Universidad Nacional Autónoma de México

Facultad de Ciencias

Cómputo Evolutivo - 7140

T A R E A 4

Equipo:

Castillo Hernández Antonio - 320017438

☎☎☎☎

Luna Campos Emiliano - 320292084

☎☎☎☎

Juárez Cruz Joshua - 320124516

☎☎☎☎

Fecha de entrega:
24 de octubre del 2025

Profesor:
Oscar Hernández Constantino

Ayudantes:
Rodrigo Fernando Velázquez Cruz
María Adelina Torres Ochoa

Ejercicio 1. Algoritmo Genético - Optimización continua

1.a) Descripción e implementación

Describe e implementa un algoritmo genético para las funciones anteriores de optimización continua.
Se deben implementar y utilizar los siguientes componentes:

i) **Representación binaria para las soluciones.** Se ha creado un método que decodifica un arreglo de números binarios. Nos centramos en la decodificación, ya que la población inicial arranca ya codificada para agilizar la transición de algún numero a binario.

```
def generar_poblacion_inicial(NIND, dim_x, n_bits):
    longitud_individuo = dim_x * n_bits
    poblacion = np.random.randint(0, 2, size=(NIND, longitud_individuo))
    return poblacion
```

Para la decodificación, usamos dos métodos, uno que recorre el arreglo, y otro que va decodificando los números con una linealización.

```
def decodifica_array(x_cod, dim_x, n_bits, a, b):
    res = [0.0] * dim_x
    for i in range(dim_x):
        bits = [0] * n_bits
        for j in range(n_bits):
            bits[j] = x_cod[i * n_bits + j]
        res[i] = decodifica(bits, n_bits, a, b)
    return res

def decodifica(x_cod, n_bits, a, b):
    k = 0
    for i in range(n_bits):
        k = (k << 1) | x_cod[i]

    max_val = (1 << n_bits) - 1
    return a + k * (b - a) / max_val
```

ii) **Selección de padres por el método de la ruleta.**

Justifica y utiliza un método de transformación de la función de aptitud que se pueda aplicar a problemas de minimización (como las funciones de prueba de esta tarea).

Como bien se ha observado, el método de ruleta por si solo funciona para problemas de maximización. En este caso hemos alterado el método original, simplemente calculando el máximo de la muestra, y a este restarle el fitness individual de cada individuo, de esta forma el numero mayor sera el del fitness mínimo.

Después se ocupa la ruleta rutinariamente, se eligen la cantidad de padres que se desea para cada generación.

```
def seleccion_ruleta(poblacion, fitness_array, num_padres):
    inv_fitness = np.max(fitness_array) - fitness_array + 1
    suma_fitness = np.sum(inv_fitness)
    probas = inv_fitness / suma_fitness
    indices = np.random.choice(len(poblacion), size=num_padres, p=probas)
    return poblacion[indices]
```

iii) **Operador de cruza uniforme.**

Considera una probabilidad de cruza; en caso de que los padres no se crucen se deberán generar clones.

En el operador de cruza, se toman dos padres, y los hijos decidirán su i-ésimo bit a través de un "volado".

```
def cruza_uniforme(padre1, padre2, prob_cruza=0.8):
    longitud = len(padre1)

    r = random.random()
    if r < prob_cruza:
        hijo1 = np.zeros(longitud, dtype=int)
        hijo2 = np.zeros(longitud, dtype=int)

        for i in range(longitud):
            bit = random.randint(0, 1)

            if bit == 0:
                hijo1[i] = padre1[i]
                hijo2[i] = padre2[i]
            else:
                hijo1[i] = padre2[i]
                hijo2[i] = padre1[i]
    else:
        hijo1 = padre1.copy()
        hijo2 = padre2.copy()

    return hijo1, hijo2
```

iv) **Mutación flip.**

Utiliza una probabilidad de mutación por cada bit. Todos los bits, de todos los individuos, deberían considerarse para la posibilidad de mutar.

Se sigue la implementación estándar. La muestra es recorrida y por cada bit de esta, con cierta probabilidad, se decide si cambiar su bit o no.

```
def mutar_flip(individuo, prob_mutacion=0.01):
    for i in range(len(individuo)):
        if random.random() < prob_mutacion:
            individuo[i] = 1 - individuo[i]
    return individuo
```

v) **Reemplazo de los peores.**

Considera incluir un parámetro para indicar qué porcentaje (o número de individuos) de los peores deben ser reemplazados. Se debe garantizar que la mejor solución siempre permanece en la población.

En el reemplazo se ordenan los mejores fitness de un porcentaje elegido de la población y se eliminan los mas débiles. La población restante se une con los hijos y esta es una nueva generación.

```
def generar_nueva_poblacion(poblacion, fitness, porcNewInd, porcMutacion,
        funSeleccion,funCruza, probCruza=0.8):
    NIND = len(poblacion)
    n_new = int(porcNewInd * NIND)
    elite_indices = np.argsort(fitness)[:NIND - n_new]
    elite = poblacion[elite_indices]
    .
    .
    .
    hijos = hijos[:n_new]
    nueva_poblacion = np.vstack((elite, hijos))
```

Ejercicio 2. Experimentación

2.a) Ejecución del algoritmo

Ejecutar el algoritmo genético al menos 10 veces para cada función de prueba (en dimensión 10).
Utiliza un criterio de paro que te permita realizar una comparación justa con la búsqueda aleatoria.

Esto se hace mediante el archivo *experimentacion.py* el cual corre 10 pruebas de cada función tanto para algoritmo genético como para búsqueda aleatoria, es decir, hace 100 ejecuciones en total. Hemos tomado como criterio de paro un numero de generaciones, que hemos tomado como 100.

Para todos ocupamos un total de 100 generaciones porque dado que usábamos poblaciones de 100, estas tendían a establecerse en las 60 o 70, por lo que hacer uso de mas generaciones era un gasto innecesario.
Hicimos uso de los siguientes parámetros para la versión original, es decir:

- Selección de padres por el método de la ruleta: Aquí utilizamos el tamaño de la población, para obtener la cantidad de padres. Decidimos una población de tamaño 100 para no sobrecargar nuestras arquitecturas, ademas para unos problemas de funciones nos pareció una cantidad razonable para no alargar la búsqueda. La cantidad de padres la elegimos de 0.8 del total de la población para que algunos puedan ser reemplazados por la próxima generación.

