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

Algoritmos Metaheurísticos: Práctica 1. Algoritmo Genético.

Alumno:
Ing. Enrique Mena Camilo

Profesor:
Dr. Marco Antonio Aceves Fernández

Agosto 2023

Índice

1. Objetivos

El objetivo de esta práctica consisten en plasmar las bases de los algoritmos metaheurísticos, en especial de los algoritmos genéticos. Se buscará implementar un algoritmo genético de naturaleza dicotómica siguiendo las siguientes reglas:

- Deberá contemplar 6 pobladores.
- Deberá realizar codificación mediante 6 genes.
- Deberá utilizar la función de aptitud x^2.
- Deberá implementar 2 métodos de selección.
- Deberá implementar 2 métodos de cruza.
- Deberá implementar 2 criterios de paro o 1 híbrido.
- Deberá implementar al menos 1 criterio elitista.
- Se podrá implementar un proceso de mutación de forma opcional.

2. Introducción

Los algoritmos genéticos son una técnica de inteligencia artificial que se inspira en la evolución biológica y la genética para resolver problemas complejos. Estos algoritmos se basan en la idea de que el individuo más adaptado al medio es el que sobrevive y transmite sus características a sus descendientes. Así, los algoritmos genéticos simulan el proceso de selección natural mediante la manipulación de cadenas de símbolos que representan posibles soluciones a un problema dado.

Los algoritmos genéticos han tenido un impacto significativo en el campo de la computación y la inteligencia artificial. Su enfoque de búsqueda basado en la evolución ha demostrado ser efectivo para resolver problemas complejos en los que los métodos tradicionales pueden ser ineficientes o inadecuados. Los algoritmos genéticos han inspirado muchas otras técnicas de optimización y metaheurísticas, lo que ha llevado al desarrollo de una amplia gama de algoritmos evolutivos.

Algunas de sus aplicaciones notables incluyen:

- Diseño de Circuitos Electrónicos.
- Optimización de Procesos Industriales.
- Diseño de Redes y Enrutamiento.
- Diseño de Estructuras.
- Aprendizaje Automático.

3. Marco teórico

3.1. Algoritmo Genético

En terminos generales, un algoritmo genético se compone de 8 pasos:

1. **Generar población inicial.** Se debe elegir de forma aleatoria n posibles soluciones dentro del dominio de nuestro problema.

2. **Codificar dominio.** Una vez seleccionadas las muestras, se deberá realizar un proceso de codificación, donde se buscará expresar a las posibles soluciones en términos de genes, individuos y población.

3. **Evaluar aptitud.** Teniendo una población, se deberá evaluar la aptitud de estos mediante la definición de una función de aptitud. Esta función suele ser inversa a las funciones de costo.

4. **Ordenar individuos.** La mayoría de los algoritmos genéticos suele seleccionar a las individuos más aptos, por lo que se recomienda ordenar de forma descendente a la población.

5. **Seleccionar parejas.** Es un proceso en el que se forman parejas, las cuales en pasos posteriores se someterán a proceso de reproducción en los que se buscará mejorar los resultados.

6. **Reproducción de individuos.** Habiendo seleccionado a los individuos que se reproducirán, se deberá definir un método mediante el cual se mezcle la información de estos.

7. **Mutación.** En la naturaleza suele ocurrir en una proporción muy pequeña procesos de mutación, estos procesos consisten en proveer a los individuos hijos de características que no poseen los padres, lo cuál en algunas ocasiones puede generar una mejora en su aptitud.

8. **Evaluar nueva generación.** Una vez que se ha realizado el proceso de cruza, se deberá evaluar que tan buenos son los individuos resultantes de la reproducción, esto será útil para en procesos posteriores determinar cuales serán los individuos que prevalezcan.

3.2. Mecanismos de selección

Los mecanismos de selección permiten determinar de forma metódica cuáles serán los individuos que realizarán su reproducción.

3.2.1. Selección poligámica aleatoria

En la selección poligámica aleatoria se forma parejas de forma aleatoria, sin importar si algún individuo es emparejado más de 1 vez. Una desventaja de este método es que existe la posibilidad de no seleccionar a un individuo, lo cuál podría ocasionar que no se explore completamente el espacio de búsqueda. En la Figura 1 podemos observar una representación gráfica de este mecanismo de selección.

