UNIVERSIDAD AUTÓNOMA DE QUERÉTARO
FACULTAD DE INGENIERÍA
DIVISIÓN DE INVESTIGACIÓN Y POSGRADO

Algoritmos Metaheurísticos: Examen 1. Algoritmo Genético: Minimizar función de 2 variables.

Alumno:
Ing. Enrique Mena Camilo

Profesor:
Dr. Marco Antonio Aceves Fernández

Octubre 2023

Índice

1. Objetivos

El objetivo de esta práctica consiste en encontrar el mínimo (o mínimos) de una función de varias variables. Esta solución deberá desarrollarse mediante el uso de algoritmos genéticos bajo las siguientes consideraciones:

- Deberá diseñar un método de codificación apto para el problema.
- Deberá incluir métodos de selección.
- Deberá incluir métodos métodos de cruza.
- Deberá incluir criterios de paro.
- Deberá incluir métodos de mutación.
- Deberá incluir criterio elitista.

2. Introducción

La *función de Rosenbrock*, también conocida como la *función de Rosenbrock's valley*, es una función matemática utilizada comúnmente como un problema de prueba en optimización y algoritmos de búsqueda. Esta función es conocida por ser un desafío para muchos algoritmos de optimización debido a su forma de valle estrecho y plana. La función de Rosenbrock está definida de la siguiente manera en dos dimensiones:

$$f(x_1, x_2) = b * (x_2 - x_1^2)^2 \ + (x_1 - a)^2$$

Donde a y b son constantes (generalmente $a = 1$ y $b = 100$ para el caso más común), mientras x_1 y x_2 son las variables de la función que se optimizan. El mínimo global de esta función se encuentra en $f(x_1, x_2) = 0$, en el punto $(x_1, x_2) = (a, a^2)$, que para el caso más común sería $(x_1, x_2) = (1, 1)$.

Los algoritmos genéticos son útiles para abordar problemas de optimización, como la función de Rosenbrock, por varias razones:

1. Exploración del espacio de búsqueda: Los algoritmos genéticos utilizan una población de soluciones candidatas que evoluciona con el tiempo. Esto permite explorar diferentes regiones del espacio de búsqueda en paralelo, lo que es beneficioso en funciones como la de Rosenbrock, que tienen valles estrechos y planos.

2. Diversificación: La selección y la operación de cruza en algoritmos genéticos fomentan la diversificación de soluciones en la población. Esto es importante en la función de Rosenbrock, ya que puede ayudar a evitar la convergencia prematura hacia soluciones locales subóptimas.

3. Adaptación gradual: Con el tiempo, los individuos de la población tienden a converger hacia soluciones más óptimas a medida que se transmiten los genes de los individuos más aptos. En el caso de la función de Rosenbrock, este proceso gradual puede permitir que la población se acerque al mínimo global.

4. Flexibilidad: Los algoritmos genéticos son flexibles y pueden adaptarse a diferentes problemas de optimización

3. Marco teórico

3.1. Roulette Selection

El mecanismo de selección de ruleta, es una técnica popularmente usada en algoritmos genéticos para seleccionar posibles soluciones de acuerdo a su aptitud. La idea es dar a cada individuo una probabilidad de ser seleccionado que sea proporcional a su aptitud, de tal manera que los individuos con mayor aptitud tengan una mayor probabilidad de ser elegidos, pero aún permitiendo que los individuos con menor aptitud tengan alguna posibilidad.

En términos generales, este método consta de los siguientes pasos:

1. Cálculo de Aptitudes: En primer lugar, se calcula la aptitud de cada individuo en la población.
2. Suma de Aptitudes: Se suman todas las aptitudes para obtener una aptitud total de la población.
3. Cálculo de Probabilidades: Se determina la probabilidad de selección de cada individuo dividiendo su aptitud por la aptitud total de la población.
4. Selección: Se genera un número aleatorio entre 0 y 1. Luego, se selecciona el individuo cuya probabilidad acumulada incluye este número aleatorio. En otras palabras, imaginamos que giramos una ruleta donde cada segmento corresponde a un individuo, y el tamaño de cada segmento es proporcional a la aptitud del individuo.

Una representación gráfica de este mecanismo se puede observar en la Figura 1.

