INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE COMPUTO

2024

SESIÓN DE LABORATORIO 4: INTRODUCCIÓN A LOS ALGORITMOS GENÉTICOS

ALGORITMOS BIOINSPIRADOS

Elaborado por: Flores Estopier Rodrigo

Boleta: 2021630260

Profesor: Rosas Trigueros Jorge Luis

Grupo: 6CV2

Fecha de realización: 04/10/2024

Fecha de entrega: 08/10/2024

Semestre: 25-1

Contenido

1. MARCO TEÓRICO

Los algoritmos genéticos (AG) son un conjunto de técnicas de optimización y búsqueda inspiradas en los principios de la selección natural y la genética de los seres vivos, propuestos por John Holland en la década de 1970. Un AG simula el proceso de evolución de una población de soluciones candidatas mediante operadores genéticos como la selección, cruce (crossover), y mutación, buscando encontrar soluciones óptimas a problemas complejos.

Los AG son útiles en la resolución de problemas que presentan un espacio de búsqueda grande, no lineal o multimodal, donde los métodos tradicionales de optimización pueden fallar. El algoritmo evoluciona una población de soluciones, seleccionando las mejores con base en una función de aptitud (fitness), y utilizando los operadores genéticos para explorar nuevas áreas del espacio de búsqueda.

Principios de Funcionamiento de un Algoritmo Genético

- **Población Inicial**: Los AG inician con una población de soluciones representadas generalmente como cromosomas (cadenas binarias o secuencias numéricas). Cada cromosoma codifica una posible solución al problema.
- **Función de Aptitud**: A cada cromosoma se le asigna un valor de aptitud basado en qué tan buena es la solución que representa en relación con el objetivo. La función de aptitud guía la evolución de la población.
- **Selección**: Se seleccionan cromosomas para la reproducción con base en su aptitud. Los cromosomas con mejor aptitud tienen mayor probabilidad de ser seleccionados para producir la siguiente generación.
- **Cruce (Crossover)**: Se seleccionan dos cromosomas (padres) y se combinan en un punto de cruce para producir uno o dos descendientes. Este operador explora nuevas combinaciones en el espacio de búsqueda.
- **Mutación**: Se aplican cambios aleatorios a uno o más genes (bits o valores) de un cromosoma, lo cual introduce diversidad en la población, ayudando a evitar soluciones subóptimas (mínimos locales).
- **Elitismo**: Los mejores cromosomas de la población actual pueden ser preservados directamente para la siguiente generación, asegurando que las mejores soluciones no se pierdan.

El ciclo de selección, cruce y mutación se repite durante varias generaciones, buscando mejorar las soluciones de la población hasta converger a una solución óptima o hasta que se alcance un criterio de parada.

Función de Ackley

La función de Ackley es una función de prueba no convexa utilizada en la optimización, conocida por su comportamiento altamente multimodal, lo que significa que contiene muchos mínimos locales. La función de Ackley en dos dimensiones se define como:

$$f(x,y)=-20\exp(-0.20.5(x2+y2))-\exp(0.5(\cos(2\pi x)+\cos(2\pi y)))+e+20$$

Donde e es el número de Euler. Esta función tiene un mínimo global en el punto (0,0), donde f(x,y)=0.

Función de Rastrigin

La función de Rastrigin es otra función de prueba comúnmente utilizada en problemas de optimización debido a su comportamiento altamente multimodal. En tres dimensiones, la función de Rastrigin se define como:

$$f(x,y,z)=10\times3+(x2-10\cos(2\pi x))+(y2-10\cos(2\pi y))+(z2-10\cos(2\pi z))$$

El mínimo global de la función ocurre en el punto (0,0,0), donde el valor de la función es 0.

Tanto la función de Ackley como la de Rastrigin son utilizadas como funciones de prueba debido a su alta complejidad, caracterizadas por la presencia de muchos mínimos locales. Sin embargo, la función de Ackley presenta una suave estructura de pozos y colinas, mientras que la función de Rastrigin tiene un comportamiento más oscilante debido a los términos trigonométricos de tipo coseno que generan valles más estrechos.

2. DESARROLLO

Se modificará el código proporcionado en clase para adaptar el algoritmo genético para optimizar la funcion de Ackley y la funcion de Rastrigin. De nuevo se utilizará Python en el entorno de desarrollo de Google Colab.

OPTIMIZACIÓN DE LA FUNCION ACKLEY

Al aplicar un algoritmo genético para optimizar la función de Ackley en dos dimensiones, los cromosomas se codifican como secuencias binarias que representan los valores de

las variables x e y. A través de la selección de los mejores cromosomas en cada generación y la aplicación de operadores genéticos como el cruce y la mutación, el algoritmo genético intenta minimizar la función y acercarse al valor óptimo de f(0,0)=0. La diversidad de la población es clave en este proceso, dado que la función de Ackley tiene múltiples mínimos locales.

Con ayuda de la librería widget de Python, realizamos una funcion para crear un botón que nos permitirá avanzar entre generaciones.

```
import ipywidgets as widgets

def create_button():
  button = widgets.Button(
    description='Next Generation',
    disabled=False,
    button_style='', # 'success', 'info', 'warning', 'danger' or ''
    tooltip='Next Generation',
    icon='check' # (FontAwesome names without the `fa-` prefix)
  )
  return button
```

Entre las librerías que utilizaremos, se encuentran:

- Random: para la ruleta de elección.
- matplotlib.pyplot: para la impresión de graficas en pantalla.
- Numpy: Para la funcion que se va a optimizar.
- Functools, cmp_to_key: para utilizer una funcion de comparación como criterio de ordenamiento.