- operador de cruza uniforme: Unicamente con un parámetro ademas del número de padres, el cual es la probabilidad de cruza. Elegimos 0.8 para que hubiera gran cantidad de nuevos hijos para las próximas generaciones y no se quedara tan estancado en la misma población. Aunque no quisimos tampoco darle mucha probabilidad, para que pudiera quedar poblaciones anteriores que se puedan cruzar con las nuevas y no caer en una convergencia muy prematura.

- Mutación flip: Recibiendo ademas del individuo, un parámetro, le asignamos una probabilidad de 0.1 para que el individuo no cambiara completamente después del uso de este método, pero que tuviera la posibilidad de poder cambiar una o dos entradas, o inclusive de no cambiar. Básicamente para seguir la linea de población que se tiene y no dar un salto de población tan brusco (búsqueda aleatoria).

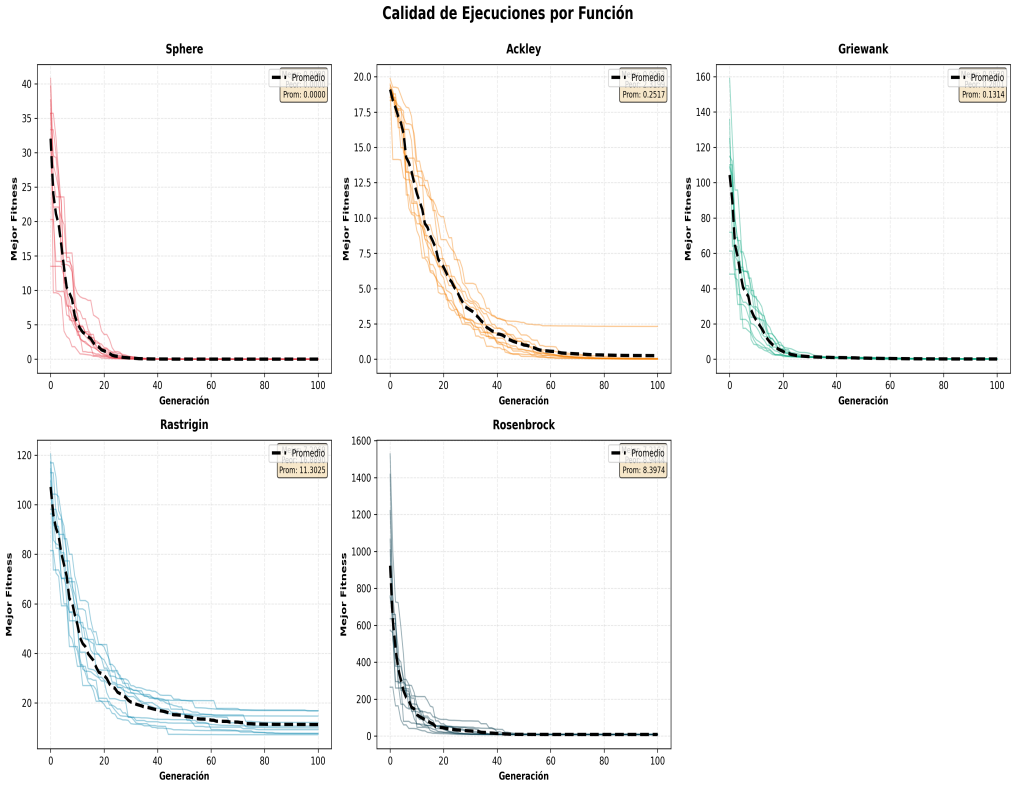
Estos son los resultados observados con el algoritmo genético con parámetros de selección por ruleta, cruza uniforme y reemplazo a través de un porcentaje contra los de búsqueda aleatoria.

Tabla 1: Resultados de desempeño de algoritmos de optimización

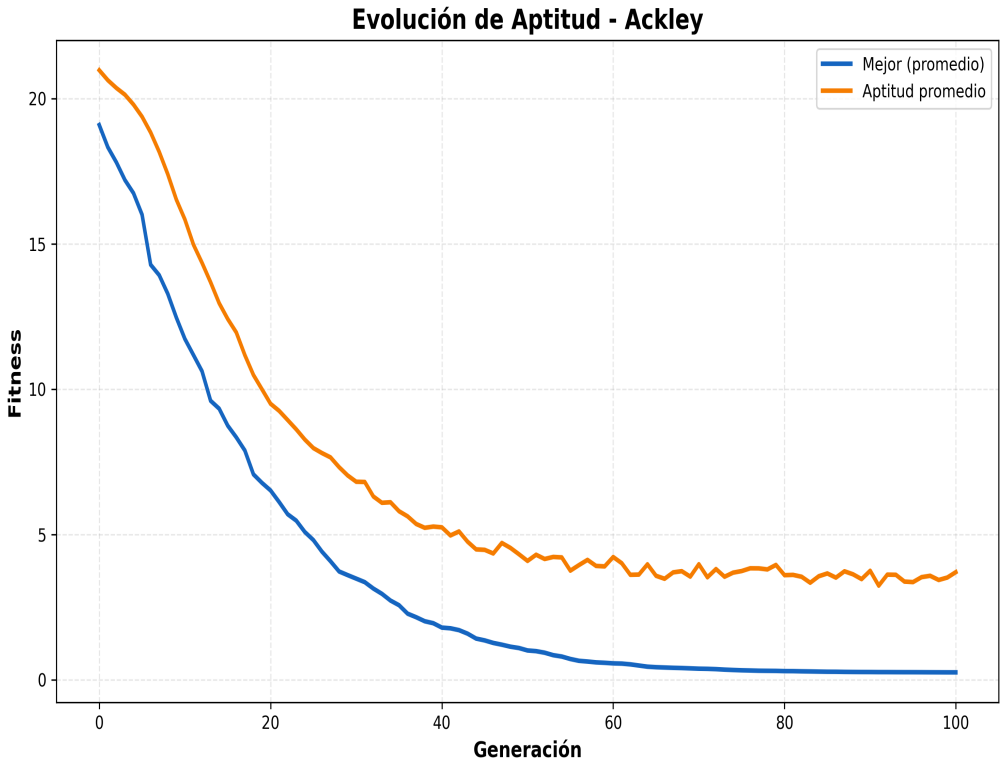
Algoritmo	**Función**	**Mejor**	**Peor**	**Promedio**	**Mediana**	**Desv. Est.**
AG	sphere	0.0000007935	0.0000099734	0.0000051344	0.0000054323	0.0000027521
AG	ackley	0.0053349333	2.3198812710	0.2516525263	0.0173199168	0.6895503562
AG	griewank	0.0520411378	0.2801471450	0.1314082094	0.0856998694	0.0839619238
AG	rastrigin	7.2068994481	16.8890315355	11.3025232220	10.3049652545	3.5064230596
AG	rosenbrock	7.2106664964	8.9443849255	8.3973611991	8.7294477664	0.5882405485
BAleatoria	sphere	11.0040297313	20.5416971532	14.5879880890	14.2527905937	2.6257210697
BAleatoria	ackley	13.1991115601	17.2538928947	15.6992996428	15.8849832559	1.2294160877
BAleatoria	griewank	33.6760566474	57.9539239374	44.2061724280	44.7919989735	7.1081312914
BAleatoria	rastrigin	62.8818304344	82.3198186211	73.0832134512	72.1739974073	4.9043020147
BAleatoria	rosenbrock	146.7033710161	225.7362270231	179.4036057419	167.8807627637	31.2947796906