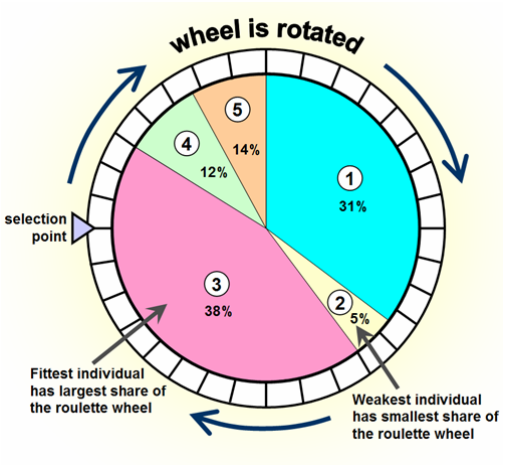


Figura 1: Diagrama de funcionamiento de selección por ruleta.

3.2. Two Point Crossover

La cruza de dos puntos (two point crossover, en inglés) es un método de recombinación comúnmente utilizado en algoritmos genéticos que permite la combinación de características de dos padres para generar una descendencia. Este método permite generar combinaciones de características de ambos padres para explorar y buscar mejores soluciones. En terminos generales, este método consta de los siguientes pasos:

1. Definición de Puntos de Corte: Se seleccionan dos puntos de corte en la representación de los padres. Estos puntos dividen a los padres en tres segmentos: el segmento antes del primer punto de corte, el segmento entre los dos puntos de corte y el segmento después del segundo punto de corte.

2. Creación de Descendencia: La descendencia se crea intercambiando los segmentos intermedios (el segmento entre los dos puntos de corte) de los dos padres. El resultado es un nuevo individuo que combina partes de ambos padres.

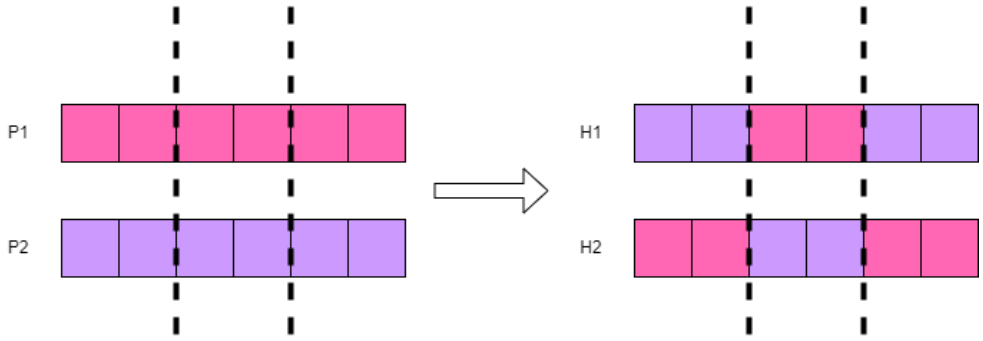


Figura 2: Mecanismo de cruza de dos puntos.

3.3. Scramble Mutation

La mutación por mezcla (scramble mutation, en inglés) es un proceso que introduce pequeños cambios aleatorios en los individuos de una población con el objetivo de aumentar la diversidad genética y explorar nuevas soluciones en el espacio de búsqueda. Dicho proceso se caracteriza por los siguientes pasos:

1. Selección de un subconjunto de genes: Se elige un subconjunto aleatorio de genes en la representación del individuo. Los genes seleccionados formarán parte del proceso de mutación.

2. Mezcla de genes: Los genes seleccionados se reordenan de manera aleatoria entre sí. Este reordenamiento aleatorio puede ser realizado de diferentes maneras, como permutando los valores de los genes dentro del subconjunto o cambiando su posición en la secuencia.

Podemos observar una representación gráfica de este mecanismo en la Figura 3.

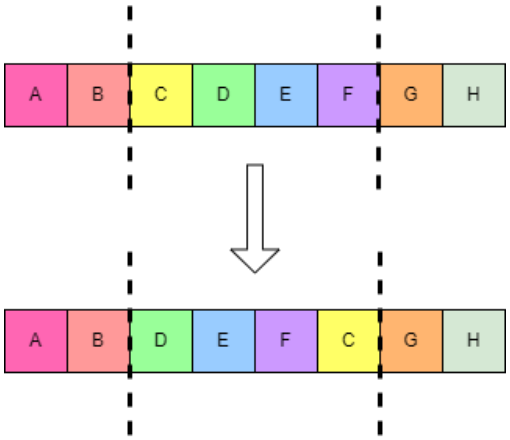


Figura 3: Diagrama de funcionamiento de scramble mutation.