Como vamos a optimizar en un espacio de dos dimensiones debemos definir nuestro tamaño de cada cromosoma, en esta caso definiremos un tamaño de 32 bits, donde los primeros 16 bits representan el valor de x y los 16 bits restantes representan a y.
También establecemos nuestro espacio de búsqueda, en este caso entre -5 y 5.

```
#Chromosomes de 32 bits de longitud
L_chromosome=32    #16 bits para x ,16 bits para y
N_chains=2**(L_chromosome//2)
#Limites del espacio de busqueda
a=-5
b=5
```

```
crossover_point= L_chromosome//2
```

Definimos una funcion para generar cromosomas de forma aleatoria. Esta se realiza en una codificación binaria.

```
def random_chromosome():
    chromosome=[]
    for i in range(0,L_chromosome):
        if random.random()<0.5:
            chromosome.append(0)
        else:
            chromosome.append(1)

    return chromosome
```

Establecemos el tamaño de nuestra población de 10 en este caso, y una probabilidad de mutación del 75%. Ademas creamos nuestra población inicial utilizando la funcion de generación aleatoria de cromosomas, mientras que establecemos un valor de aptitud de 0 a todos los cromosomas.

```
#Number of chromosomes
N_chromosomes=10
#probability of mutation
prob_m=0.75

#Creamos la poblacion inicial
F0=[]
fitness_values=[]

for i in range(0,N_chromosomes):
    F0.append(random_chromosome())
    fitness_values.append(0)
```

Modificamos la funcion que se encarga de decodificar los cromosomas, ya que, en este caso, obtendremos dos valores (x,y) de cada cromosoma. Primero decodificamos la primera mitad del cromosoma, lo que corresponde al valor de x, y posteriormente decodificamos el valor de y. Esta funcion convierte las cadenas binarias en valores decimales entre el espacio de búsqueda (a,b).

```
def decode_chromosome(chromosome):
    global L_chromosome,N_chains,a,b
    half_length=L_chromosome//2
    value=0
    # Decodificamos la primera mitad para x
    value_x = sum((2**p) * chromosome[-1-p] for p in range(half_length))
    x = a + (b - a) * float(value_x) / (N_chains - 1)

    # Decodificamos la segunda mitad para y
    value_y = sum((2**p) * chromosome[half_length-1-p] for p in
range(half_length))
    y = a + (b - a) * float(value_y) / (N_chains - 1)

    return x, y
```

Creamos la funcion de aptitud, que en este caso es la funcion de Ackley, la cual queremos optimizar.

```
def ackley(x,y):
    return -20*np.exp(-0.2*np.sqrt(0.5*(x*x+y*y)))-
np.exp(0.5*(np.cos(2*np.pi*x)+np.cos(2*np.pi*y)))+np.e+20
```

La funcion para evaluar cromosomas utiliza la funcion de decodificación y la funcion de aptitud para obtener los valores x,y de cada cromosoma en la población actual y calcular su aptitud para ingresarlos a una lista que representa la aptitud de cada cromosoma.

```
def evaluate_chromosomes():
    global F0,fitness_values

    for p in range(N_chromosomes):
        x,y=decode_chromosome(F0[p])
        fitness_values[p]=ackley(x,y)
```

También, definimos la funcion de comparación de cromosomas, la cual recibe dos cromosomas y compara sus valores de aptitud. Esta funcion será utilizada como criterio para ordenar a los cromosomas por aptitud.

```
def compare_chromosomes(chromosome1,chromosome2):
    x1, y1 = decode_chromosome(chromosome1)
    x2, y2 = decode_chromosome(chromosome2)

    fvc1=ackley(x1,y1)
```

```
    fvc2=ackley(x2,y2)
    if fvc1 > fvc2:
        return 1
    elif fvc1 == fvc2:
        return 0
    else: #fvg1<fvg2
        return -1
```

Para elegir a los cromosomas para la generación de nuevas generaciones, implementamos un mecanismo de selección basado en la aptitud mediante una ruleta que da más probabilidades de ser seleccionados a los cromosomas con mejor aptitud.
Primero definimos las variables:

- **Lwheel**: Define el tamaño de la "ruleta", donde cada cromosoma tendrá una fracción proporcional a su aptitud. El tamaño de la ruleta se multiplica por 10 para aumentar la granularidad de la selección.
- **maxv**: Valor máximo de aptitud en la población, que se utilizará para calcular las probabilidades relativas de los cromosomas.
- **acc**: Variable para acumular el "fitness relativo" de cada cromosoma en comparación con el valor máximo.

Posteriormente se calcula la fracción de la ruleta que corresponde a cada cromosoma. La fracción para cada cromosoma se obtiene dividiendo su aptitud relativa por el total acumulado (acc).
Si la fracción es demasiado pequeña, es ajustada a un mínimo de 1.0 / Lwheel para garantizar que ningún cromosoma tenga una probabilidad nula de ser seleccionado.
Después, se ajustan las fracciones de los dos primeros cromosomas para asegurarse de que la suma total de las fracciones sea exactamente 1.