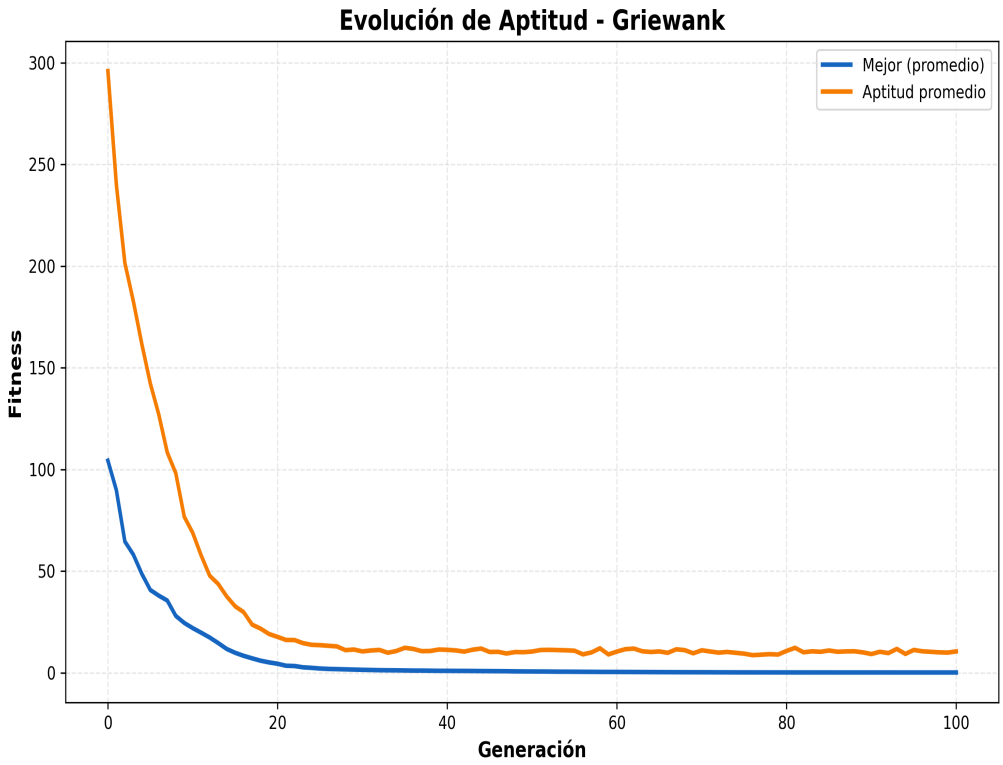
Observando la tabla para todas las funciones, el algoritmo genético mejora bastante las soluciones obtenidas a la búsqueda aleatoria, tanto así que incluso las peores de GA son mucho mejores que RS.

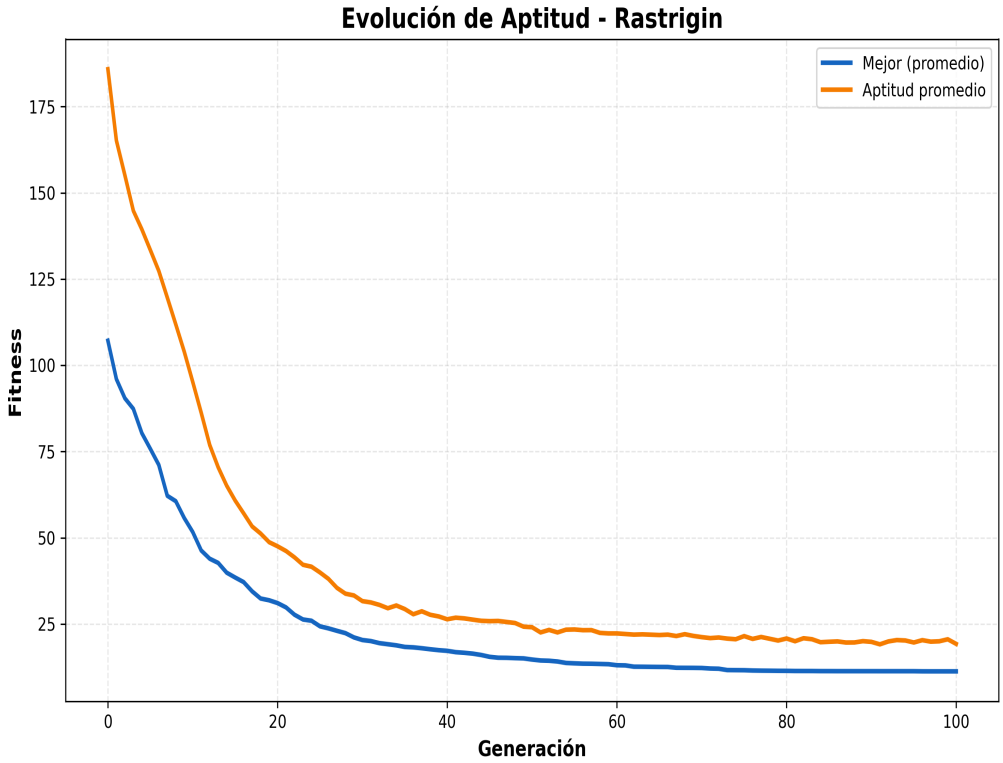
Igualmente a continuación se muestran las distintas ejecuciones y el promedio de GA.