3.4. Competencia genética

El concepto de competencia genética se asemeja a la lucha por la supervivencia en la naturaleza, donde los individuos más aptos tienen una mayor probabilidad de sobrevivir y reproducirse, transmitiendo así sus genes a la siguiente generación. La idea principal detrás de la competencia genética en algoritmos evolutivos es simular este proceso de selección natural para buscar soluciones óptimas o mejores en un espacio de búsqueda.

En este proceso se toma a todos los padres y a todos los hijos, se ordenan de mayor a menor aptitud, y solamente se permitirá que pasen a la siguiente generación aquellos que presentan mejor aptitud, sin importar sin son padres o hijos.

4. Materiales y métodos

4.1. Algoritmo genético

Para esta práctica se desarrolló un algoritmo genético que se apega al proceso mostrado en la Figura 4.

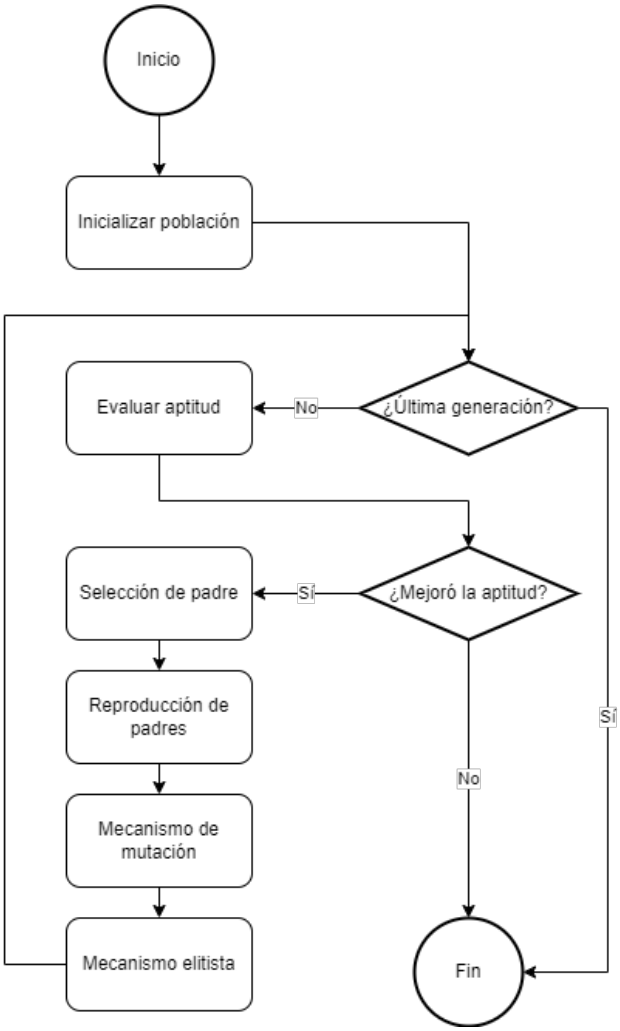


Figura 4: Diagrama de flujo del algoritmo genético implementado.

Para llevar a cabo dicho proceso, se optó por utilizar el lenguaje de programación Python, donde se diseñaron métodos genéricos para realizar los procesos de generación de población, evaluación de aptitud, selección de individuos, reproducción de individuos, mutación y mecanismos elitistas.

4.2. Implementación en Python

4.2.1. Inicializador de población

```python
def int_to_oct_array(number: int, padding: int = 8):
    oct_str = oct(number)[2:].zfill(padding)
    oct_array = np.zeros(padding)
    for idx in range(padding):
        oct_array[idx] = oct_str[idx]

    return oct_array


def oct_array_to_int(oct_array: np.ndarray):
    n = oct_array.shape[0]
    pows = [8**pow for pow in range(n)]
    pows = np.array(pows[::-1])
    int_number = np.dot(oct_array, pows)

    return int_number


def init_octal_population(n: int = 10, genes: int = 10):
    min_octal = 0o0
    max_octal = 0o7 * (8**genes)
    population = np.random.randint(min_octal, max_octal, n, dtype=np.uint64)
    population = [int_to_oct_array(individue, genes) for individue in population]
    population = np.array(population)

    return population
```