Por último, construimos la ruleta, donde para cada fracción **f** calculada, se determina cuántos "espacios" (o "tickets") tiene ese cromosoma en la ruleta mediante **Np = int(f * Lwheel)**. Cromosomas con mayor aptitud ocupan más espacios en la ruleta, y, por lo tanto, tienen mayor probabilidad de ser seleccionados.
Luego, se añade el índice del cromosoma correspondiente (almacenado en **pc**) a la ruleta tantas veces como indique **Np**.

```
def create_wheel():
    global F0,fitness_values
    Lwheel=N_chromosomes*10
    maxv=max(fitness_values)
    acc=0
    for p in range(N_chromosomes):
        acc+=maxv-fitness_values[p]
    fraction=[]
    for p in range(N_chromosomes):
        fraction.append( float(maxv-fitness_values[p])/acc)
        if fraction[-1]<=1.0/Lwheel:
            fraction[-1]=1.0/Lwheel
    #print(fraction)
    fraction[0]-=(sum(fraction)-1.0)/2
    fraction[1]-=(sum(fraction)-1.0)/2
    #print(fraction)

    wheel=[]

    pc=0

    for f in fraction:
        Np=int(f*Lwheel)
        for i in range(Np):
            wheel.append(pc)
        pc+=1

    return wheel
```

Posteriormente, inicializamos la siguiente generación haciendo una copia de la generación actual. Y establecemos una variable n que nos permitirá saber cuantas generaciones se han creado.

```
F1=F0[:]
n=0
```

La funcion mas importante de este algoritmo es la funcion de generar una siguiente generación, aquí es donde utilizamos operadores genéticos para crear más candidatos a una solución.

Primero, ordenamos los cromosomas de acuerdo con su aptitud utilizando como criterio la funcion de comparación de cromosomas. Además, la mejor solución es mostrada, con los valores de x,y y el valor de la aptitud.

```
def nextgeneration():
    global F0, F1, n
    display.clear_output(wait=True)
    #display.display(button)

    # Sort the population by fitness
    F0.sort(key=cmp_to_key(compare_chromosomes))

    print( "Best solution so far:")
    xx,yy = decode_chromosome(F0[0])
    n +=1
    print("Generation: ", n)
    print("f",decode_chromosome(F0[0]),"= ", ackley(xx,yy) )
```

Posteriormente, reservamos los dos mejores cromosomas de la generación actual, para mantenerlos en la siguiente generación, además creamos la ruleta para seleccionar a los padres para la siguiente generación.

```
    # Elitism: Keep the two best chromosomes
    F1[0] = F0[0]
    F1[1] = F0[1]

    # Create the rest of the new generation
    roulette = create_wheel()
    print(roulette)
```

Ahora realizamos el proceso de selección de padres, cruce (crossover) y mutación para generar nuevos descendientes en la siguiente generación.

En un bucle donde el número de iteraciones es la mitad del tamaño de la población restante, porque en cada iteración se generan dos descendientes.

Con random.choice(roulette) seleccionamos aleatoriamente uno de los padres basándose en la ruleta, donde los cromosomas con mejor aptitud tienen más probabilidades de ser seleccionados.

El crossover se lleva a cabo tomando partes de los dos padres p1 y p2 para crear dos descendientes.

La variable crossover_point es el punto de cruce, que divide los cromosomas de los padres en dos partes. Los descendientes se generan combinando la primera mitad del cromosoma del padre 1 con la segunda mitad del cromosoma del padre 2 (y viceversa).
Después del cruce, se aplica mutación con una probabilidad prob_m.
Si la condición de mutación se cumple, un bit aleatorio del descendiente o1 o o2 se modifica usando una operación XOR (^= 1). Esto invierte el valor de un bit: si era 0, se convierte en 1, y si era 1, se convierte en 0.
Finalmente, asignamos los descendientes a la siguiente generación. Y después del bucle remplazamos la generación actual con la generación generada.

```
    for i in range(0,int((N_chromosomes-2)/2)):
        #Seleccionamos dos padres
        p1 = random.choice(roulette)
        p2 = random.choice(roulette)

        # Generamos dos descendientes mediante crossover
        o1 = F0[p1][:crossover_point] + F0[p2][crossover_point:]
        o2 = F0[p2][:crossover_point] + F0[p1][crossover_point:]

        # Mutacion
        if random.random() < prob_m:
            o1[random.randint(0, L_chromosome - 1)] ^= 1
        if random.random() < prob_m:
            o2[random.randint(0, L_chromosome - 1)] ^= 1

        #Agregamos los descendientes a la siguiente generacion F1
        F1[2+2*i]=o1
        F1[3+2*i]=o2

    # Remplazamos la generacion
    F0[:] = F1[:]
```

Por último, generamos las funciones para graficar la funcion de ackley y los puntos donde se encuentran las poblaciones.

```
def graph_f():
    global a,b
    xini=a
    xfin=b
    x=np.linspace(xini,xfin,500)
    y=np.linspace(xini,xfin,500)
    X,Y=np.meshgrid(x,y)
```

```
    Z=ackley(X,Y)
    plt.contourf(X,Y,Z,cmap='viridis')
    plt.colorbar()

def graph_population(F):
    x_population = []
    y_population = []

    for chromosome in F:
        x, y = decode_chromosome(chromosome)
        x_population.append(x)
        y_population.append(y)
    graph_f()
    plt.scatter(x_population, y_population, color='red',
label='Population')
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('Population Visualization')
    plt.legend()
    plt.show()
```

Para inicializar el programa creamos un botón para avanzar entre generaciones, evaluamos la generación inicial y graficamos su población.

```
button=create_button()
button.on_click(nextgeneration)
display.display(button)

evaluate_chromosomes()
graph_population(F0)
```