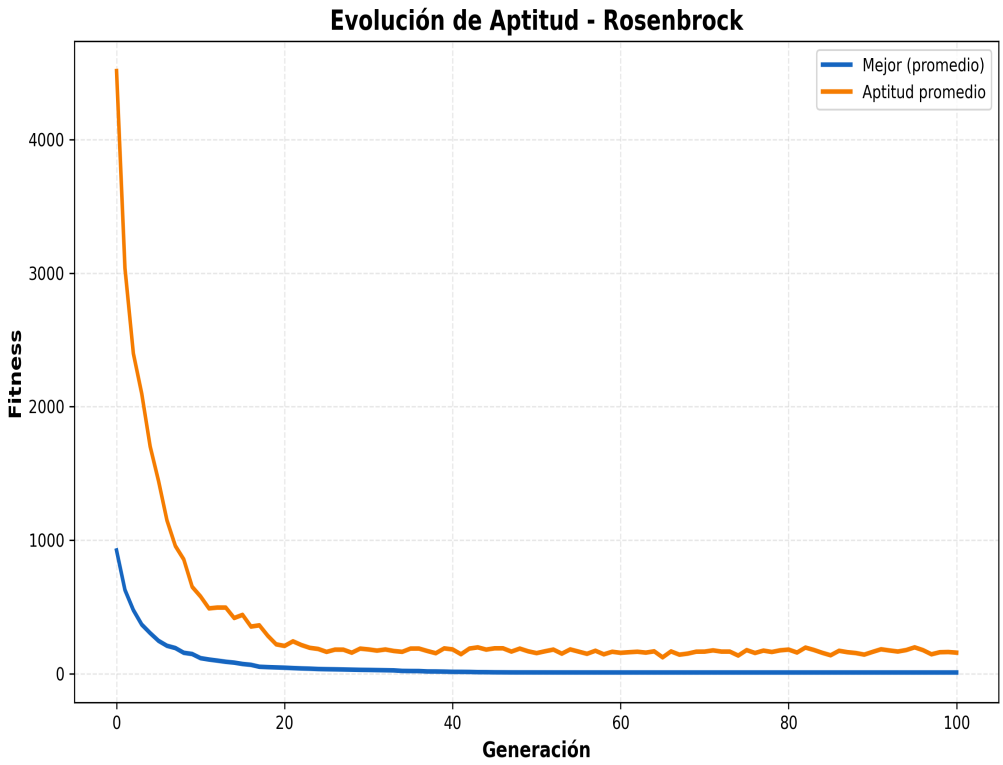


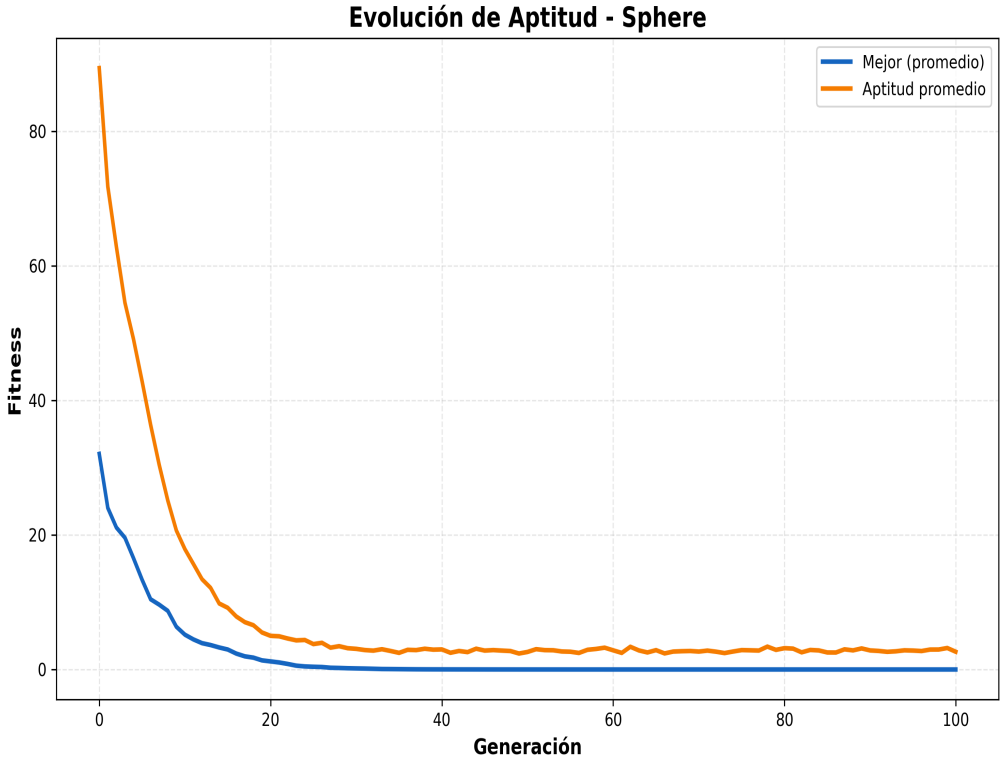
Y estas son las gráficas obtenidas del promedio de la población y la mejor obtenida para cada una de las funciones de acuerdo al algoritmo genético (como se plantea originalmente en el ejercicio 1). En general casi todas alcanzan una aparente convergencia para la generación 40.




Evolución de Aptitud - Griewank
Mejor (promedio)
Aptitud promedio
Fitness
Generación
300
250
200
150
100
50
0
0
20
40
60
80
100


Evolución de Aptitud - Rastrigin
Mejor (promedio)
Aptitud promedio
Fitness
Generación
175
150
125
100
75
50
25
0
20
40
60
80
100