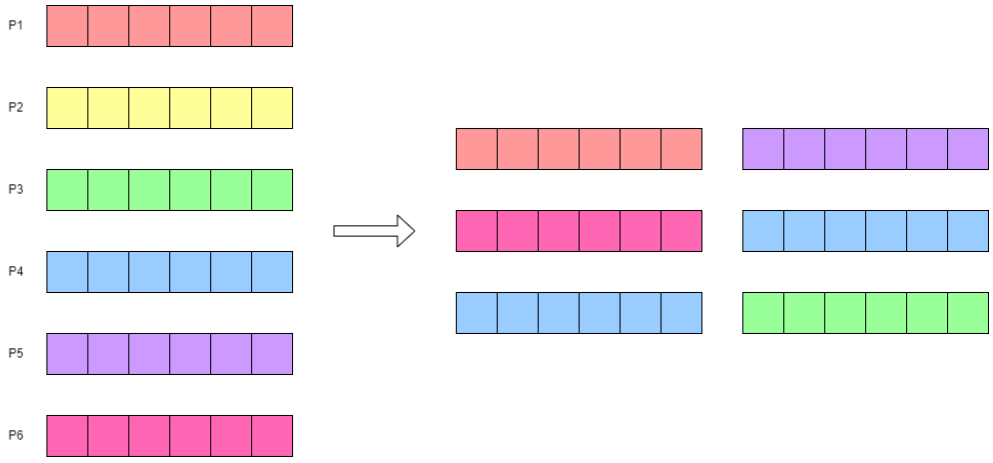

Figura 1: Mecanismo de selección poligámica aleatoria.

3.2.2. Selección monogámica aleatoria

Es un mecanismo similar a la selección monogámica aleatoria, con la peculiaridad de que una vez que un individuo a sido seleccionado, no se puede volver a seleccionar en esa generación. Este algoritmo garantiza que todos los individuos se reproduzcan, sin embargo, esto podría traer el problema de que prevalezcan individuos no aptos. La Figura 2 muestra una representación gráfica de este proceso.

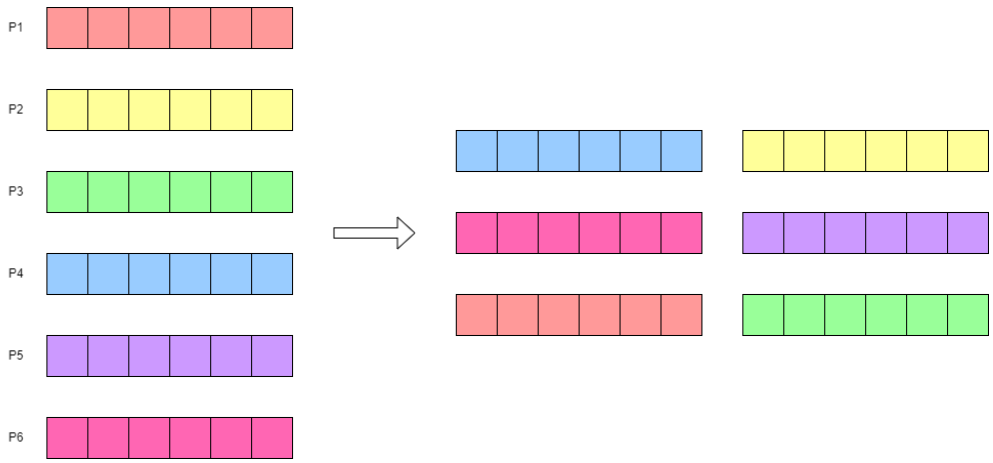

Figura 2: Mecanismo de selección monogámica aleatoria.

3.3. Mecanismos de reproducción

Los mecanismos de reproducción establecen las reglas mediante las cuales los individuos mezclarán sus rasgos genéticos.

3.3.1. Cruza de un punto

En la cruza de un punto se define un punto por el que los individuos será dividido, usualmente se escoge la mitad de estos. Una vez que se han generado los cortes, se procede a realizar una combinación cruzada de estos. La Figura 3 muestra este proceso de forma gráfica.