También podemos generar un pequeño código para ejecutar las generaciones que deseamos de manera automática.

```
for generaciones in range(10):
    nextgeneration(None)
    evaluate_chromosomes()

#Graficar la ultima generacion
graph_population(F0)
```

Ejecución

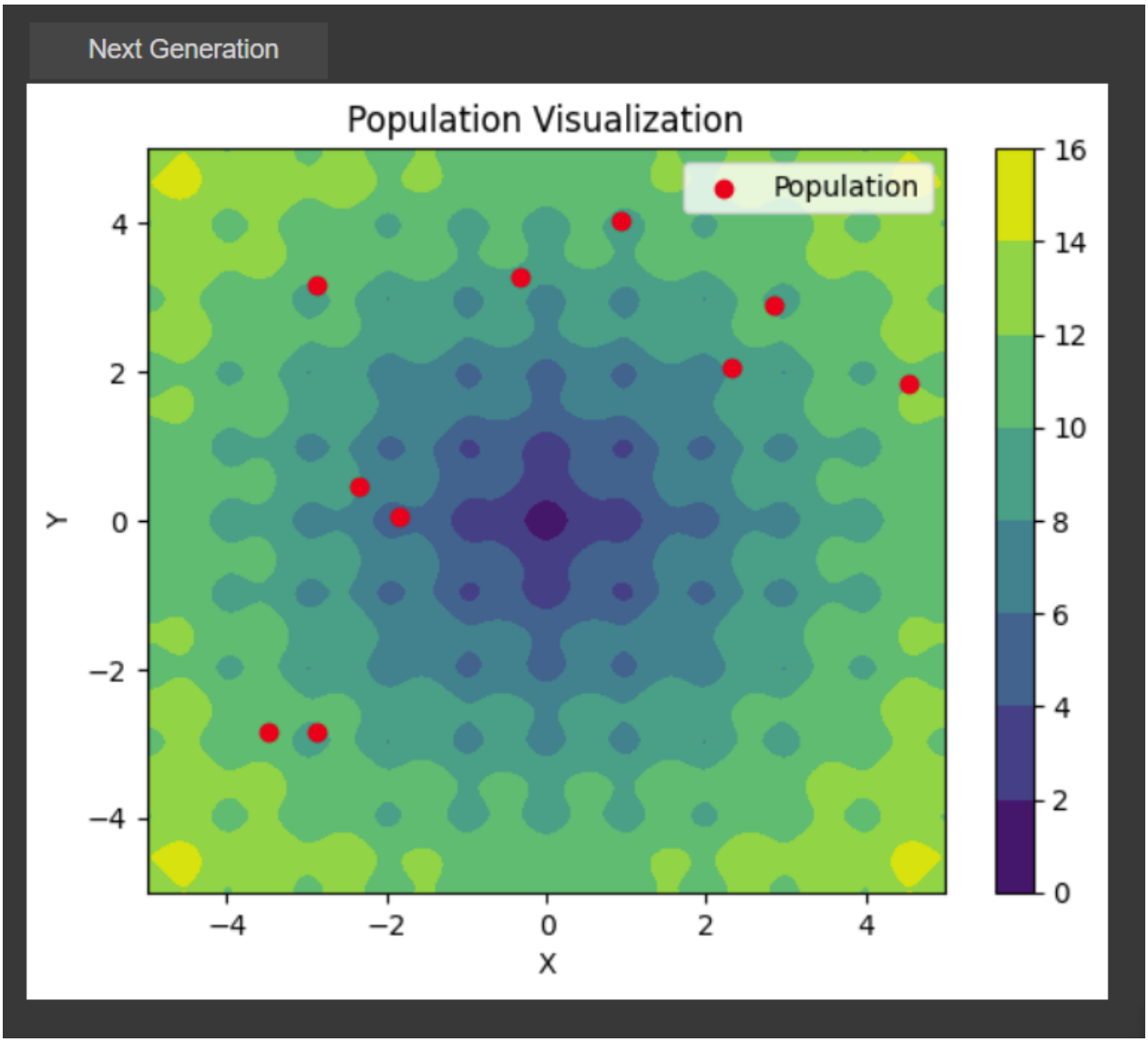

Ilustración 1. Generación inicial de la optimización de la funcion de Ackley.

