Tarea 4 - Algoritmo Genético

1. Ejercicio 1. Optimización continua

   (a) Describe e implementa un algoritmo genético para las funciones de optimización continua utilizadas en la tarea 3. ** Se deben implementar y utilizar los siguientes componentes:

   i) Representación binaria para las soluciones

   La representación binaria para soluciones continuas ya ha sido explicada e implementada en en las 2 tareas anteriores, así que en esta ya no lo abordaremos, simplemente reutilizaremos las implementaciones previamente hechas.

   Si se desea saber el funcionamiento y la implementación se encuentra en nuestro reporte de la tarea 2, donde está el algoritmo detallado.

   ii) Selección de padres por el método de la ruleta

   La ruleta vista en clase funcionaba para maximización, pero en este caso queríamos aplicarla para minimización, de modo que los individuos con menor aptitud tuvieran una mayor probabilidad.

   Para lograr esto la aptitud de cada individuo j la vimos como

   $$\text{apt}(j) = \frac{1}{\text{evaluación}(j)}$$

   De modo que la aptitud total es:

   $$f = \Sigma_{j=1}^{tam} \quad \text{apt}(j)$$

   Para calcular la probabilidad de cada individuo j se hace de la siguiente forma:

   $$\text{proba}(j) = \frac{\text{apt}(j)}{f}$$

   y por último los cuantiles se calculan de la siguiente forma:

   $$\text{probaAcumulada}(j) = \Sigma_{j=1}^{tam}$$

   De esta forma implementamos la ruleta de selección de padres para minimización.

   De tal forma que su algoritmo implementado queda de la siguiente forma:

Algorithm 1 Selección por Ruleta

```
1: function RULETA
2:     f ← 0
3:     for i ← 0 to tam − 1 do
4:         f ← f + 1/evaluaciones[i]
5:     end for
6:     arreglo proba[tam]
7:     for i ← 0 to tam − 1 do
8:         proba[i] ← (1/evaluaciones[i]) / f
9:     end for
10:    r ←numero aleatorio entre 0 y 1
11:    probaAcumulada ← 0
12:    for i ← 0 to tam − 1 do
13:        probaAcumulada ← probaAcumulada + proba[i]
14:        if r < probaAcumulada then
15:            return i
16:        end if
17:    end for
18: end function
```

De la línea 2 a 5 calculamos la aptitud total, de la línea 6 a 9 calculamos la probabilidad de selección de cada individuo, de la línea 10 a 17 giramos la ruleta y vemos en qué cuantil quedó.

Adicionalmente en el código java pide que tengas un retorno fuera de condicionales, entonces agregamos el índice del último elemento aunque nunca debería llegar a esa línea de retorno ya que el número aleatorio generado con la biblioteca Math.random excluye al 1, aún así podría ocurrir algún error de redondeo. En cualquiera de los casos el elemento que se debe seleccionar es al que le corresponde el último cuantil

iii) Operador de cruza de n puntos

Para el operador de cruza en n puntos primero mostraremos un ejemplo de cómo funciona la implementación.

Teniendo los dos arreglos que son los padres con índices de 0 a tamaño del arreglo (en este caso tam=6), generamos un número aleatorio en ese rango En este caso supongamos que generamos el número 2.

0	1	2	3	4	5
a_0	a_1	a_2	a_3	a_4	a_5
b_0	b_1	b_2	b_3	b_4	b_5

Figure 1: Padres con índice 2 seleccionado

Para saber de qué padre va a ir heredando cada hijo, lo haremos intercalado, por cada punto de cruza el hijo va a ir cambiando de padre para recibir sus valores.

Teniendo este primer punto empezaremos a generar a los hijos, guardando en el hijo 1 los valores del padre a desde la casilla 0 hasta la casilla índice menos uno $(2 - 1 = 1)$, para el hijo 2 será lo mismo pero va a recibir del padre b

0	1	2	3	4	5
a_0	a_1				
b_0	b_1				

Figure 2: Hijos primer punto

Para la siguiente iteración ahora generamos un número aleatorio entre el primer punto que valía 2 y el tamaño de arreglo (que vale 6).

Supongamos que al generar ese entero en el intervalo $[2, 6]$ generamos el 4.

Ahora el índice en los padres va a apuntar en la casilla 4:

0	1	2	3	4	5
a_0	a_1	a_2	a_3	a_4	a_5
b_0	b_1	b_2	b_3	b_4	b_5

Figure 3: Padres con índice 4 seleccionado

Como en la iteración anterior el hijo 1 recibía del padre a, ahora recibirá del padre b, lo mismo aplica para el hijo dos.

Ahora en el hijo 1 guardaremos los valores del padre b desde el índice anterior 2 hasta el índice actual menos uno $(4 - 1 = 3)$ y se hará lo mismo apara el hijo 2 pero recibiendo del padre a

0	1	2	3	4	5
a_0	a_1	b_2	b_3		
b_0	b_1	a_2	a_3		

Figure 4: Hijos segundo punto de cruza

Ahora con este nuevo índice, generamos un número aleatorio entre el índice y el tamaño del arreglo, generamos un número aleatorio entre 4 y 6.

Supongamos que se genera el número 5, entonces apuntamos en los padres hacia el índice 5.

0	1	2	3	4	5
a_0	a_1	a_2	a_3	a_4	a_5
b_0	b_1	b_2	b_3	b_4	b_5

Figure 5: Padres con índice 5 seleccionado

Nuevamente notamos que en la iteración anterior el hijo 1 recibía del padre b entonces en esta nueva iteración recibirá los datos del padre a, lo mismo para el hijo 2 pero al revés.

El hijo 1 recibirá los datos del padre a desde la casilla del índice anterior hasta la casilla del índice actual menos uno.

Lo mismo para el hijo 2, pero recibiendo del padre b.

En este caso reciben valores en los índices 4 a $5 - 1 = 4$

0	1	2	3	4	5
a_0	a_1	b_2	b_3	a_4	
b_0	b_1	a_2	a_3	b_4	

Figure 6: Hijos tercer punto de cruza

Para la siguiente iteración generamos un número entre 5 y 6. Se generará el 6 porque en la implementación si se genera el 5 simplemente hacemos un autoincremento..

0	1	2	3	4	5
a_0	a_1	b_2	b_3	a_4	b_5
b_0	b_1	a_2	a_3	b_4	a_5

Figure 7: Hijos cuarto punto de cruza

Repetimos lo mismo del anterior pero cambiando el padre del que reciben.

Y así nos quedaría la cruza de los padres a y b, la cual genera dos hijos. En este caso la cruza en n puntos fueron 4 puntos aunque realmente son 3 porque el cuarto punto ya fue el final del arreglo.