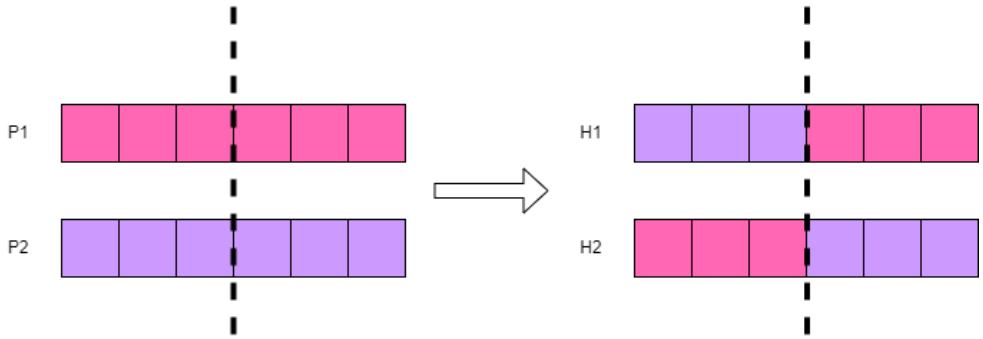

Figura 3: Mecanismo de cruza de un punto.

3.3.2. Cruza de dos puntos

Un proceo de cruza similar a la cruza de un punto, sin embargo, en este caso de definen dos puntos de corte y se realiza una combinación en forma de V entre los individuos. En la Figura 4 se puede observar este proceso.

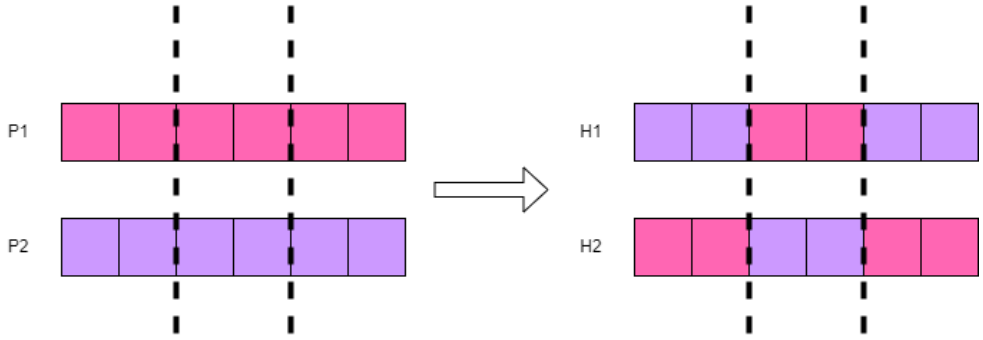

Figura 4: Mecanismo de cruza de dos puntos.

3.4. Criterios elitistas

La base de los algoritmos genéticos es la idea de que los individuos más aptos serán los que logren reproducirse y pasar su información genética a la siguiente generación, es por ello que los criterios elitistas buscan realizar un proceso de *depuración* de los individuos, permitiendo que solamente prevalezcan los individuos más aptos.

3.4.1. Competencia genética

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 dicotómico que se apega al proceso mostrado en la Figura 5