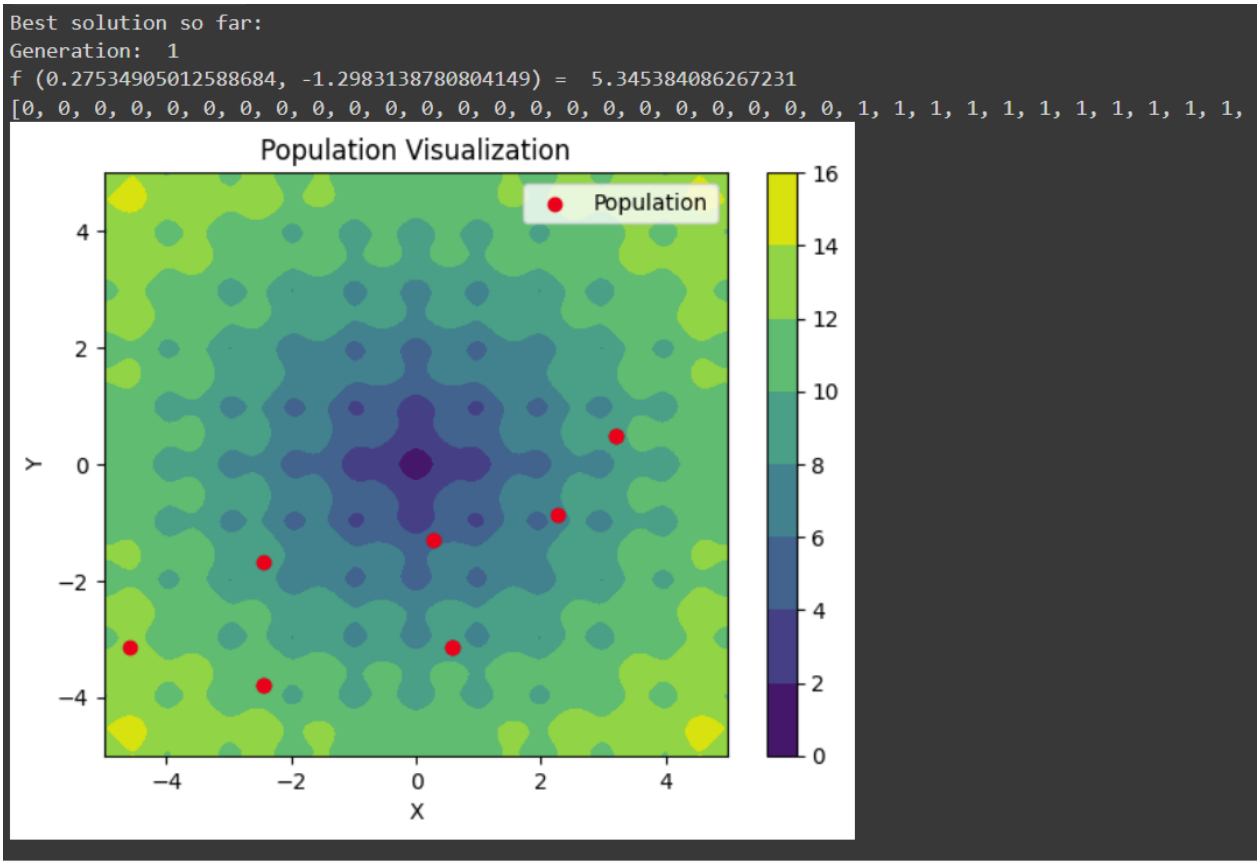

Ilustración 2. Primera generación de la optimización de la funcion de Ackley.

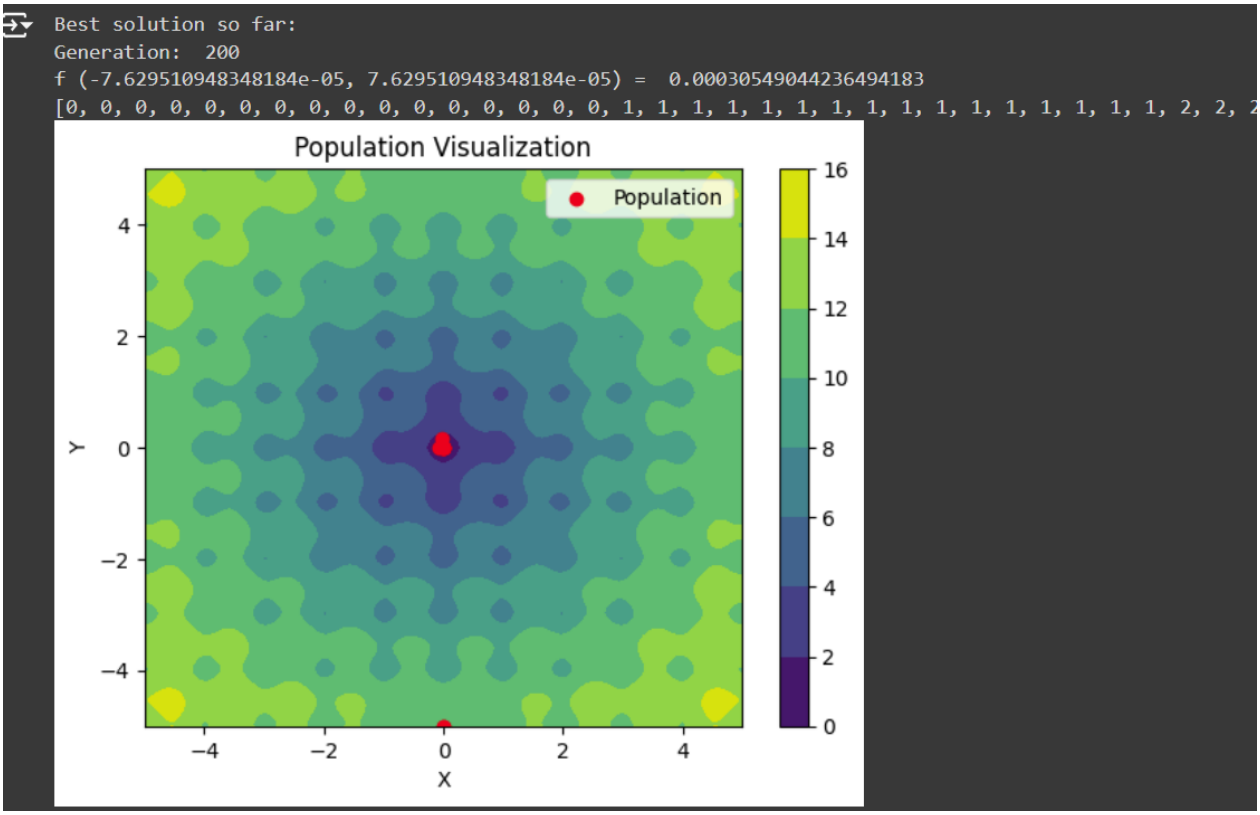

Ilustración 3. Generación 200 de la optimización de la funcion de Ackley.

