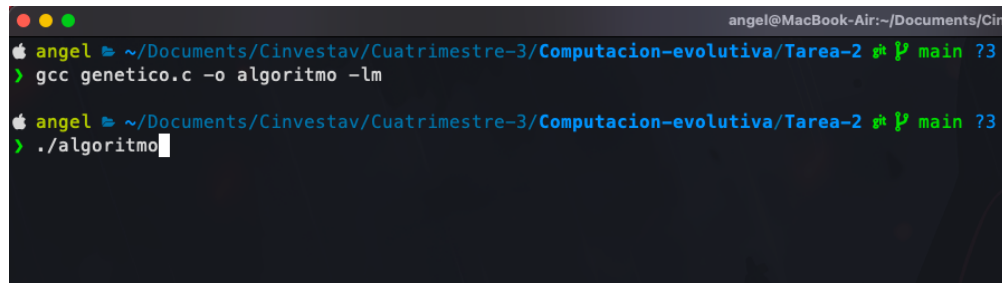
La siguiente función *mutation* se implemento la mutación más común para codificaciones binarias que considera cada gen por separado e invierte el bit con una pequeña probabilidad

P_m . Con esto se asegura que la cantidad de valores que cambian no es fija. Así, para una codificación de longitud L , en promedio se cambiarán $L \cdot p_m$ valores.

```
1 // FUNCION DE MUTACION MODIFICADA PARA INTERCAMBIAR EL BIT A LO LARGO DE
  LA
2 // CADENA SI SE CUMPLE UNA PROBABILIDAD (Pm)
3 void mutation(struct individual *ind) {
4     // MUTACION POR BIT EN TODA LA CADENA
5
6     for (int i = 0; i < codesize; i++) {
7         if (flip(Pm)) {
8             if (ind->chrom[i] == 0) {
9                 ind->chrom[i] = 1;
10            } else {
11                ind->chrom[i] = 0;
12            }
13            nmutation++;
14        }
15    }
16 }
```

Ejecución del Algoritmo

Para la compilación y ejecución del sistema se requieren los archivos fuente en C (.c) y sus correspondientes cabeceras (.h), donde se implementan las estructuras de datos y funciones principales. El proceso de compilación se realiza mediante el siguiente comando en terminal:



```
angel@MacBook-Air:~/Documents/Cin...
angel ~ ~/Documents/Cinvestav/Cuatrimestre-3/Computacion-evolutiva/Tarea-2 # main ?3
> gcc genetico.c -o algoritmo -lm

angel ~ ~/Documents/Cinvestav/Cuatrimestre-3/Computacion-evolutiva/Tarea-2 # main ?3
> ./algoritmo
```

Figura 1: Proceso de compilación del código fuente mediante GCC

Configuración de Parámetros

Durante la ejecución, el algoritmo solicita interactivamente los siguientes parámetros de configuración:

- Tasa de mutación (*mutation rate*)
- Probabilidad de cruce (*probability of crossover*)
- Número de generaciones (*number of generations*)

- Semilla para números aleatorios (*random number seed*)

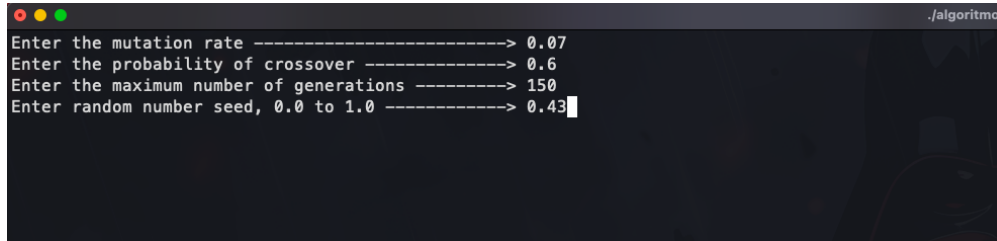


Figura 2: Interfaz de entrada de parámetros del algoritmo genético

Salida de Resultados

Al finalizar la ejecución, el programa genera:

1. Un resumen detallado con:
 - Parámetros de configuración utilizados
 - Estadísticas por generación
 - Mejor solución encontrada (individuo óptimo global)
2. Un archivo de datos (**convergencia.txt**) que registra el fitness del mejor individuo en cada generación, utilizado posteriormente para generar la gráfica de convergencia mostrada en la Figura 3.