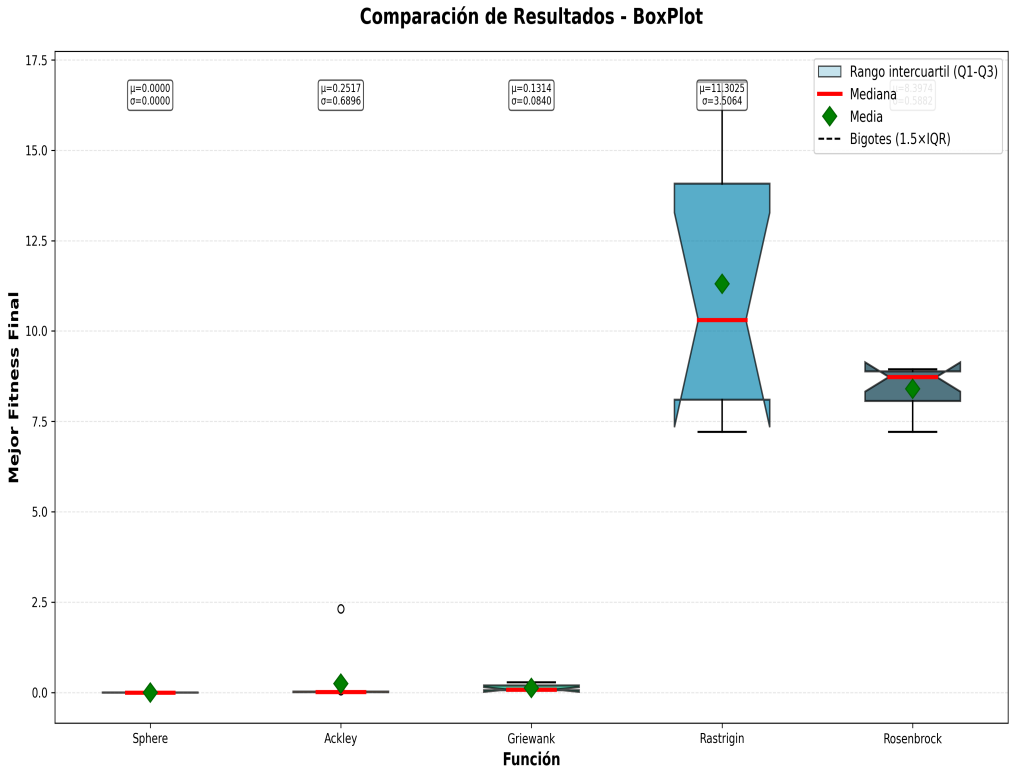




Esta por otra parte es la gráfica boxplot.

- Sphere: Como se puede observar todas las ejecuciones estuvieron muy cerca del 0, es por esto que tanto media como la mediana aparecen prácticamente en el mínimo (que es 0).

- Ackley: Es similar al anterior, tuvo casi todas las ejecuciones en valores muy cercanos al mínimo, sin embargo hubo una ejecución que se quedo en 2.5, por lo que la media se elevo un poco, pero realmente es un valor bastante factible aun así.
- Griewank: Este tampoco tuvo valores anormales, todos se mantuvieron muy cercanos al 0 (no tanto como los anteriores pero si muy cerca)
- Rastrigin: Esta fue la que mas trabajo dio y en la que peores resultados se obtuvieron, ya que la mayoría fue mayor a 10: Aunque principalmente limitándose a valores entre 10 y 14, hubo algunos que tuvieron como limite hasta 17. La segunda mayor cantidad fueron valore entre 8 y 10, los que estuvieron por debajo fueron los mas escasos de todos.
- Rosenbrock: Fue la segunda con peores resultados, aunque en esta los valores tendieron a ser menores, ya que la gran mayoría ocurrió entre 8 y 9, los que tuvieron menos de eso fueron muy pocos y los que tuvieron mas fueron aun más escasos.



2.b) Variantes por integrante

Por cada integrante del equipo, consideren implementar alguna de las siguientes versiones:

i) Cambiar método de selección de padres.

ii) Cambiar método de mutación.

iii) Cambiar operador de cruza.

iv) Cambiar método de reemplazo.

Por cada cambio, repite las diez ejecuciones para cada función de prueba.
El criterio de paro debería ser el mismo que en el ejercicio anterior, siempre considerando que se desea hacer una comparación justa (en recursos utilizados) con la búsqueda aleatoria.

En este caso, hemos hecho variantes para las selecciones, en donde incluimos la selección por torneo; para mutación hicimos mutación de un bit y para la cruza, agregamos cruza de un punto.

Variante 1) Seleccion por torneo

Para selección por torneo simplemente elegimos entre tres individuos al mejor con cierta probabilidad (en este caso siempre ganara el mejor porque lo asignamos como 1) .

Mas allá de la población, le pasamos el tamaño que queremos de nuevos individuos que como ya se explico con anterioridad, es el 0.8 de la población. A partir de aquí se hacen pequeños torneos que elegimos fueran entre tres individuos para que salieran los mejores, y ademas agregamos el parámetro de presión de convergencia igual a uno, que indica que siempre tomara al mejor en el torneo.

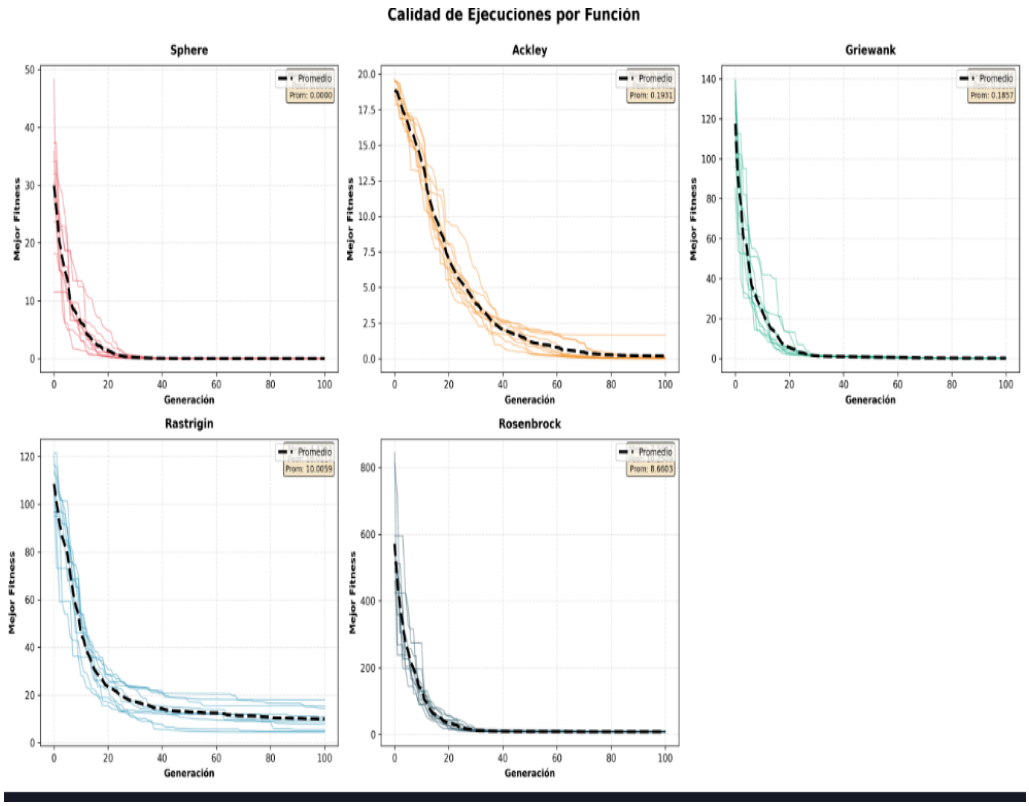
```
def seleccion_torneo(population, fitnesses, num_selections,
            tournament_size=3, pressure=1.0):
    selected = []
    for _ in range(num_selections):
        tournament_indices = random.sample(range(len(population)), tournament_size)
        # Minimización: menor fitness es mejor
        tournament_indices.sort(key=lambda i: fitnesses[i], reverse=False)
        if random.random() < pressure:
            winner = tournament_indices[0]
        else:
            winner = random.choice(tournament_indices[1:] if len(tournament_indices) >
            1 else tournament_indices)
        selected.append(population[winner])
    return selected
```

Estos son los resultados obtenidos con la variante de selección por torneo.