4.2.2. Evaluación de individuos

```python
def aptitude_function(x1, x2):
    return 100*(x2 - x1**2)**2 + (x1 - 1)**2


def decode_individue(individue: np.ndarray):
    min_domain = -5
    max_domain = 10
    bits = individue.shape[0]
    min_octal = 0
    max_octal = 8**bits - 1
    decoded_individue = oct_array_to_int(individue)
    decoded_individue = (decoded_individue - min_octal) / (max_octal - min_octal)
    decoded_individue = (max_domain - min_domain) * decoded_individue + min_domain

    return decoded_individue


def evaluate_individue(x1: np.ndarray, x2: np.ndarray):
    x1_decoded = decode_individue(x1)
    x2_decoded = decode_individue(x2)
    aptitude = aptitude_function(x1_decoded, x2_decoded)

    return aptitude


def evaluate_population(p_x1: np.ndarray, p_x2: np.ndarray, desc: bool = False):
    aptitudes = np.vstack([evaluate_individue(x1, x2) for x1, x2 in zip(p_x1, p_x2)])

    direction = -1 if desc else 1
    sorted_indexes = aptitudes[:, -1].argsort()[::direction]
    sorted_px1 = p_x1[sorted_indexes]
    sorted_px2 = p_x2[sorted_indexes]
    sorted_aptitudes = aptitudes[sorted_indexes]
    avg_aptitude = np.mean(sorted_aptitudes)
    best_aptitude = np.min(sorted_aptitudes)

    return sorted_px1, sorted_px2, sorted_aptitudes, avg_aptitude, best_aptitude
```

4.2.3. Selección de parejas

```python
def roulette_selection(aptitudes: np.ndarray):
    max_aptitude = np.max(aptitudes)
    fitness = max_aptitude - aptitudes
    total_fitness = fitness.sum()

    couples = []
    couple = []

    for _ in range(aptitudes.shape[0]):
        accumulated = 0
        selection = np.random.uniform(0, total_fitness)

        for idx, aptitude  in enumerate(fitness):
            accumulated += aptitude
            if accumulated >= selection:
                couple.append(idx)
                break

        if len(couple) == 2:
            couples.append(couple)
            couple = []

    return np.vstack(couples)
```

4.2.4. Reproducción de individuos

```python
def two_point_crossover(population: np.ndarray, parents: np.ndarray):
    childrens = np.empty_like(population)
    for idx, couple in enumerate(parents):
        crossover_point = population.shape[1]//3
        childrens[idx*2, :crossover_point] = population[couple[0], :crossover_point]
        childrens[idx*2, crossover_point:2*crossover_point] = population[couple[1], crossover_point:2*cr
        childrens[idx*2, 2*crossover_point:] = population[couple[0], 2*crossover_point:]
        childrens[idx*2+1, :crossover_point] = population[couple[1], :crossover_point]
        childrens[idx*2+1, crossover_point:2*crossover_point] = population[couple[0], crossover_point:2*
        childrens[idx*2+1, 2*crossover_point:] = population[couple[1], 2*crossover_point:]

    return childrens
```

4.2.5. Mutación

```python
def scramble_mutation(individues_x1: np.ndarray, individues_x2: np.ndarray, mr: float = 0.10):
    total_individues = individues_x1.shape[0]
    individue_len = individues_x1.shape[1]
    to_mutate = np.random.choice(total_individues, int(total_individues * mr))

    for idx in to_mutate:
        idx_l, idx_r = sorted(np.random.choice(individue_len, 2))
        idx_r = idx_r if idx_r == individue_len else idx_r + 1
        individues_x1[idx][idx_l:idx_r] = np.random.shuffle(individues_x1[idx][idx_l:idx_r][::-1])
        individues_x2[idx][idx_l:idx_r] = np.random.shuffle(individues_x2[idx][idx_l:idx_r][::-1])

    return individues_x1, individues_x2
```

4.2.6. Elitismo

```python
def genetic_competence(p_x1: np.ndarray, p_x2: np.ndarray, c_x1: np.ndarray, c_x2: np.ndarray):
    all_px1 = np.vstack([p_x1, c_x1])
    all_px2 = np.vstack([p_x2, c_x2])
    sorted_px1, sorted_px2, _, _, _ = evaluate_population(all_px1, all_px2)

    return sorted_px1[:p_x1.shape[0]], sorted_px2[:p_x2.shape[0]]
```