Parámetro	Valor
Total population size	100
Chromosome length	156
Maximum number of generations	150
Crossover probability	0.600000
Mutation probability	0.070000

Gen	Media	Máxima	Mínima	Cruzas	Bits mutados
0	0.000037	0.000387	0.000000	23	1048
1	0.000062	0.000779	0.000000	55	2124
2	0.000096	0.000909	0.000000	75	3219
⋮	⋮	⋮	⋮	⋮	⋮
148	0.014117	0.104779	0.000000	4373	136922
149	0.016118	0.104779	0.000000	4405	137780

Mejor solución encontrada (generación 124)
Cadena binaria: 10001001111010111111111100011110100001010111100000100101000110110110000110 010100111111111111101101000110101111000100100011111111011011000101111011011101111110 Valores reales: $x_1 = -0,045053$, $x_2 = -0,002883$, $x_3 = -1,227352$, $x_4 = 1,538082$ Fitness: 0.104779 $f(x) = 8,543886$

Estadísticas

En esta sección se presentan los resultados de 20 ejecuciones independientes del algoritmo, junto con el análisis de la gráfica de convergencia y la exploración del espacio de búsqueda.

Las estadísticas obtenidas revelan un problema en la ejecución del algoritmo. En primer lugar, se observa una reproducibilidad exacta de resultados, donde independientemente de la semilla aleatoria utilizada, todas las ejecuciones convergen a idénticos valores finales ($f(x) = 3,152396$, $Fitness = 0,240825$). Este se acompaña de una completa falta de variabilidad, evidenciada por una varianza nula tanto en los valores de fitness como en $f(x)$ a través de todas las corridas realizadas.

Cuadro 1: Resultados de las 20 corridas con parámetros x_1 a x_4 , fitness y $f(x)$

Corrida	Semilla	x_1	x_2	x_3	x_4	Fitness	$f(x)$
1	0.1083	-0.022639	-0.001358	1.247814	1.610554	0.240825	3.152396
2	0.0132	-0.022639	-0.001358	1.247814	1.610554	0.240825	3.152396
3	0.1949	-0.022639	-0.001358	1.247814	1.610554	0.240825	3.152396
4	0.3766	-0.022639	-0.001358	1.247814	1.610554	0.240825	3.152396
5	0.5583	-0.022639	-0.001358	1.247814	1.610554	0.240825	3.152396
6	0.7401	-0.022639	-0.001358	1.247814	1.610554	0.240825	3.152396
7	0.9218	-0.022639	-0.001358	1.247814	1.610554	0.240825	3.152396
8	0.1035	-0.022639	-0.001358	1.247814	1.610554	0.240825	3.152396
9	0.2852	-0.022639	-0.001358	1.247814	1.610554	0.240825	3.152396
10	0.4670	-0.022639	-0.001358	1.247814	1.610554	0.240825	3.152396
11	0.6487	-0.022639	-0.001358	1.247814	1.610554	0.240825	3.152396
12	0.8304	-0.022639	-0.001358	1.247814	1.610554	0.240825	3.152396
13	0.0121	-0.022639	-0.001358	1.247814	1.610554	0.240825	3.152396
14	0.2847	-0.022639	-0.001358	1.247814	1.610554	0.240825	3.152396
15	0.8735	-0.022639	-0.001358	1.247814	1.610554	0.240825	3.152396
16	0.7390	-0.022639	-0.001358	1.247814	1.610554	0.240825	3.152396
17	0.3278	-0.022639	-0.001358	1.247814	1.610554	0.240825	3.152396
18	0.1933	-0.022639	-0.001358	1.247814	1.610554	0.240825	3.152396
19	0.7821	-0.022639	-0.001358	1.247814	1.610554	0.240825	3.152396
20	0.0547	-0.022639	-0.001358	1.247814	1.610554	0.240825	3.152396

Estadísticas de los valores de Fitness y $f(x)$:

Media Fitness (μ): 0,240825

Varianza Fitness (σ^2): 0,0

Desviación estándar Fitness (σ): 0,0

Media $f(x)$: 3,152396

Varianza $f(x)$: 0,0

Desviación estándar $f(x)$: 0,0

Mejor y peor valor $f(x)$: 3,152396 (todas las corridas)

Gráfica de convergencia

En la Figura 3 se muestra la gráfica de convergencia del algoritmo genético, donde se analiza la evolución del *fitness* del mejor individuo en cada generación. Se identifican tres fases claramente diferenciadas:

1. **Fase inicial (generaciones 0–30):** Se observan mejoras significativas en el *fitness*, con incrementos notables en cortos intervalos generacionales.
2. **Fase de estancamiento (generaciones 30–110):** El *fitness* se mantiene aproximadamente constante durante unas 80 generaciones, lo que sugiere que el algoritmo ha encontrado un óptimo local del que le resulta difícil escapar.