La implementación también usa una probabilidad para saber si cruzarlo o no, pero lo que vamos a poner es solo el algoritmo para cruzar en n puntos, no el algoritmo completo con la probabilidad de cruza.

Dicho esto el algoritmo para cruzar dos individuos con n puntos es el siguiente:

Algorithm 2 Cruza en N Puntos

```
1: function CRUZARN(padre1, padre2, probCruza)
2:     hijos ← arreglo de enteros de tamaño 2 × longitud(padres)
3:     indActual ← 0
4:     intercambiador ← 0
5:     tamPadre ← padre1.length
6:     while indActual < tamPadre do
7:         salto ← número aleatorio entre (indActual) y (tamPadre)
8:         if salto == indActual then
9:             salto ← salto + 1
10:        end if
11:        for i ← indActual to salto − 1 do
12:            hijos[intercambiador][i] ← padre1[i]
13:            hijos[1 − intercambiador][i] ← padre2[i]
14:        end for
15:        intercambiador ← 1 − intercambiador
16:        indActual ← salto
17:    end while
18:    return hijos
19: end function
```

El intercambiador es lo que nos asegura que se vaya cambiando de padre, ya que en cada iteración del while le vamos a hacer un flip al valor binario (de 1 a 0 o de 0 a 1).

iv) Mutación flip Como en el componente anterior, la implementación completa usa una probabilidad, pero esta probabilidad solo nos dice si se aplica la mutación o no a cada uno de los individuos.

Omitiremos esa parte e iremos directo al algoritmo e implementación de la mutación flip.

Algorithm 3 Mutación Flip

```
1: function MUTARFLIP(individuo j)
2:     for i ← 0 to longitud(individuo j) − 1 do
3:         individuo j[i] ← 1 − individuo j[i]
4:     end for
5: end function
```

v) Reemplazo generacional con elitismo [Se debe garantizar que la mejor solución siempre permanece en la población]

En nuestra implementación al realizar la evaluación en cada iteración guardamos al mejor individuo en sus representación binaria. Entonces antes de evaluar de nuevo, después de realizar la cruza y la mutación, aún tenemos guardado el mejor individuo de la generación anterior.

Es por eso que nuestro algoritmo toma al mejor individuo ya guardado como atributo y lo agrega en una casilla aleatoria del arreglo de individuos de la población.

Algorithm 4 Elitismo

```
1: function ELITE
2:     mejorIndividuo ← atributo mejor individuo
3:     r ← número entero aleatorio en intervalo [0,tam)
4:     //individuos es el atributo que guarda los individuos de la población
5:     individuos[r] ← mejorIndividuo
6: end function
```

(b) Genera al menos una gráfica de evolución de aptitud ***, con una ejecución para cada una de las funciones de prueba.

Debido a que el algoritmo estaba convergiendo muy rápido, las mejores soluciones no eran tan buenas y no era una comparación justa con los otros métodos en cuanto a costo computacional. Para hacer más justa la comparación, aumentamos la población y disminuimos las iteraciones.

El número de generaciones será de 1000, el tamaño de población que usaremos para todas las ejecuciones es de 200, la probabilidad de cruza será 1 para no generar clones tan rápido y la probabilidad de mutación será de $\frac{2}{\text{tam de población}}$ para asegurar que al menos dos de esos individuos son mutados.

La dimensión de las funciones a optimizar, será de 10 como en la tarea anterior, para poder comprar resultados.

Adicionalmente de la aptitud, agregamos el promedio para asegurarnos que nuestra algoritmo se comportaba de la forma correcta.

Sphere:

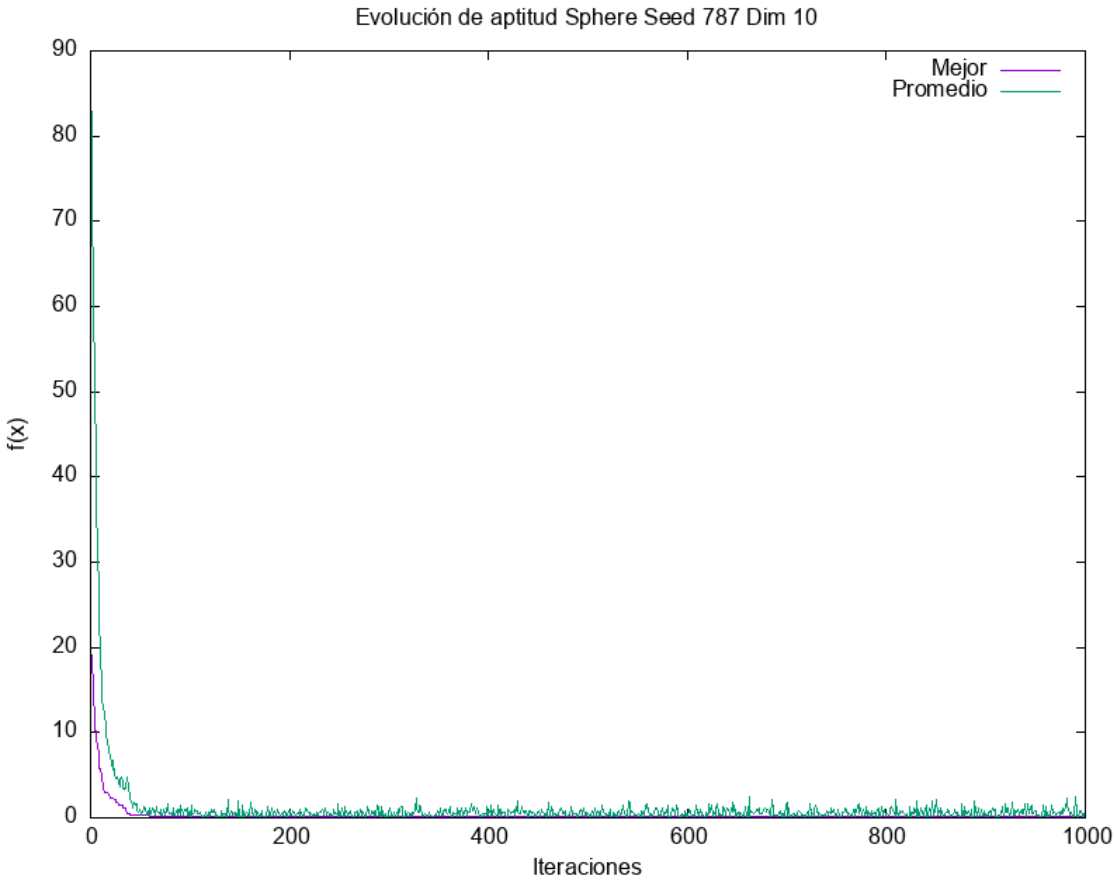

Figure 8: Evolución de aptitud

Ackley:

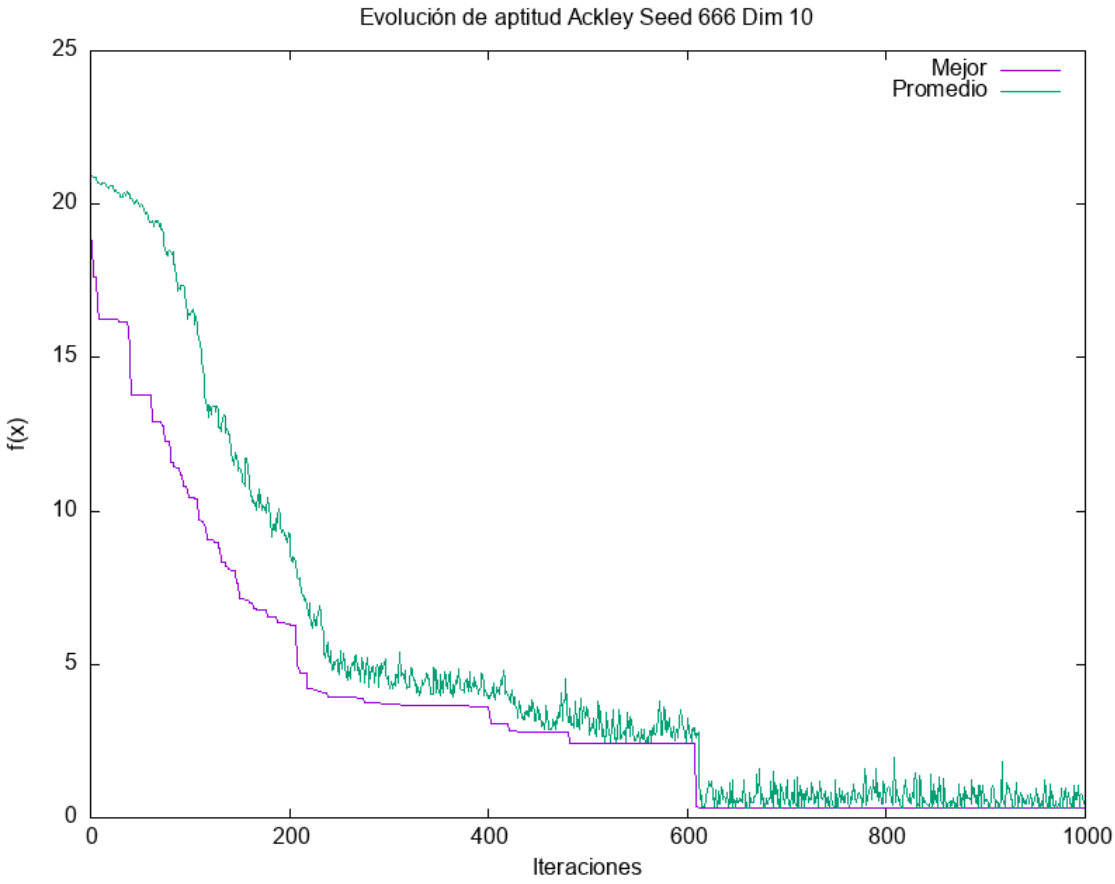

Figure 9: Evolución de aptitud

Griewank:

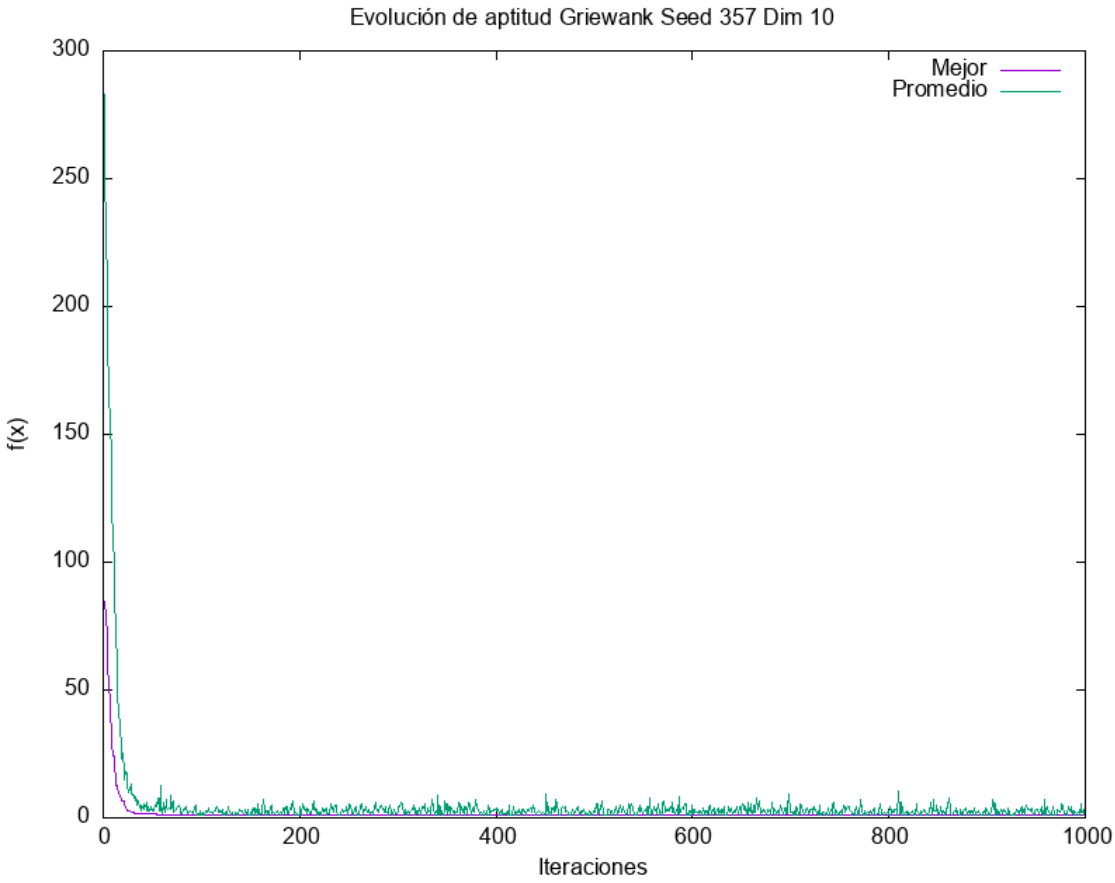

Figure 10: Evolución de aptitud

Rastrigin:

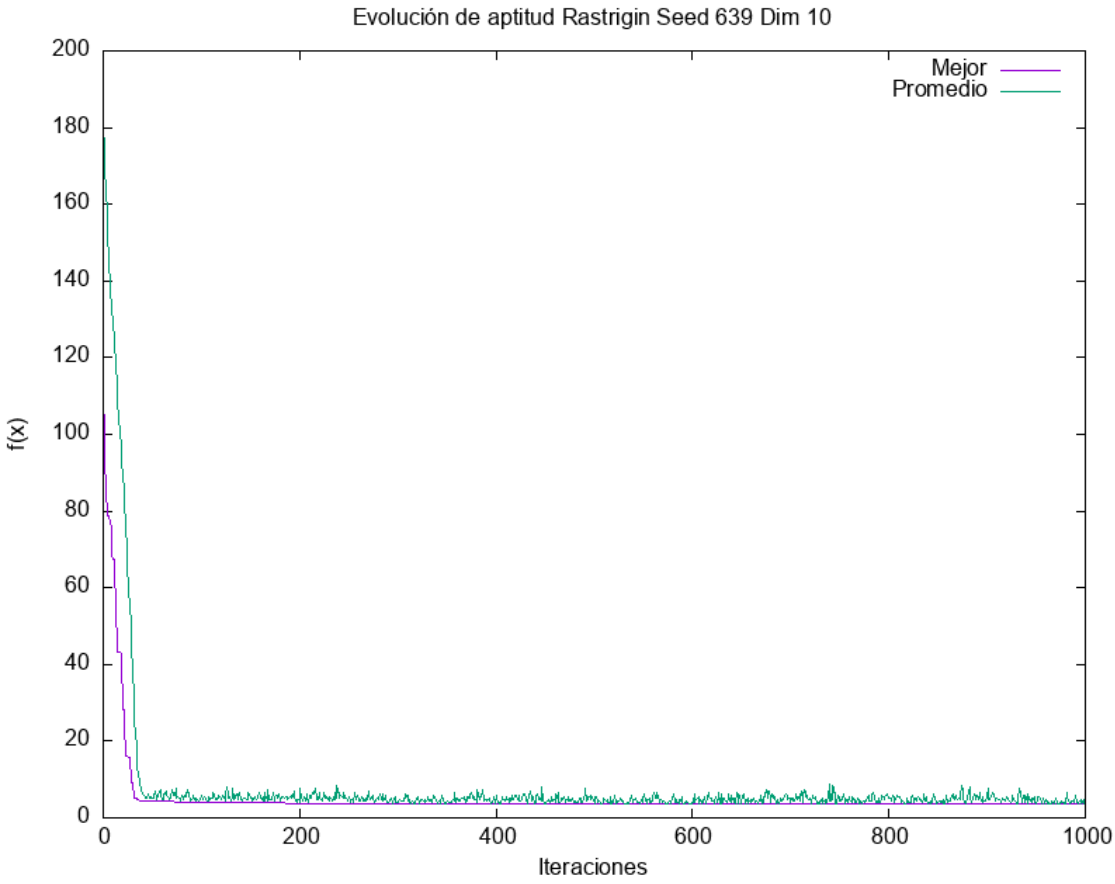

Figure 11: Evolución de aptitud

Rosenbrock:

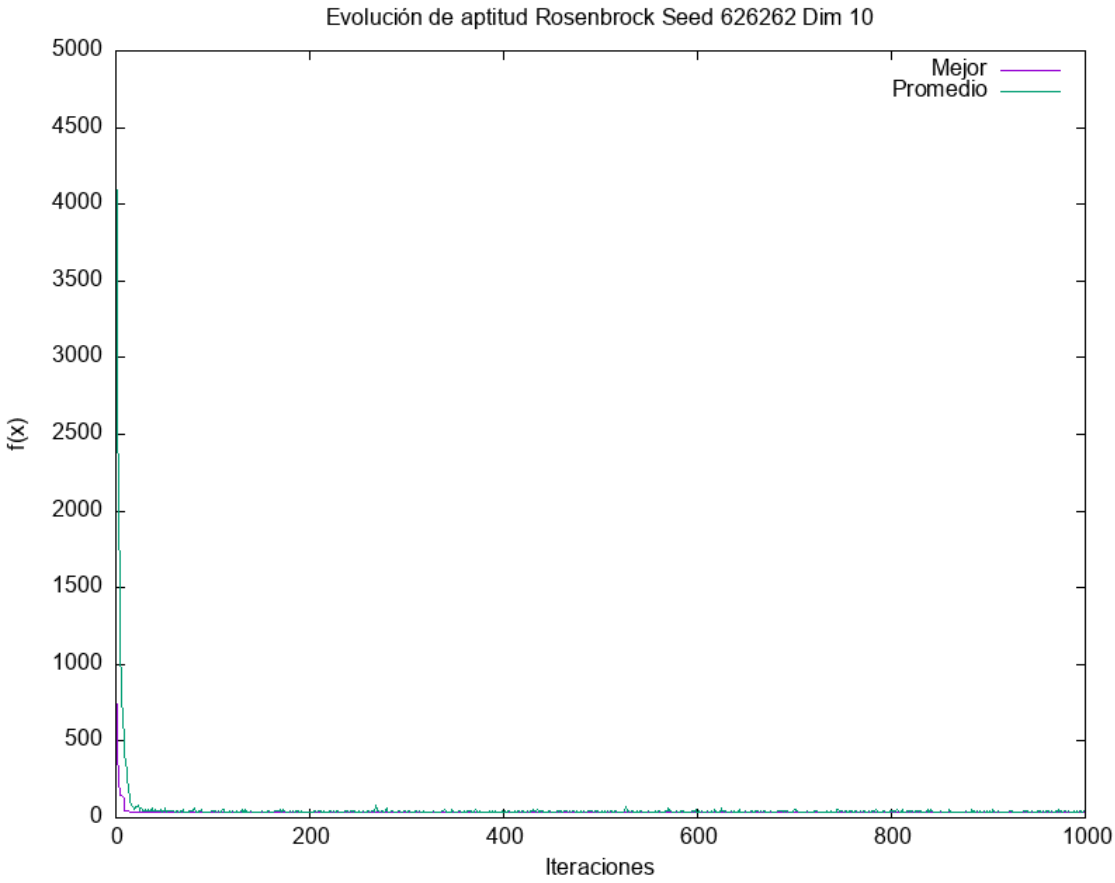

Figure 12: Evolución de aptitud

Podríamos analizar una por una, pero en realidad estas gráficas lo que nos muestran no es tanto el comportamiento de la función, en realidad nos muestran el comportamiento del algoritmo genético, donde podemos observar que conforme pasan las generaciones en la población, el valor de su aptitud promedio aunque se acerca al valor de la mejor aptitud, este el promedio aumenta y disminuye.