4.2.7. Algoritmo completo

```python
generations = 20

population_x1 = init_octal_population(100, 10)
population_x2 = init_octal_population(100, 10)

last_avg_aptitude = float("inf")
max_stagnant_generations = 2
stagnant_generations = 0
delta = 0.05

avg_aptitudes = []
best_aptitudes = []
evolution = []


```

```python
for _ in tqdm(range(generations), desc="Generation"):
    population_x1, population_x2, aptitudes, avg_aptitude, best_aptitude = evaluate_population(populatio
    avg_aptitudes.append(avg_aptitude)
    best_aptitudes.append(best_aptitude)
    evolution.append([population_x1, population_x2, best_aptitude, avg_aptitude])

    if abs(avg_aptitude - last_avg_aptitude) < delta:
        stagnant_generations += 1
        if stagnant_generations > max_stagnant_generations:
            break
    else:
        stagnant_generations = 0

    last_avg_aptitude = avg_aptitude
    parents = roulette_selection(aptitudes)
    childrens_x1 = two_point_crossover(population_x1, parents)
    childrens_x2 = two_point_crossover(population_x2, parents)
    childrens_x1, childrens_x2 = scramble_mutation(childrens_x1, childrens_x2)
    population_x1, population_x2 = genetic_competence(population_x1, population_x2, childrens_x1, childr

print(f"Best individue: {decode_individue(evolution[-1][0][0])}, {decode_individue(evolution[-1][1][0])}
print(f"Best aptitude: {evolution[-1][2]}")
print(f"Avg aptitude: {evolution[-1][3]}")

title = "Roulette selection + two point crossover"
plot_aptitude(avg_aptitudes, best_aptitudes, generations, title)
filename = plot_evolution(evolution, title)
HTML(f'<img src="{filename}">')
```

4.3. Pruebas realizadas

Se implementó una solución en la cuál se utilizan poblaciones con codificación octal, selección mediante el método ruleta, cruza de dos puntos, mutación scramble y competencia genética.

Debido a que el objetivo es determinar un conjunto de soluciones, se realizaron 3 ejecuciones del algoritmo, esto con el objetivo de obtener soluciones posibles al problema.

En cada ejecución del algoritmo se generaron gráficas que muestran el proceso de de evolución, en términos de aptitud y convergencia dentro del espacio de búsqueda.

El criterio de paro utilizado se estableción según los siguientes criterios:

- Número de generaciones. Se estableció un límite de 20 generaciones para todos los algoritmos.
- Término por δ. Se estableció un valor $\delta = 0{,}05$ con un máximo de 2 generaciones.

5. Resultados

Relacionado a la primer ejecución del algoritmo, este tomó un total de 11 épocas para lograr la convergencia, obteniendo los siguientes resultados:

- Mejor individuo: $(1{,}2670, 1{,}6521)$
- Aptitud: 0,2907

La Figura 5 muestra el proceso de evolución en términos de aptitud, mientras que la Figura 6 muestra una comparación entre los individuos durante la primer época contra los individuos de la última época.