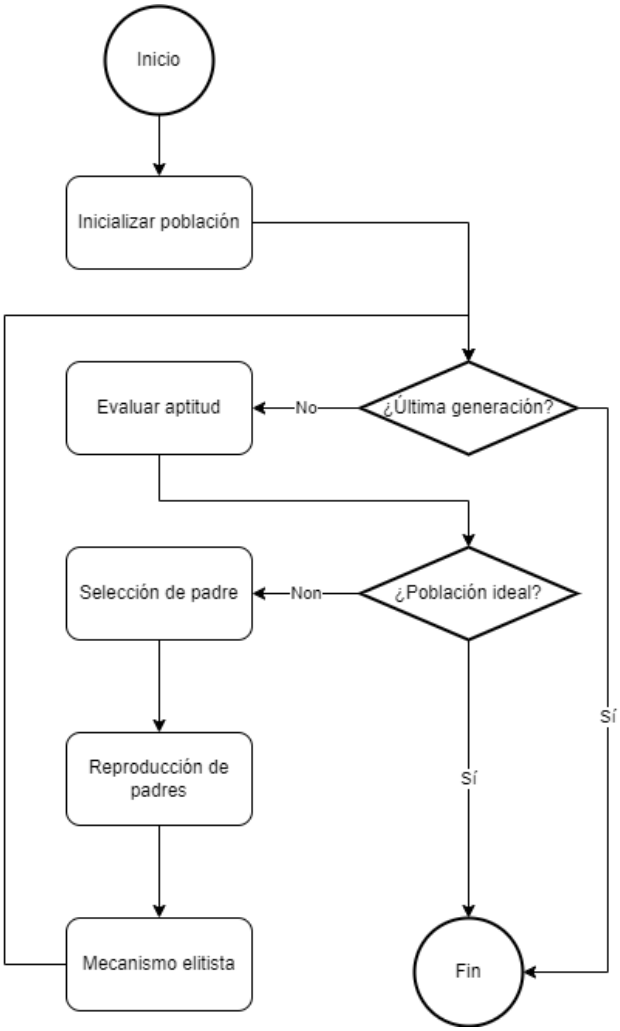

Figura 5: Diagrama de flujo del algoritmo genético implementado.

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 y mecanismos elitistas.

4.2. Implementación en Python

4.2.1. Inicializador de población

```python
#! /usr/bin/env python3
"""Population utilities."""

import numpy as np

from utils.casting import int_to_bin_array


def init_binary_population(n: int = 10, genes: int = 10):
    min = 0
    max = 2**genes
    population = np.random.randint(min, max, n)
    population = [int_to_bin_array(individue, genes) for individue in population]
    population = np.array(population, dtype=int)

    return population
```

4.2.2. Evaluación de individuos

```python
#! /usr/bin/env python3
"""Population utilities."""

import numpy as np

from utils.casting import bin_array_to_int


def evaluate_binary_aptitude(individue: np.ndarray, domain: tuple, aptitude_function: callable):
    bits = individue.shape[0]
    min = 0
    max = 2**bits - 1
    decoded_individue = bin_array_to_int(individue)
    decoded_individue = (decoded_individue - min) / (max - min)
    decoded_individue = (domain[1] - domain[0]) * decoded_individue + domain[0]
    aptitude = aptitude_function(decoded_individue)

    return aptitude


def evaluate_population(population: np.ndarray, domain: tuple, aptitude_function: callable, desc: bool = True):
    aptitudes = np.vstack([evaluate_binary_aptitude(individue, domain, aptitude_function) for individue in population])

    direction = -1 if desc else 1
    sorted_indexes = aptitudes[:, -1].argsort()[::direction]
    sorted_population = population[sorted_indexes]
    sorted_aptitudes = aptitudes[sorted_indexes]
    avg_aptitude = np.mean(sorted_aptitudes)
    max_aptitude = np.max(sorted_aptitudes)

    return sorted_population, sorted_aptitudes, avg_aptitude, max_aptitude
```

4.2.3. Selección de parejas

```python
#! /usr/bin/env python3
"""Selection utilities."""

import numpy as np


def polygamous_random_selection(population: np.ndarray):
    n_couples = population.shape[0]//2
    couples = np.random.choice(population.shape[0], (n_couples, 2), replace=True)

    return couples


def monogamous_random_selection(population: np.ndarray):
    n_couples = population.shape[0]//2
    couples = np.random.choice(population.shape[0], (n_couples, 2), replace=False)

    return np.array(couples)


def roulette_selection(population: np.ndarray):
    pass
```

4.2.4. Reproducción de individuos

```python
#! /usr/bin/env python3
"""Crossover utilities."""

import numpy as np


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

    return childrens


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*crossover_point]
        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*crossover_point]
        childrens[idx*2+1, 2*crossover_point:] = population[couple[1], 2*crossover_point:]

    return childrens

```

4.2.5. Mecanismo elitista

```python
#! /usr/bin/env python3
"""Elitist utilities."""

import numpy as np

from utils.aptitude import evaluate_population


def genetic_competence(population: np.ndarray, childrens: np.ndarray):
    all_population = np.vstack([population, childrens])
    sorted_population, _, _, _ = evaluate_population(all_population, (-1, 1), lambda x: x**2)
    return sorted_population[:population.shape[0]]
```