3. **Fase de reactivación (generación 142 en adelante):** Se produce un nuevo salto significativo en el *fitness*, seguido de otra etapa de estabilización.

La figura 3 evidencia la implementación del mecanismo de elitismo, que garantiza la preservación del mejor individuo entre generaciones. Sin embargo, los prolongados periodos sin mejora indican una posible pérdida de diversidad genética en la población.

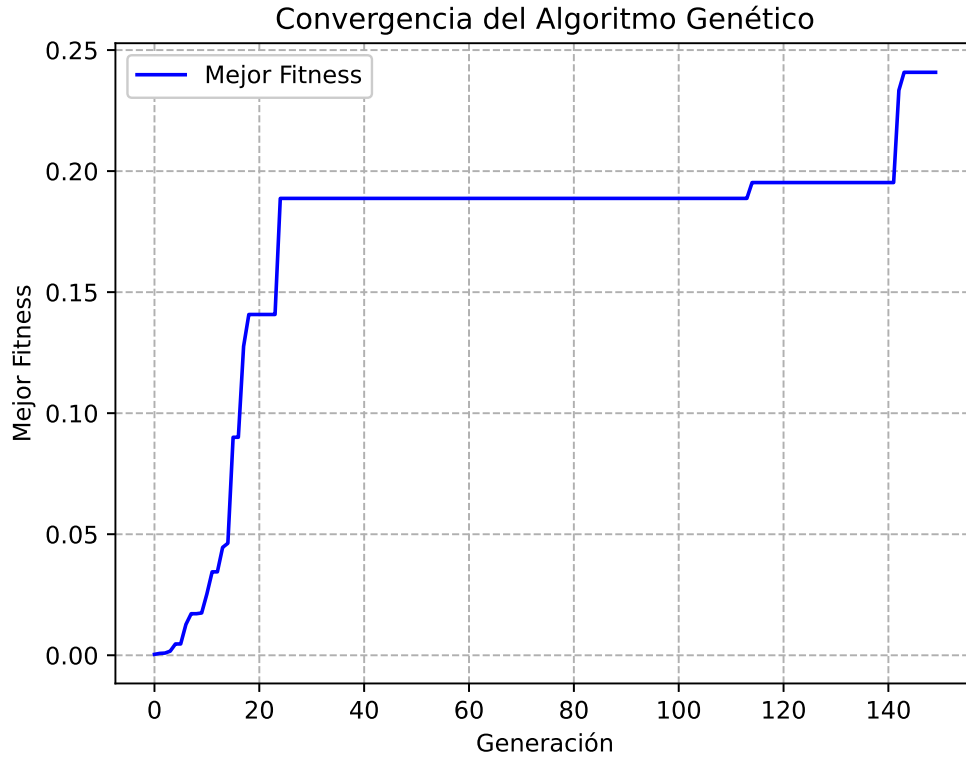


Figura 3: Gráfica de convergencia

Tamaño de exploración del problema

Cada individuo está representado por una cadena binaria de 156 bits, dividida en 4 variables de 39 bits cada una.

Para cada variable x_j ($j = 1, \dots, 4$):

$$\text{Número de valores posibles por variable} = 2^{39}$$

Como las variables son independientes, el número total de combinaciones es:

$$\text{Espacio de búsqueda} = 2^{39} \times 2^{39} \times 2^{39} \times 2^{39} = 2^{156}$$

Aproximando $2^{10} \approx 10^3$, entonces:

$$2^{156} \approx 10^{47} \text{ soluciones posibles}$$

Los parametros utilizados para el algoritmo genetico son los siguientes:

- Tamaño de la población: 100 individuos
- Número de generaciones: 150
- Total de individuos evaluados:

$$100 \times 150 = 15,000 \text{ individuos}$$

$$\text{Porcentaje explorado} = \left(\frac{15,000}{2^{156}} \right) \times 100 \approx \left(\frac{15,000}{10^{47}} \right) \times 100 \approx 1,5 \times 10^{-44} \%$$

El algoritmo genético explora una fracción pequeña del espacio de búsqueda ($\sim 10^{-44} \%$). Esto es suficiente para encontrar soluciones aceptables gracias a los mecanismos de selección, cruza y mutación, sin necesidad de enumerar todas las posibilidades.

Referencias

- [1] A E Eiben y J E Smith. *Introduction to evolutionary computing*. en. 2.^a ed. Natural Computing Series. Berlin, Germany: Springer, jul. de 2015.