OPTIMIZACIÓN DE LA FUNCION RASTRIGIN

Se utilizaron las mismas funciones utilizadas anteriormente, solo se modifico el algoritmo para trabajar con 3 variables, x,y,z y ahora la funcion de aptitud es la funcion de Rastrigin. A continuación, se muestran las funciones que fueron modificadas.

Primero, establecemos una longitud de cromosomas de 48 bits donde cada valor x,y,z tienen una longitud de 16 bits cada una. Ademas, definimos el espacio de búsqueda entre -5.12 y 5.12.

```
#Chromosomes de 48 bits de longitud
L_chromosome=48    #16 bits para x ,16 bits para y, 16 bits para z
N_chains=2**(L_chromosome//3)
#Limites del espacio de busqueda
a=-5.12
b=5.12
```

Posteriormente, la funcion de decodificación de cromosomas fue modificada para generar 3 valores x,y,z de un solo cromosoma, dividiendo este entre 3.

```
def decode_chromosome(chromosome):
    global L_chromosome,N_chains,a,b
    third_length=L_chromosome//3
    # Decode the first third for x
    value_x = sum((2**p) * chromosome[-1-p] for p in range(third_length))
    x = a + (b - a) * float(value_x) / (N_chains - 1)

    # Decode the second third for y
    value_y = sum((2**p) * chromosome[third_length-1-p] for p in
range(third_length))
    y = a + (b - a) * float(value_y) / (N_chains - 1)

    # Decode the last third for z
    value_z = sum((2**p) * chromosome[(2 * third_length) - 1 - p] for p in
range(third_length))
    z = a + (b - a) * float(value_z) / (N_chains - 1)

    return x, y, z
```

Ahora, la funcion de aptitud es la funcion de Rastrigin.

```
def rastrigin(x, y, z):
    return 10 * 3 + (x**2 - 10 * np.cos(2 * np.pi * x)) + (y**2 - 10 *
np.cos(2 * np.pi * y)) + (z**2 - 10 * np.cos(2 * np.pi * z))
```

Además, se modificó la funcion para representar la grafica y se realizo en una sola. Donde se muestra la funcion de Rastrigin y las poblaciones en dos dimensiones. Aunque la optimización se realice en 3 dimensiones.

```
# Graph the population and Rastrigin surface in 2D with color depth
def graph_population(F):
    x_population = []
    y_population = []
    z_population = []

    for chromosome in F:
        x, y, z = decode_chromosome(chromosome)
        x_population.append(x)
        y_population.append(y)
        z_population.append(z)

    fig, ax = plt.subplots()
```

```
    # Create grid for the surface
    x = np.linspace(a, b, 500)
    y = np.linspace(a, b, 500)
    X, Y = np.meshgrid(x, y)
    Z = rastrigin(X, Y, np.zeros_like(X))  # 2D Rastrigin with z=0

    # Plot the surface as a 2D heatmap
    contour = ax.contourf(X, Y, Z, cmap='viridis')
    plt.colorbar(contour)

    # Scatter plot of the population points
    ax.scatter(x_population, y_population, color='red',
label='Population')

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_title('Population on Rastrigin Function (2D with depth)')

    plt.legend()
    plt.show()
```