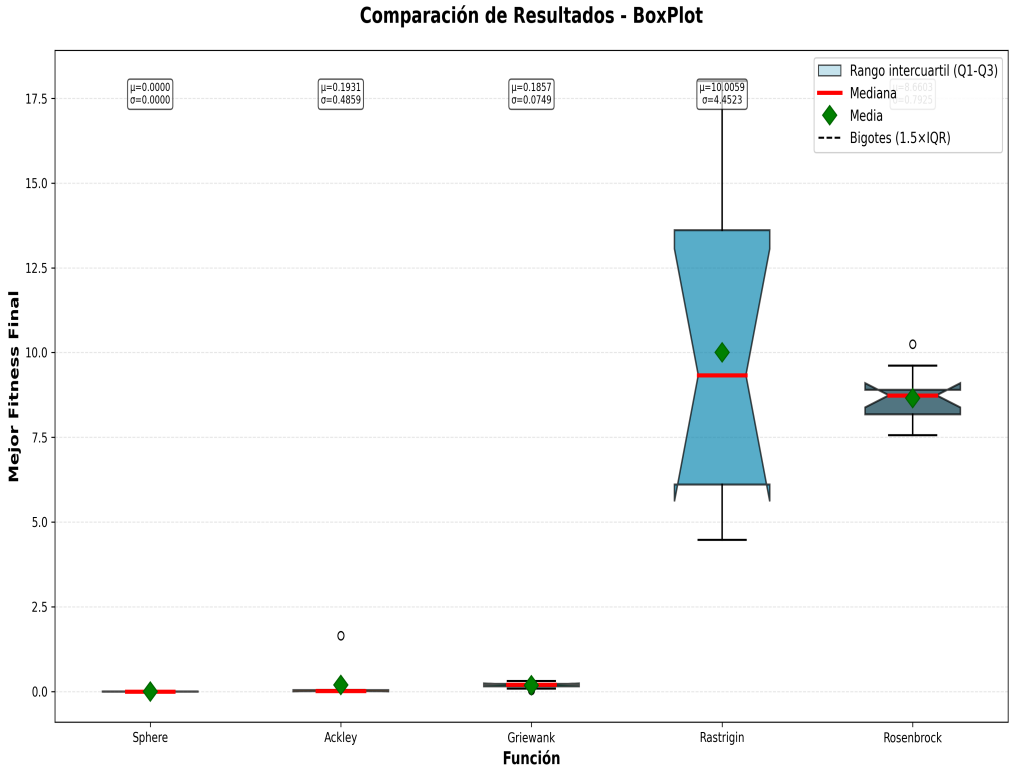
Tabla 2: Resultados con la variante de selección por torneo

Algoritmo	Función	Mejor	Peor	Promedio	Mediana	Desv. Est.
AG	sphere	0.0000006958	0.0000255501	0.0000073855	0.0000052614	0.0000071082
AG	ackley	0.0084551111	1.6481517701	0.1931301890	0.0201640564	0.4858959668
AG	griewank	0.0374483500	0.3154546639	0.1856724684	0.2040721772	0.0749484249
AG	rastrigin	4.4753163553	18.0116865721	10.0059127722	9.3337138682	4.4522811971
AG	rosenbrock	7.5666963680	10.2509470692	8.6603465232	8.7392354496	0.7924944783
BusquedaAleatoria	sphere	6.5993964119	15.0230923834	12.5447492485	13.1083520318	2.5094899713
BusquedaAleatoria	ackley	14.9665126337	17.5810794452	16.4902064612	16.6460888512	0.8460079212
BusquedaAleatoria	griewank	21.4645907397	60.7664636329	40.0423940608	39.5318921623	12.4648441241
BusquedaAleatoria	rastrigin	66.3054094333	77.0576016742	71.8415753421	72.8710752350	3.5692556594
BusquedaAleatoria	rosenbrock	108.7122263529	281.4296052041	200.2158969623	198.1909143027	46.4381948721

En general, con este tipo de selección se converge mas rápido porque se eligen unicamente los mejores, así que creemos hay menos probabilidad de cruza con resultados diferentes.



Sin embargo, es curioso que con en este también haya habido una ejecución particular para ackley que llega casi al 2, ademas que Rastrigin llego a tener mejores resultados comparado con ruleta.



Variante 2) Mutacion un bit

Para mutación simplemente tomamos un bit random y lo cambiamos. Asi que realmente no hay argumentos que pasar aqui

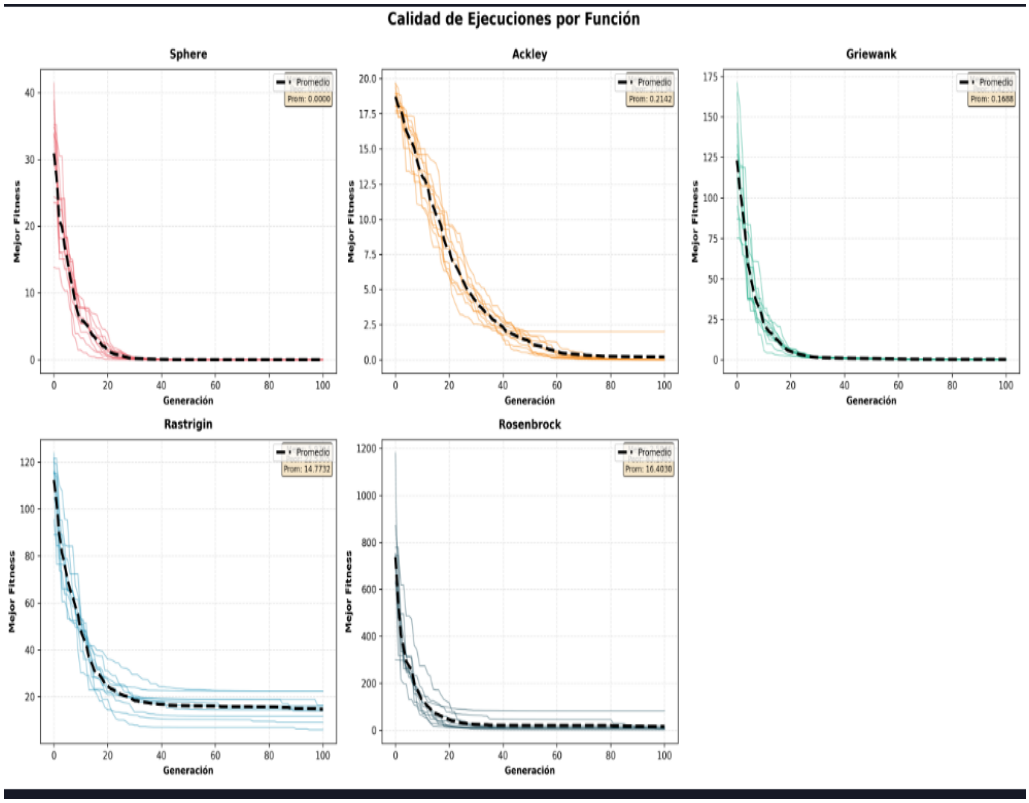
```
def mutar_un_bit(individuo):
    i = random.randrange(len(individuo))
    individuo[i] = 1 - individuo[i]
    return individuo
```