4.2.6. Algoritmo completo

```python
import sys

from IPython.display import HTML

from utils.population import init_binary_population
from utils.aptitude import evaluate_population
from utils.selection import polygamous_random_selection, monogamous_random_selection
from utils.crossover import one_point_crossover, two_point_crossover
from utils.elitist import genetic_competence
from utils.visualization import plot_aptitude, plot_evolution


initial_population = init_binary_population(6, 6)
generations = 10
epsilon = 0.8

population = initial_population.copy()
avg_aptitudes_p_op = []
max_aptitudes_p_op = []
evolution_p_op = []

for _ in range(generations):
    population, _, avg_aptitude, max_aptitude = evaluate_population(population, (-1, 1), lambda x: x**2)
    avg_aptitudes_p_op.append(avg_aptitude)
    max_aptitudes_p_op.append(max_aptitude)
    evolution_p_op.append(population.copy())

    if avg_aptitude > epsilon:
        break

    parents = polygamous_random_selection(population)
    childrens = one_point_crossover(population, parents)
    population = genetic_competence(population, childrens)

print(f"Best aptitude: {max_aptitudes_p_op[-1]}")
print(f"Avg aptitude: {avg_aptitudes_p_op[-1]}")

title = "Polygamous selection + one point crossover"
plot_aptitude(avg_aptitudes_p_op, max_aptitudes_p_op, generations, title)

filename = plot_evolution(evolution_p_op, title)
HTML(f'<img src="{filename}">')
```

4.3. Pruebas realizadas

Se implementaron un total de 2 mecanismos de selección, 2 mecanismos de cruza y 1 mecanismo elitista, por lo que se optó por realizar una combinación de todos los mecanismos diseñados para evaluar el desempeño de cada combinación. Las combinaciones resultantes fueron:

- **Algoritmo 1.** Selección monogámica aleatoria + cruza de un punto.
- **Algoritmo 2.** Selección monogámica aleatoria + cruza de dos puntos.
- **Algoritmo 3.** Selección poligámica aleatoria + cruza de un punto.
- **Algoritmo 4.** Selección poligámica aleatoria + cruza de dos puntos.

Cabe destacar que, dado que solamente se implementó 1 mecanismo elitista, este fue aplicado a todos los algoritmos por igual. Además, para garantizar una comparación equitativa, se utilizó la misma población inicial para cada uno de los algoritmos.

Es importante mencionar que, para garantizar que el algoritmo se detuviera, se implementaron 2 criterios de paro:

- **Número de generaciones.** Se estableció un límite de 10 generaciones para todos los algoritmos.
- **Término por ϵ.** Se estableció un valor $\epsilon = 0{,}8$ para determinar el fin del algoritmo.

5. Resultados

Relacionado a los resultados del algoritmo 1. Este algoritmo tardó un total de 3 generaciones en alcanzar el criterio de paro por épsilon, obteniendo una aptitud promedio final de 0,9583 y presentando a su mejor individuo con aptitud de 1,0000. El proceso de convergencia se puede observar en la Figura 6.

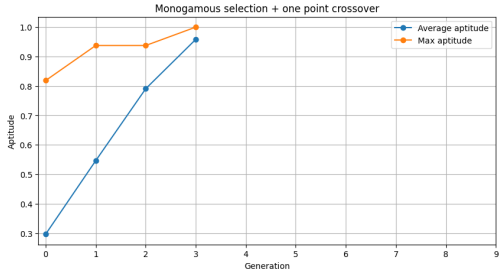

Figura 6: Evolución de la aptitud de los individuos del algoritmo 1.

Este algoritmo muestra un buen rendimiento en términos de aptitud promedio y mejor aptitud, ya que logra una aptitud promedio alta y alcanza la mejor aptitud posible de 1. Sin embargo, se requirieron 3 generaciones para llegar a este punto. Es posible que el algoritmo haya tenido que explorar varias soluciones antes de converger a la solución óptima.

Respecto al desempeño del algoritmo 2. Se tomó un total de 2 generaciones en alcanzar el criterio de paro por épsilon, presentando una aptitud promedio final de 0,8434 y teniendo a su mejor individuo con aptitud de 1,0000. Este proceso de evolución se puede observar en la Figura 7.

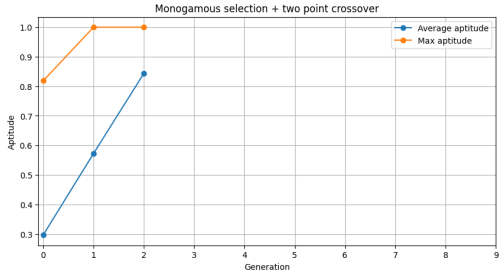

Figura 7: Evolución de la aptitud de los individuos del algoritmo 2.