Esto nos indica que con el paso de las generaciones, la población va convergiendo hacia un mismo individuo y después de cierta generación los nuevos individuos mutados o generados van a ser opacados por los individuos similares, ya no hay suficiente variación genética.

En nuestro caso con las parámetros dados y la implementación notamos una convergencia prematura que al hacer pruebas me di cuenta que el elitismo metía ruido en lo rápido que se hacía la convergencia.

Al aplicar el elitismo, encontramos en menos generaciones la convergencia, lo que podría parecer malo, pero sin el elitismo aunque la convergencia no era tan prematura, las soluciones no eran tan buenas, ya que estábamos desechando toda la generación anterior.

Y al usar una probabilidad de cruza de 1, tener clones dependía únicamente de si la ruleta selecciona el mismo individuo para ambos padres.

En conclusión para este caso el elitismo favoreció una convergencia prematura pero también dio mejores resultados al usar poblaciones mayores.

Una suposición que yo tenía de observar el comportamiento de las ejecuciones, es que la ruleta también favorecía la convergencia prematura, ya que al elegir a los de mejor aptitud con mayor probabilidad, si alguno de los individuos supera por mucho en aptitud a los demás, la probabilidad de que lo seleccione en ambos padres es alta.

Sin embargo el profesor en clase nos mencionó que la ruleta mantiene el promedio, entonces la cruza y la mutación es lo que va generando los cambios en el promedio.

(c) Ejecutar el algoritmo genético al menos 30 veces para cada función de prueba, e incluir una tabla con los resultados estadísticos (mejor, peor y valor promedio en cada función). * En cada ejecución se debe utilizar una semilla diferente para el generador de números aleatorios.

Función	Mejor BA	Promedio BA	Peor BA	Mejor RC	Promedio RC	Peor RC
Sphere	4.54240	87.37550	209.95746	1.49×10^{-11}	1.49×10^{-11}	1.49×10^{-11}
Ackley	13.86400	20.95400	22.08995	2.86×10^{-5}	2.86×10^{-5}	2.86×10^{-5}
Griwank	19.07111	300.99936	715.12404	0.024	0.254	0.582
Rastrign	38.62993	185.32163	336.48723	6.216	16.370	26.494
Rosenbrock	75.06251	4446.55986	19781.91129	6.3221	28.533	79.272

Table 1: Comparación de valores para diferentes funciones utilizando los algoritmos BA (Búsqueda aleatoria) y RC (Recocido Simulado)

Función	Mejor AG	Peor AG	Promedio AG
Sphere	0.022529791	1.498948301	0.505273971
Ackley	2.664108576	10.15473223	5.583006558
Griwank	0.957056732	8.82382243	2.554489937
Rastrign	3.572308287	18.05703915	8.861523469
Rosenbrock	8.349262196	55.01819555	19.88604718

Table 2: Comparación de valores para el algoritmo AG (Algoritmo Genético)

Lo que podemos notar de la tabla es que las funciones con mínimos globales más marcados son mejor optimizadas con el recocido simulado, recordando que el recocido simulado con una temperatura alta es básicamente una búsqueda aleatoria, pero a medida que se va enfriando se convierte en una búsqueda local.

Análisis recocido simulado:
En las primeras iteraciones del recocido simulado la temperatura va a ser alta

y nos va a permitir realizar una búsqueda aleatoria por exploración y cuando la temperatura baje se convierte en búsqueda local por explotación. Esto se ve reflejado en las funciones que estamos optimizando.

Por ejemplo, en la función sphere y ackley, si logras llegar a la bajada alrededor del mínimo global antes de que se enfríe mucho la temperatura, al momento de tener una temperatura baja la búsqueda por explotación te va a llevar directo al mínimo global. Ya que las vecindades (binarias, porque en decimal podríamos definir una vecindad más grande que ya no cumpla esto) alrededor de este mínimo global tanto en sphere como en ackley no tiene ruido, es decir casi siempre hay un vecino que haga menor la evaluación. Digo casi siempre porque es posible que se llegue al mínimo global permitido por la representación binaria.

Por otro lado funciones como la Rastringin (Griewank también) donde hay muchos mínimos locales y no hay bajadas tan marcadas en el mínimo global, es más probable que cuando la temperatura se enfríe no estemos cerca del mínimo global y con la búsqueda local por explotación solo llegaremos al mínimo local.

Algo similar pasa con la función Rosenbrock que en dos dimensiones podemos verla como una parábola invertida en la que hay todo un eje donde las evaluaciones son muy similares y en ese eje está el mínimo global, lo que puede hacer que nos movamos bastantes veces por el espacio de búsqueda y la evaluación no esté cambiando demasiado, es una bajada muy poco pronunciada.

Por eso al recocido simulado se le complica optimizar dichas funciones.

Análisis algoritmo genético:
Para las funciones sphere y ackley podemos notar que tanto el promedio como el mejor no son tan buenos como el el promedio y el mejor del recocido simulado, esto se debe a que el algoritmo genético no hace una búsqueda por explotación tan intensiva como el recocido simulado.

Sin embargo por esta misma propiedad de que la búsqueda sea más general, es que en funciones como en la Griwank podemos encontrar mejores soluciones, ya que el algoritmo genético tiene toda una población de individuos al mismos tiempo, en la que si alguno de esos individuos o sus hijos cae en una óptimo local (deja de mejorar), aún cabe la posibilidad de que otro de los individuos de la población esté fuera del descenso de ese óptimo local y pueda seguir mejorando.

En general eso es lo que yo creo que es lo más importante del algoritmo genético, disminuye los estancamientos en mínimos locales pero no pule tanto a los mejores individuos en lugar de buscar vecindades los va reproduciendo de tal forma que la ruleta elimina a los peores individuos.

Adicionalmente si los individuos no son tan variados va a tener una convergencia prematura. Por eso es importante distribuir a los individuos más o menos de forma uniforme.