Ejecución

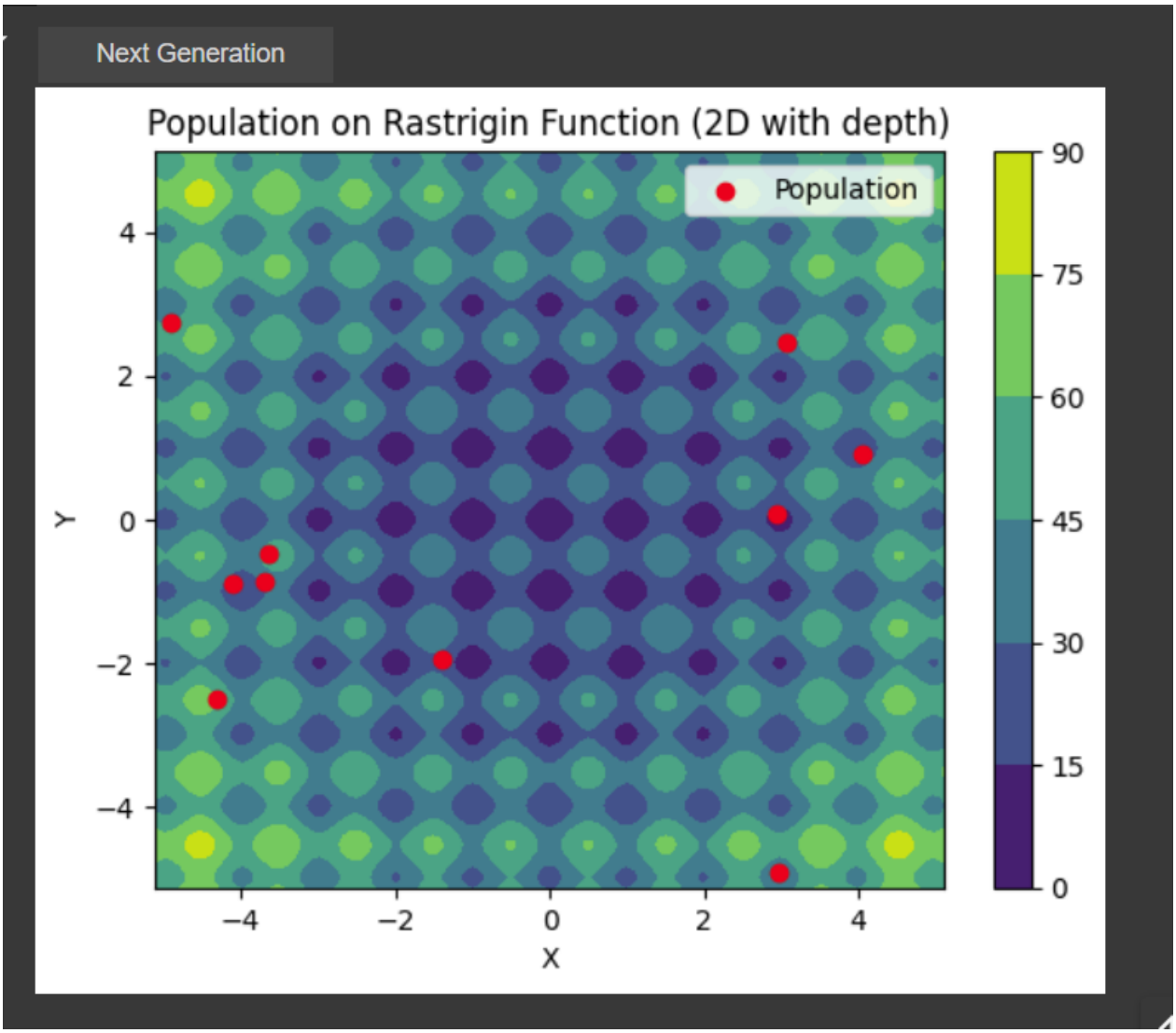

Ilustración 4. Generación inicial de la optimización de la funcion de Rastrigin.

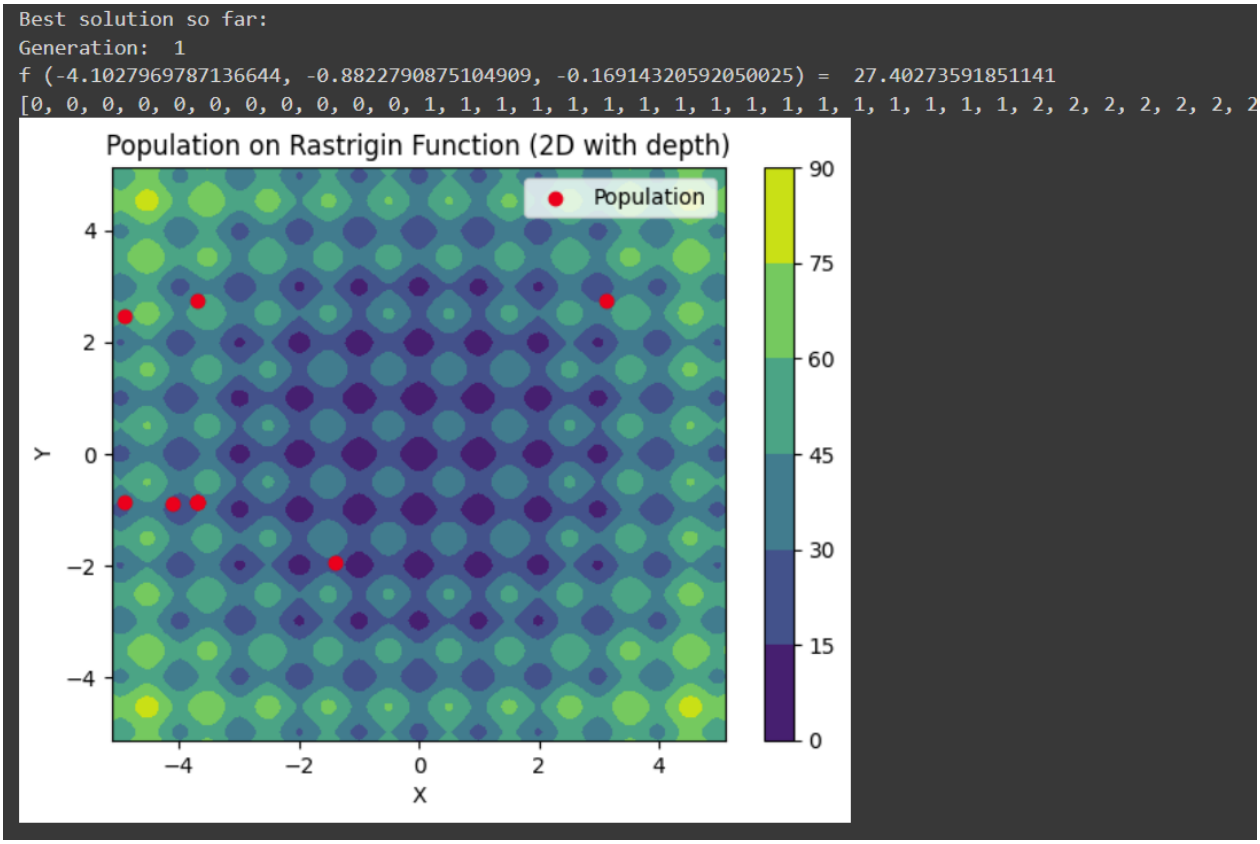

Ilustración 5. Primera generación de la optimización de la funcion de Rastrigin.