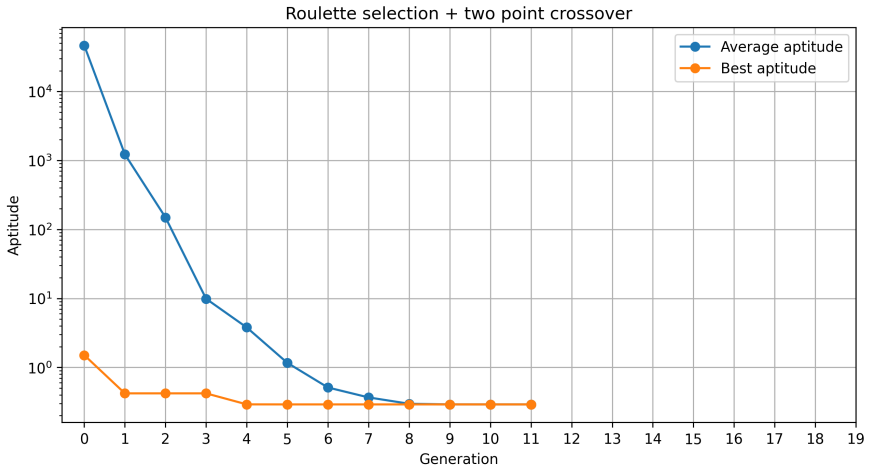
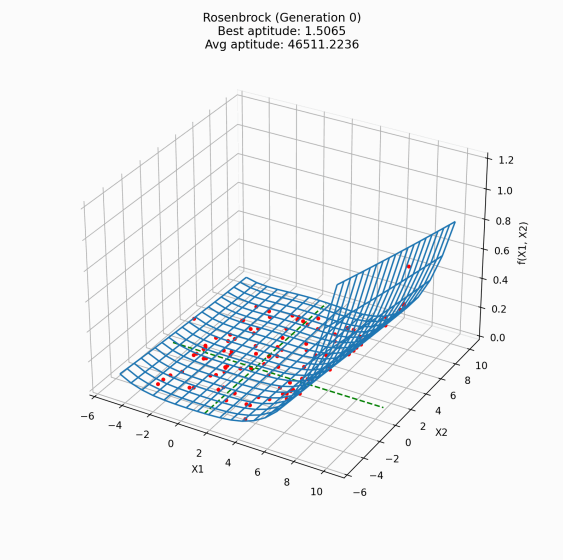
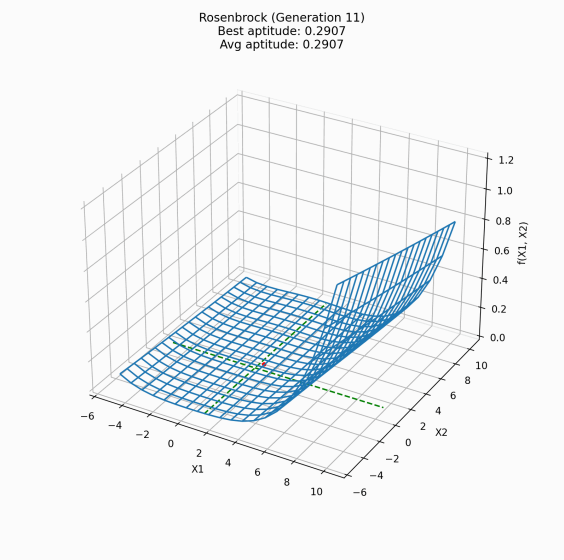


Figura 5: Evolución de la aptitud de los individuos en primer ejecución.



(a) Individuos en la primer época.



(b) Individuos en la última época.

Figura 6: Comparación de los individuos entre la primer y última época de la primer ejecución del algoritmo.

Relacionado a la segunda ejecución del algoritmo, este tomó un total de 10 épocas para lograr la convergencia, obteniendo los siguientes resultados:

- Mejor individuo: $(1{,}3542, 1{,}8367)$
- Aptitud: 0,1262

La Figura 7 muestra el proceso de evolución en términos de aptitud, mientras que la Figura 8 muestra una comparación entre los individuos durante la primer época contra los individuos de la última época.