En este caso, también presenta una alta mejor aptitud de 1, pero en comparación con el Algoritmo 1, requiere solo 2 generaciones. Aunque el valor promedio de aptitud es menor que el del Algoritmo 1, la rápida convergencia en solo 2 generaciones es una característica destacable.

En cuanto al algoritmo 3. Este tardó un total de 5 generaciones en alcanzar el paro por épsilon, logrando obtener una aptitud promedio final de 0,8219, y teniendo a su mejor individuo con una aptitud de 1,0000. Para este algoritmo podemos observar en la Figura 8 su proceso de evolución.

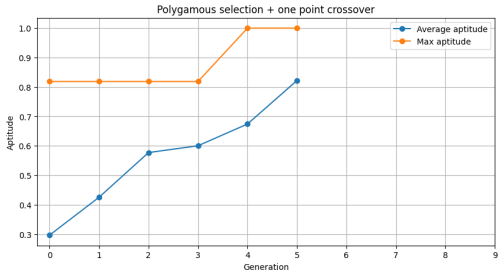

Figura 8: Evolución de la aptitud de los individuos del algoritmo 3.

Dicho algoritmo tiene un mayor número de generaciones en comparación con los dos algoritmos anteriores. Aunque la aptitud promedio y la mejor aptitud siguen siendo altas, el mayor número de generaciones podría indicar que este algoritmo requiere más tiempo para converger hacia soluciones óptimas.

Por último, relacionado al algoritmo 4. Se tomo un total de 2 generaciones en alcanzar el paro por épsilon, donde se obtuvo una aptitud promedio final de 0,8185, y un mejor individuo con un valor igual de aptitud. El proceso de evolución de este algoritmo se puede observa en la Figura 9.

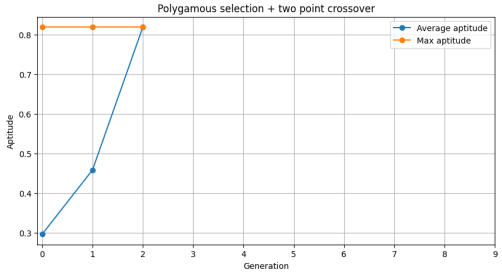

Figura 9: Evolución de la aptitud de los individuos del algoritmo 4.

Es importante notar que la mejor aptitud de este algoritmo no alcanza el valor máximo posible de 1. Aunque este algoritmo requiere solo 2 generaciones, su incapacidad para alcanzar la mejor aptitud posible podría indicar que está quedando atrapado en un óptimo local o necesita ajustes en su configuración.

6. Conclusiones

En términos generales, cada uno de los algoritmos exhibe resultados alentadores en relación tanto a la aptitud promedio como a la mejor aptitud obtenida. No obstante, la elección del algoritmo óptimo está totalmente ligado a las prioridades del problema en cuestión. Si la velocidad de convergencia es una consideración primordial, el Algoritmo 2 podría destacarse, dado su logro de una excelente mejor aptitud en tan solo dos generaciones. En caso de que tanto la aptitud promedio como el tiempo necesario para converger sean factores determinantes, valdría la pena contemplar la viabilidad de los Algoritmos 1 o 3. Por otro lado, el Algoritmo 4, a pesar de su rápida convergencia, no alcanza la deseada mejor aptitud, sugiriendo así la posibilidad de ajustar su diseño.

Por último, es relevante mencionar que si bien todos los algoritmos han logrado la convergencia esperada sin mostrar signos de estagnación, la elección adecuada de los criterios de paro resulta crucial. En el caso específico de esta práctica, la elección de un épsilon más elevado con seguridad hubiera generado una mejora en la aptitud de todos los algoritmos.

Referencias bibliográficas

[1] E. Wirsansky, *Hands-On Genetic Algorithms with Python*, 1st ed. Packt Publishing, 2020, ISBN: 1838557741.

[2] I. Gridin, *Learning Genetic Algorithms with Python*, 1st ed. BPB Publications, 2021, ISBN: 8194837758.