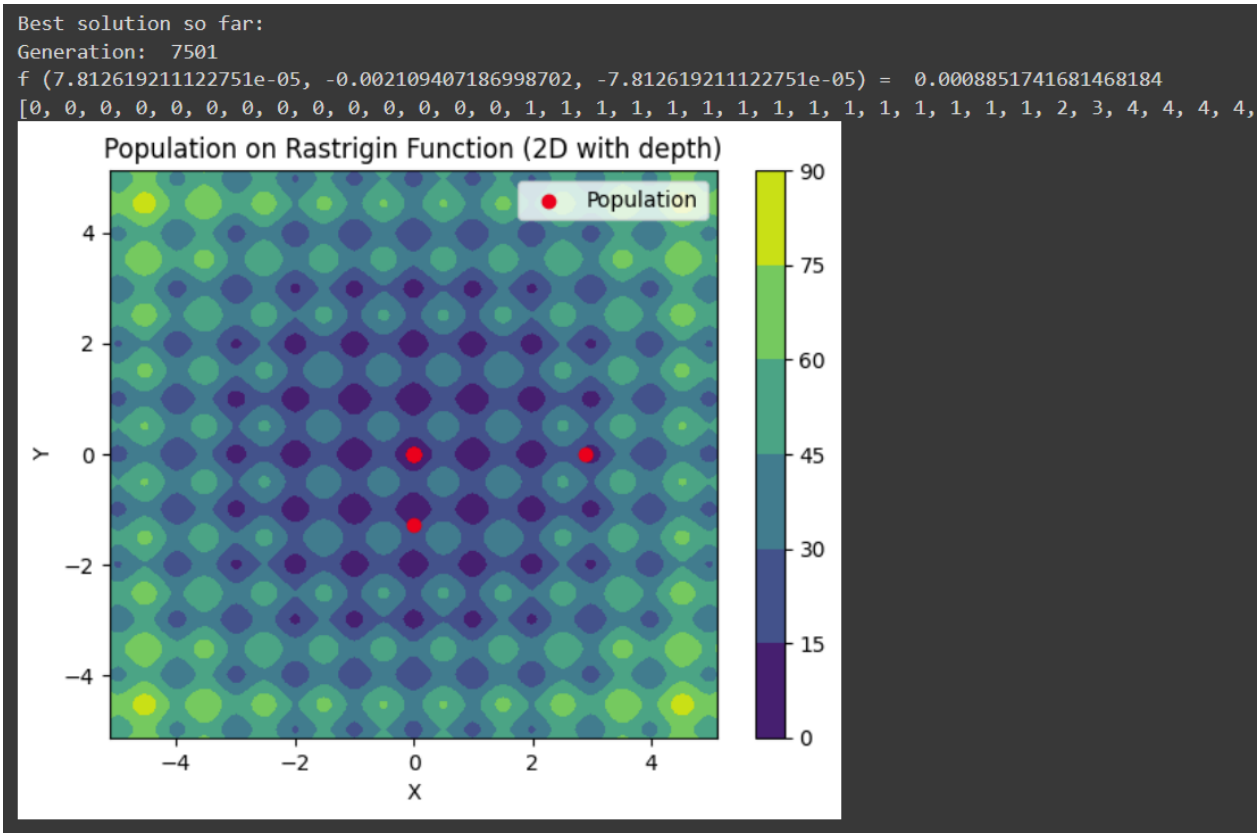

Ilustración 6. Generación 7501 de la optimización de la funcion de Rastrigin.

3. CONCLUSIONES

En esta práctica se abordó la optimización de funciones complejas utilizando algoritmos genéticos, específicamente las funciones de Ackley en dos dimensiones y Rastrigin en tres dimensiones. Durante el desarrollo, se pudo observar cómo los operadores genéticos (selección, cruce y mutación) son fundamentales para la evolución de la población y cómo la diversidad de soluciones permite evitar caer en mínimos locales. Se logró comprender que el balance adecuado de mutación es crucial para mantener la exploración sin perder la calidad de las soluciones.

Una observación importante es que la visualización gráfica de la evolución de la población permitió identificar patrones de convergencia y los puntos de mejora. Para futuras prácticas, sería útil explorar configuraciones adicionales de parámetros, como ajustar las tasas de mutación y cruce, o implementar estrategias avanzadas de selección como el torneo. Esto podría mejorar la velocidad de convergencia sin comprometer la diversidad.

Además, se experimentó con diferentes tamaños de poblaciones, lo que proporcionó una mejor comprensión de cómo afectan la eficiencia del algoritmo y la capacidad de explorar el espacio de soluciones. Estos experimentos demostraron que es fundamental encontrar un equilibrio entre la exploración y la explotación para lograr una convergencia efectiva.

4. BIBLIOGRAFÍA

Ackley Function. (s/f). Sfu.ca. Recuperado el 6 de octubre de 2024, de https://www.sfu.ca/~ssurjano/ackley.html

Rastrigin Function. (s/f). Sfu.ca. Recuperado el 6 de octubre de 2024, de https://www.sfu.ca/~ssurjano/rastr.html