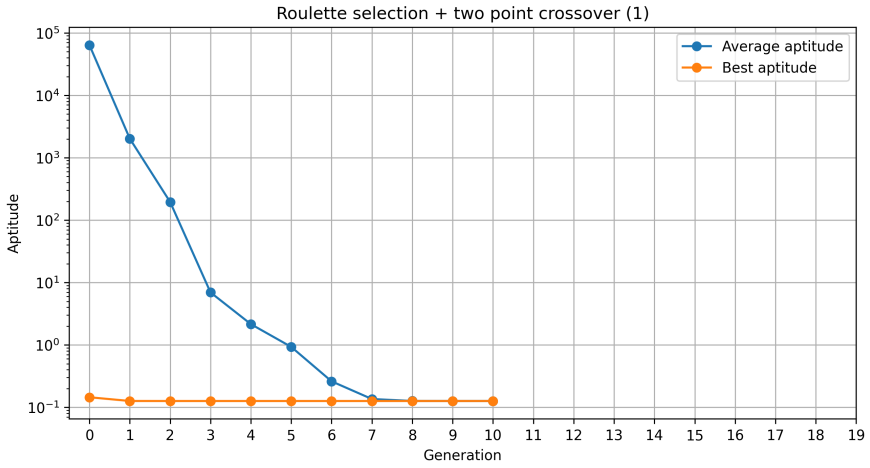
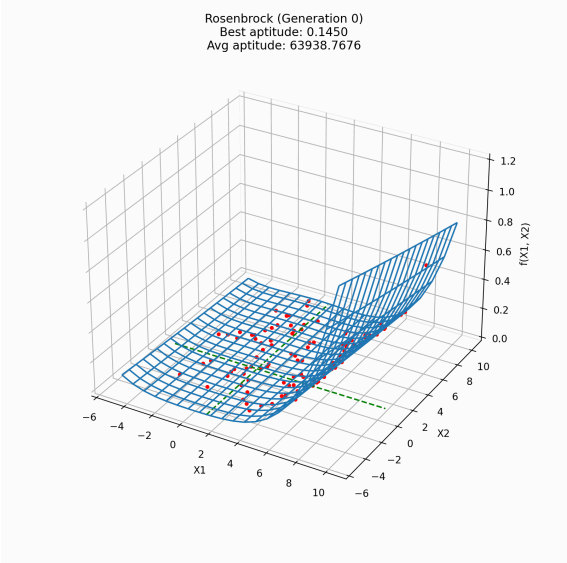
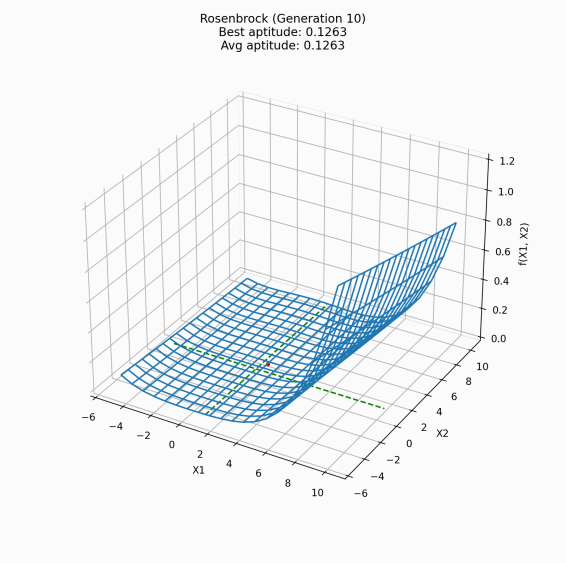


Figura 7: Evolución de la aptitud de los individuos en segunda ejecución.



(a) Individuos en la primer época.



(b) Individuos en la última época.

Figura 8: Comparación de los individuos entre la primer y última época de la segunda ejecución del algoritmo.

Relacionado a la tercera ejecución del algoritmo, este tomó un total de 11 épocas para lograr la convergencia, obteniendo los siguientes resultados:

- Mejor individuo: $(0{,}8287, 0{,}7170)$
- Aptitud: 0,1203

La Figura 9 muestra el proceso de evolución en términos de aptitud, mientras que la Figura 10 muestra una comparación entre los individuos durante la primer época contra los individuos de la última época.