Dando estos resultados. Es de recalcar que para las los mas complicadas (rastrigin y rosenbrock) se tuvieron peores reultados.

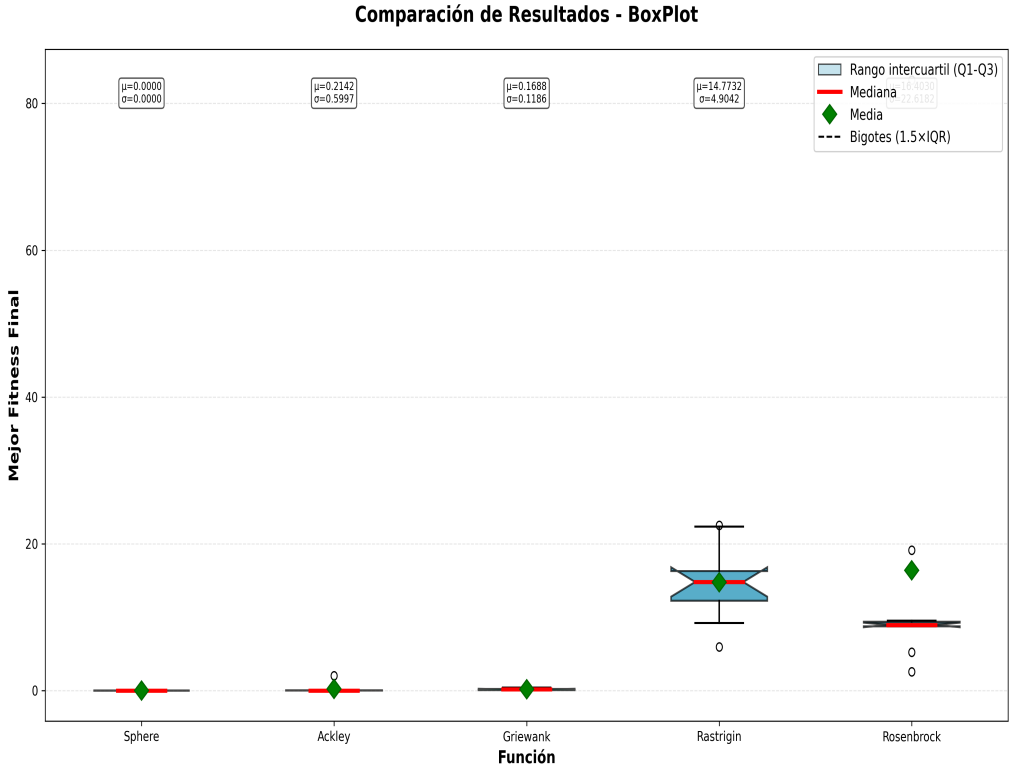
Tabla 3: Resultados de variante mutacion de un bit

Algoritmo	Función	Mejor	Peor	Promedio	Mediana	Desv. Est.
AG	sphere	0.0000004517	0.0000083621	0.0000042213	0.0000035768	0.0000028470
AG	ackley	0.0100384082	2.0133543018	0.2141837213	0.0123646736	0.5997366588
AG	griewank	0.0284320014	0.4252783044	0.1687952462	0.1658811309	0.1185811815
AG	rastrigin	5.9763787411	22.5305267957	14.7731816840	14.7962623637	4.9041788134
AG	rosenbrock	2.5845577964	83.1984813907	16.4029739035	8.9459903780	22.6182357493
BusquedaAleatoria	sphere	7.0453140467	14.5621983442	12.3671017561	12.9069944949	2.1453564331
BusquedaAleatoria	ackley	15.8333230003	17.4845607632	16.7793089500	16.9715334729	0.5320284738
BusquedaAleatoria	griewank	31.1052068358	61.1950317289	44.9234779733	43.2407126227	7.7902086597
BusquedaAleatoria	rastrigin	57.5521988835	77.7692209211	69.1943359353	71.9655410167	7.5290212440
BusquedaAleatoria	rosenbrock	111.4928829607	311.2169127327	208.8000622878	231.1807514579	76.3299647622

En esta, se puede observar que aunque se tienda a un resultado, los fitness varían mas entre si,.



Tambien se nota que cuando hay mutacion de un bit, tambien hay mas casos especiales, ahora ademas de ackley, rastrigin y rosenbrock tuvieron dos y tres casos especiales respectivamente. Añadiendo, estas dos tuvieron comportaminetos distintos, ratrigintuvo mayor concentracion de resultados peores; y rosenbrock estuvo muy concentrado en valores de 10 aproximadamente.



Variante 3) Cruza de un punto

Finalmente para la cruza de un punto se elige un i-esimo bit random, y se generan dos hijos, uno tendrá hasta el i-esimo bit parte del padre uno y el resto del padre dos; y el hijo dos lo mismo pero inversamente. En este caso tampoco hay parametros que podamos variar