Adicionalmente hay algo que observé, pero necesito ejemplificarlo para explicarlo. Supongamos que hay 4 individuos, tres muy similares en su representación binara y uno distinto. Supongamos que los tres primeros individuos tienen de aptitud 1.22, 1.23 y 1.21 respectivamente y el cuarto individuo tiene una aptitud de 0.6 Al tirar la ruleta como el cuarto individuo es el de mejor aptitud entonces su cuantil será el más grande que el de los otros por separado. Aptitud total:

$$f = \frac{1}{1.22} + \frac{1}{1.23} + \frac{1}{1.21} + \frac{1}{0.6} = 4.125793209$$

La probabilidad de elegir al cuarto individuo es de:

$$P = \frac{\frac{1}{0.6}}{f} = 0.4039627248$$

Pero recordando que los otros tres individuos son similares, elegir a cualquiera de ellos va a ser casi lo mismo a la hora de reproducirlo, entonces podemos verlo como si fueran uno solo.
De esta forma la probabilidad de elegirlos sería de:

$$P = \frac{\frac{1}{1.22}}{f} + \frac{\frac{1}{1.23}}{f} + \frac{\frac{1}{1.21}}{f} = 0.5960372752$$

¿Esto que nos quiere decir?
Esto nos indica que la probabilidad de elegir como padre para reproducir a un individuo muy similar va a ser mayor a la probabilidad de elegir un individuo completamente nuevo y diferente, aunque el individuo diferente tenga una mayor aptitud.

Es por esto que las poblaciones convergen, no es necesariamente porque ya no sigan apareciendo mejores individuos o que estemos en un mínimo global, es porque la población es tan similar que elegir individuos distintos ya se vuelve muy poco probable, es decir la ruleta termina eliminando individuos aunque sean mejores (gracias al elitismo esto se contrarresta pero solo para un individuo).

Conclusión:
El algoritmo genético no nos asegura que la mejora solución encontrada sea un mínimo local.
El algoritmo genético me parece mejor para funciones que se sepa que son difíciles de optimizar, como lo podrían ser funciones con un espacio de búsqueda muy grande, funciones con múltiples mínimos globales, funciones con ruido, funciones con descensos al mínimo global no muy pronunciados, etc.
El algoritmo genético es computacionalmente muy costoso por la cantidad de individuos que genera por cada iteración, sin embargo después de ciertas iteraciones las poblaciones convergen y ya no tiene caso hacer más iteraciones, se podría optimizar para que se detenga después de ciertas generaciones sin mejoras o con

mejores muy chicas, pero definir esto va a depender mucho del problema a resolver.
El algoritmo genético es muy buena opción si no conoces el comportamiento de la función ya que se distribuye mejor por el espacio de búsqueda.

En general no podría decir que uno sea mejor que el otro porque depende de qué quieras optimizar y qué conocimientos tengas de ese problema.
También la selección de los parámetros no es una tarea fácil, es muy probable que con una mejor elección de parámetros hubiera podido encontrar mejores soluciones, quizá con convergencias menos prematuras.

(d) **Ejecución de prueba:**

```
PS E:\Documentos\4t sem\Cómputo Evolutivo\CE_Tarea04> java -jar .\target\ejecuta.jar
Algoritmo genético
Seleccione una opción:
1. Problema de coloración BLI
2. Problema de optimización continua
2
Seleccione la función a evaluar:
1. Sphere Function
2. Ackley Function
3. Griewank Function
4. Rastrigin Function
5. Rosenbrock Function
6. Sum Square Function
7. Styblinski-Tang Function
8. Dixon-Price Function
1
Ingrese el tamaño de la población:
200
Ingrese la semilla para generar la población inicial:
765
Ingrese la probabilidad de cruza:
1
Ingrese la dimensión:
10
Texto guardado en el archivo: src/output/solucionesContinuas/reporte.txt
El valor de la función es: 0.13089263740724133
```

Figure 13: Ejecución

2. Ejercicio 2. Coloración en Gráficas

(a) Describe e Implementa un algoritmo genético para el problema de optimización de gráficas. **

i) La elección de los componentes a utilizar e implementar es libre, pero deben utilizar al menos dos componentes diferentes a los utilizados en el ejercicio anterior.

Componentes Utilizados

i. **Inicialización**

Se genera una población de soluciones aleatorias válidas, en donde a cada vértice se le asigna un color distinto a sus adyacentes.

Algorithm 5 Generar Población Inicial

```
1: function POBLACIONINICIAL(nPoblacion)
2:     for i ← 0 to nPoblacion − 1 do
3:         poblacion.add(SOLUCIONALEATORIA)        ▷ Añade una solución aleatoria a la población
4:     end for
5: end function
```

El tamaño de la población es pasada como parámetro al método *poblacionInicial* que utiliza otro método auxiliar que se encarga de generar una solución aleatoria, coloreando cada vértice de la gráfica respetando las adyacencias. (Método ya utilizando anteriormente)

ii. **Evaluación**

Se evalúan las soluciones contando el número de colores totales utilizados para colorear la gráfica, mientras menor sea el número entonces la solución es mejor. De éste mismo método surgen otras dos variantes, el promedio de evaluación de la población y la mejor solución de la población, donde se itera sobre todas las soluciones de la población y después calcula el promedio o devuelve la solución con menor número de colores utilizados, respectivamente.

Algorithm 6 Evaluar Solución

```
1: function EVALUARSOLUCION(solucion)
2:     coloresUsados ← new boolean[numVertices + 1]    ▷ Inicializa arreglo para marcar colores usados
3:     totalColores ← 0                                ▷ Inicializa el contador de colores únicos usados
4:     for each color in solucion do
5:         if not coloresUsados[color] then
6:             coloresUsados[color] ← true             ▷ Marca el color como usado
7:             totalColores ← totalColores + 1         ▷ Incrementa el contador de colores únicos
8:         end if
9:     end for
10:    return totalColores                             ▷ Devuelve el número de colores únicos usados
11: end function
```

iii. **Selección**

Seleccionamos las soluciones más aptas utilizando el método de *torneo*, el cual escoge un porcentaje de la población inicial (En nuestro caso un 30%) y las compara entre ellas, la solución con mejor evaluación, es decir, la que utiliza menos colores, es candidata para reproducirse.