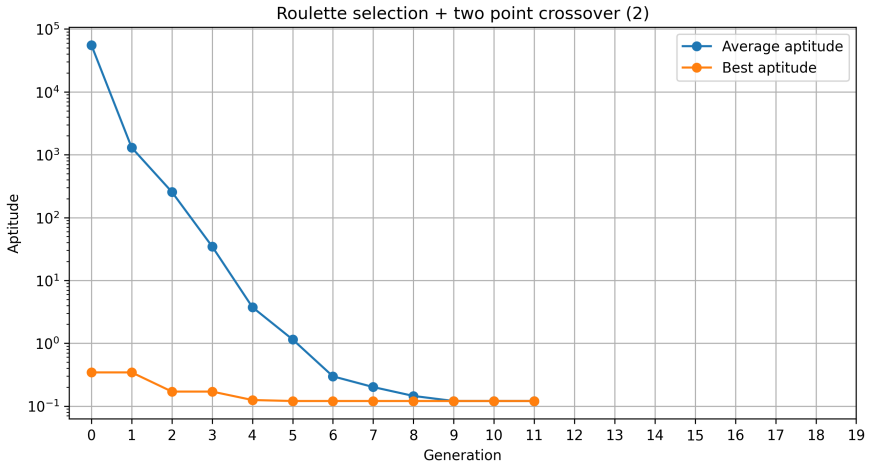
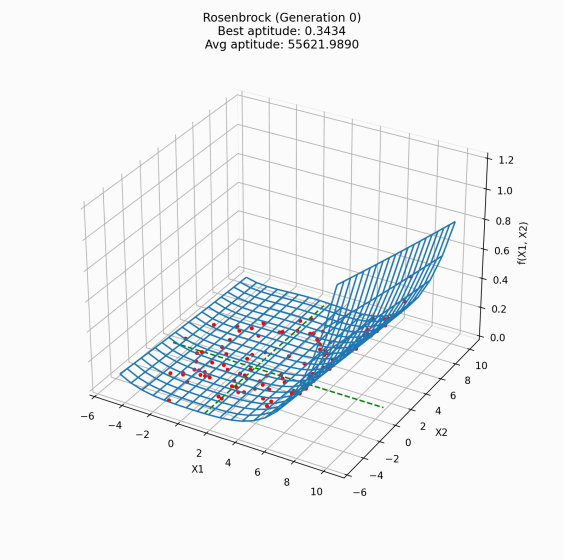
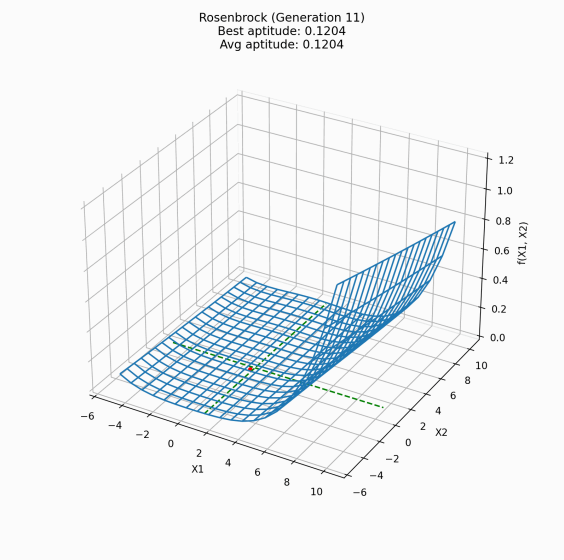


Figura 9: Evolución de la aptitud de los individuos en tercera ejecución.



(a) Individuos en la primer época.



(b) Individuos en la última época.

Figura 10: Comparación de los individuos entre la primer y última época de la tercera ejecución del algoritmo.

6. Discusión y conclusiones

Dado nuestro conocimiento previo, sabemos que el mínimo de la función de Rosenbrock se encuentra en $(x_1, x_2) = (1, 1)$, donde el valor de la función es $f(x_1, x_2) = 0$. Tomando esto como referencia, podemos considerar los valores de aptitud obtenidos por el algoritmo como una métrica de error. En este caso, hemos obtenido un error promedio de 0,1791, lo cual es un valor bastante aceptable.

A pesar de que el algoritmo no logró encontrar el mínimo exacto de la función, logró acercarse significativamente. La limitación que enfrentamos podría estar relacionada con la elección de la codificación utilizada, en este caso, una codificación octal. Es posible que el valor preciso del mínimo no se encuentre dentro de los valores posibles para la codificación, o que la precisión generada por esta codificación ocasione que el algoritmo oscile en valores muy cercanos al mínimo real.

En conclusión, hemos demostrado la utilidad de los algoritmos genéticos para determinar mínimos en una función, incluso en un caso de dos variables como la función de Rosenbrock. Esto abre la puerta para utilizar estas técnicas heurísticas en la búsqueda de mínimos o máximos en una amplia variedad de funciones, lo que puede tener aplicaciones prácticas en la resolución de problemas del mundo real. Aunque no hemos alcanzado el mínimo exacto en este caso, hemos demostrado que estos algoritmos pueden ser valiosos en la optimización de funciones complejas.

Referencias bibliográficas

[1] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*, 2nd ed. Springer, 2015, ISBN: 9783662448731.

[2] K. Dahal, K. C. Tan, and P. I. Cowling, *Evolutionary Scheduling*, 1st ed. Springer, 2007, ISBN: 9783540485827.

[3] S. N. Sivanandam and S. N. Deepa, *Introduction to Genetic Algorithms*, 1st ed. Springer, 2008, ISBN: 9783540731894.