Algorithm 7 Método de Selección de Torneo

```
1: function TORNEO(poblacion)
2:     rand ← new Random()
3:     tamanoTorneo ← int(poblacion.size() × 0.3)      ▷ Selecciona el 30% de la población
4:     mejorIndice ← −1
5:     mejorEvaluacion ← Integer.MAX_VALUE
6:     for i ← 0 to tamanoTorneo − 1 do
7:         indiceAleatorio ← rand.nextInt(poblacion.size())
8:         solucionCandidata ← poblacion.get(indiceAleatorio)
9:         evaluacionCandidata ← EVALUARSOLUCION(solucionCandidata)
10:        if evaluacionCandidata < mejorEvaluacion then
11:            mejorEvaluacion ← evaluacionCandidata
12:            mejorIndice ← indiceAleatorio
13:        end if
14:    end for
15:    return poblacion.get(mejorIndice).clone()      ▷ Devuelve el mejor individuo del
   torneo
16: end function
```

iv. **Reproducción**

Para reproducir nuestras soluciones padres generadas por el torneo, se utilizará una cruza de 2 puntos, el cual selecciona dos puntos aleatorios de nuestra solución padre e intercambia sus soluciones, lo que genera dos hijos, el primero con dos partes del primero padre y una del segundo, y el segundo hijo con dos partes del segundo padre y una del primero.
Utilizamos este tipo de reproducción para diversificar las mejores soluciones, sorpresivamente las soluciones resultantes son en su mayoría válidas.

Algorithm 8 Cruza de Dos Puntos

```
1: function CRUZAR(padre1, padre2)
2:     rand ← new Random()
3:     hijos ← new int[2][numVertices]
4:     puntoCruce1 ← rand.nextInt(numVertices − 1)
5:     puntoCruce2 ← rand.nextInt(numVertices − puntoCruce1) + puntoCruce1 + 1
6:     if puntoCruce1 > puntoCruce2 then
7:         temp ← puntoCruce1
8:         puntoCruce1 ← puntoCruce2
9:         puntoCruce2 ← temp
10:    end if
11:    for i ← 0 to numVertices − 1 do
12:        if i ≥ puntoCruce1 and i ≤ puntoCruce2 then
13:            hijos[0][i] ← padre2[i]
14:            hijos[1][i] ← padre1[i]
15:        else
16:            hijos[0][i] ← padre1[i]
17:            hijos[1][i] ← padre2[i]
18:        end if
19:    end for
20:    return hijos
21: end function
```

v. **Mutación**

Para mutar las soluciones, seleccionamos un vértice aleatorio y almacenamos la información de los colores utilizados por sus adyacencias, para que las solución mutada de igual forma sea válida. El color del vértice seleccionado es cambiado por otro color en uso aleatorio. Es posible disminuir el número de colores utilizados con esta mutación y genera variaciones que ayudan a nutrir a la población. La probabilidad de mutación puede variar, en nuestro caso es 0.1, puesto que es un porcentaje adecuado para explorar otras soluciones y al mismo tiempo explotar las mejores soluciones ya obtenidas.

Algorithm 9 Mutación de una Solución

```
1: function MUTACION(solucion)
2:     rand ← new Random()
3:     mutada ← solucion.clone()
4:     vertice ← rand.nextInt(numVertices)
5:     coloresInvalidos ← new boolean[numVertices + 1]
6:     for i ← 0 to numVertices − 1 do
7:         if matrizAdyacencia[vertice][i] = 1 then
8:             coloresInvalidos[solucion[i]] ← true
9:         end if
10:    end for
11:    coloresValidos ← new ArrayList()
12:    for color ← 1 to numVertices do
13:        if not coloresInvalidos[color] then
14:            coloresValidos.add(color)
15:        end if
16:    end for
17:    if coloresValidos ≠ empty then
18:        colorNuevo ← coloresValidos.get(rand.nextInt(coloresValidos.size()))
19:        mutada[vertice] ← colorNuevo
20:    end if
21:    return mutada
22: end function
```

vi. **Reemplazo**

Realizamos un reemplazo elitista, es decir, conservamos la solución mejor evaluada de la solución actual para después agregarla a la siguiente generación, los demás miembros de la solución son los hijos de las soluciones ganadoras de un torneo.
Éste reemplazo se realiza en el método que junta todos los métodos anteriores para ejecutar el algoritmo genético, es por eso que adjuntaremos el pseudocódigo únicamente del reemplazo.

Algorithm 10 Algoritmo Genético Elitista para Coloración de Grafos

```
1: function AlgoritmoGeneticoElitista(numGeneraciones, tasaMutacion)
2:     rand ← new Random()
3:     mejorEvaluacionPorGeneracion ← new ArrayList()
4:     promedioEvaluacionPorGeneracion ← new ArrayList()
5:     for generacion ← 0 to numGeneraciones − 1 do
6:         nuevaPoblacion ← new ArrayList()
7:         mejorIndividuo ← seleccionarMejorSolucion(poblacion)
8:         while nuevaPoblacion.size() < poblacion.size() − 1 do
9:             padre1 ← torneo(poblacion)
10:            padre2 ← torneo(poblacion)
11:            hijos ← cruzar(padre1, padre2)
12:            for i ← 0 to hijos.length − 1 do
13:                if rand.nextDouble() < tasaMutacion then
14:                    hijos[i] ← mutar(hijos[i])
15:                end if
16:                if esSolucionValida(hijos[i]) then
17:                    nuevaPoblacion.add(hijos[i])
18:                    if nuevaPoblacion.size() == poblacion.size() − 1 then
19:                        break
20:                    end if
21:                end if
22:            end for
23:        end while
24:        nuevaPoblacion.add(mejorIndividuo)
25:        poblacion ← nuevaPoblacion
26:    end for
27:    return seleccionarMejorSolucion(poblacion)
28: end function
```

vii. **Termino**
Definimos el número de generaciones como nuestro criterio de término, es decir, el algoritmo genético recibe un número entero que representa la cantidad de generaciones y regresamos la solución mejor evaluada tras esas iteraciones.

Para abordar el problema de la coloración utilizando algoritmo genético, variamos la implementación en la selección y cruza, en donde utilizamos torneo y cruza de dos puntos.

Selección de torneo nos ayuda a mantener la diversidad genética, es decir, podemos hacer más o menos grande el tamaño del torneo dependiendo las necesidades, es decir, un torneo grande favorece a las soluciones mejor evaluadas, mientras que un torneo pequeño favorece a la diversidad y exploración de las soluciones.

Cruza de dos puntos la implementamos tomando en cuenta que al intercambiar los genes de los padres no siempre los hijos resultantes serían soluciones válidas, nuestra solución fue generar nuevos hijos hasta que ambos sean válidos y éste tipo de cruza lo permitió.

(b) **Tabla de comparación de resultados con los resultados de tareas anteriores**
Utilizamos la siguiente gráfica como ejemplo

```
c Gráfica con 18 vértices y 25 aristas, mínimo global = 3
p edge 18 25
e 1 2
e 1 3
e 1 18
e 2 4
e 2 5
e 2 17
e 3 6
e 3 16
e 4 7
e 4 8
e 5 9
e 5 10
e 6 11
e 6 12
e 7 13
e 8 14
e 9 15
e 10 16
e 11 17
e 12 18
e 13 14
e 14 15
```

```
e 15 16
e 16 17
e 17 18
```

Ahora compararemos los resultados de diferentes búsquedas
Algoritmo genético
número de generaciones = 50
miembros de la población = 100
probabilidad de mutación = 0.1
Búsqueda local iterada
número de iteraciones = 50
tolerancia máxima = 5
Búsqueda por escalada
tolerancia máxima = 50

Notamos que el algoritmo genético es el único que logra llegar al mínimo global, esto se logra gracias a que podemos escapar de mínimos locales mutando ciertas soluciones y agregándolas a nuestra población. De igual forma partimos de una evaluación mejor que las otras búsquedas puesto que tenemos varias soluciones en nuestra población, alguna de ellas nos dará la mejor solución en la primer iteración y ésta se guardará para la siguiente generación, pues utilizamos elitismo. La búsqueda local iterada se atasca en un mínimo local, es posible que las soluciones vecinas no sean tan agresivas y tampoco puedan salir del mínimo local, sin embargo, ésta misma búsqueda con diferentes parámetros puede llegar a dar mucho mejores soluciones a costa de rendimiento computacional, lo mismo sucede con la búsqueda por escalada, que no tiene tanta diversidad en sus soluciones y el número de iteraciones no basta para llegar al mínimo global.

(c) **Gráfica de evolución de aptitud.**
Comenzamos ejecutando nuestro algoritmo con los siguientes argumentos
número de generaciones = 100
miembros de la población = 100
probabilidad de mutación = 0.1
Utilizamos la misma gráfica del inciso anterior, la cual utiliza únicamente 3 colores.
Notamos que nuestro algoritmo genético comienza la población aleatoria donde el peor resultado es 11 colores, al pasar las 10 generaciones rápidamente disminuye los colores utilizados a solamente 5. Se queda estancada la mejor solución durante 20 iteraciones hasta llegar el mínimo global, donde ninguna otra generación puede superar puesto que ya hemos encontrado la mejor solución. Notamos que fueron necesarias aproximadamente 40 iteraciones para llegar al mínimo global, sin embargo las poblaciones eran de 100 soluciones y al ser una gráfica relativamente pequeña, consideramos que no es tan eficiente, comparando el costo computacional con la calidad de la solución.

La eficiencia de éste algoritmo depende de muchos factores, como la mutación y la probabilidad de mutación, elección de padres, elección de cruza, etc., por

Iteración	Algoritmo Genético	Búsqueda Local Iterada	Búsqueda por Escalada
1	8	12	10
2	7	10	10
3	6	7	10
4	5	7	9
5	5	6	9
6	5	6	8
7	5	6	8
8	5	6	8
9	5	6	8
10	5	6	8
11	5	6	8
12	4	6	8
13	4	6	8
14	4	5	8
15	4	5	8
16	4	5	8
17	4	5	8
18	4	5	8
19	4	5	8
20	4	5	8
21	4	5	8
22	4	5	8
23	4	5	8
24	4	5	8
25	4	5	8
26	4	5	8
27	4	5	7
28	4	5	7
29	4	5	7
30	4	5	7
31	4	5	7
32	4	5	7
33	3	5	7
34	3	5	7
35	3	5	7
36	3	5	7
37	3	5	7
38	3	5	6
39	3	5	6
40	3	5	5
41	3	5	5

lo tanto es posible que el enfoque no haya sido correcto y que realmente estemos desaprovechando el potencial del algoritmo y es por eso que no vemos una solución eficaz en una gráfica tan sencilla.

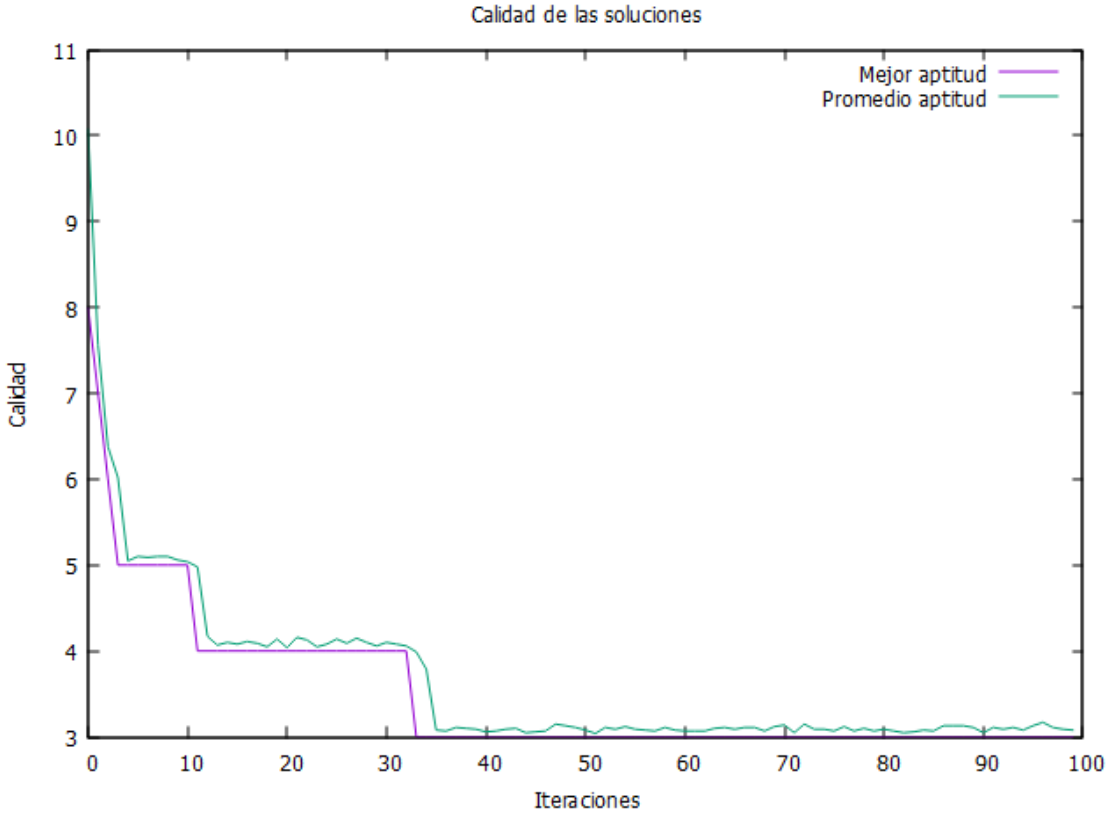

Figure 14: Evolución de aptitud