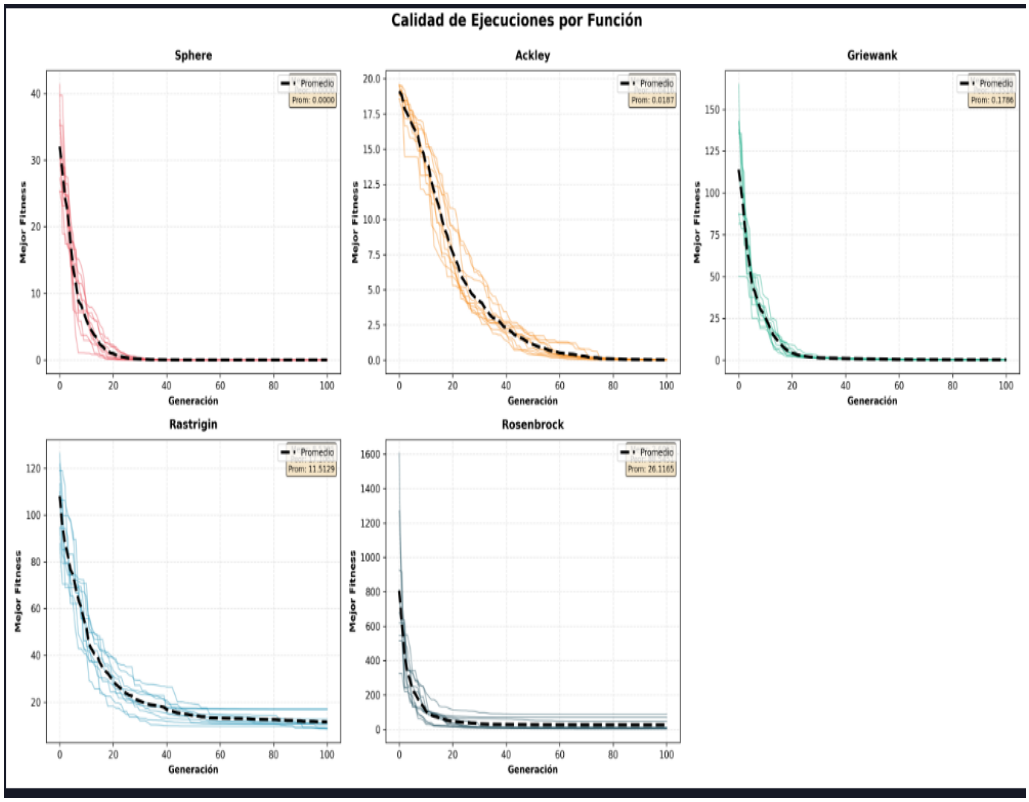
```
def cruza_un_punto(padre1, padre2):
    punto = random.randint(1, len(padre1) - 1)
    hijo1 = np.concatenate([padre1[:punto], padre2[punto:]])
    hijo2 = np.concatenate([padre2[:punto], padre1[punto:]])

    return hijo1, hijo2
```

Estos son los resultados obtenidos. Aunque tiene valores peores que la versión anterior, no son resultados que estén muy alejados, son de aproximadamente tres unidades de diferencia como mucho, salvaguardado el caso de rosenbrock que ahora si se vio fuertemente afectada y que se mostrara con mas detenimiento adelante.

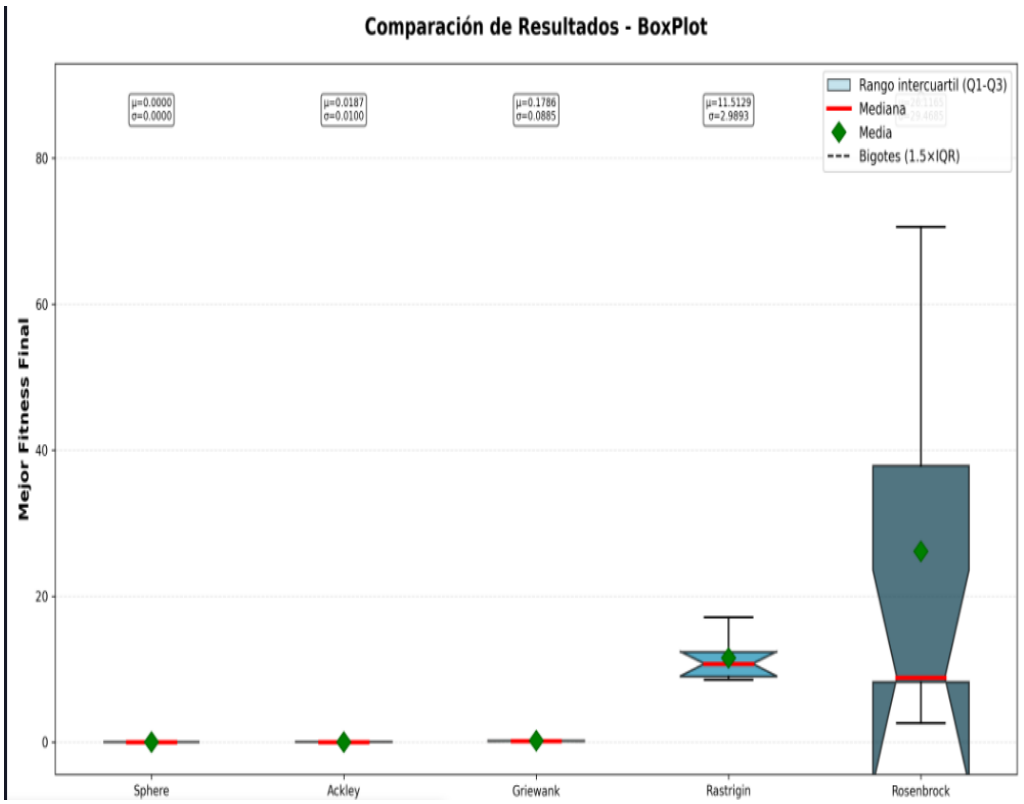
Tabla 4: Resultados de desempeño de algoritmos de optimización

Algoritmo	Función	Mejor	Peor	Promedio	Mediana	Desv. Est.
AG	sphere	0.0000008911	0.0000135868	0.0000043776	0.0000032594	0.0000038669
AG	ackley	0.0067471352	0.0415887503	0.0186967536	0.0156254345	0.0099593797
AG	griewank	0.0694523470	0.3513631375	0.1786453429	0.1552359672	0.0885038803
AG	rastrigin	8.5304762603	17.1062733285	11.5129287901	10.7262640626	2.9893219492
AG	rosenbrock	2.6053113627	88.5451058102	26.1164908176	8.8441405322	29.4685435606
BusquedaAleatoria	sphere	4.0463862398	14.4471094266	10.5916741096	10.4430019974	3.1941906365
BusquedaAleatoria	ackley	14.8679758789	16.9581010376	16.0234466302	16.2717435789	0.6492673847
BusquedaAleatoria	griewank	18.7544213082	48.8498103581	38.3081754433	38.6525676645	8.1825383800
BusquedaAleatoria	rastrigin	57.0133471409	76.1246018042	69.3080260800	72.1595291981	5.6806496541
BusquedaAleatoria	rosenbrock	113.3501273910	321.3735807172	217.0861156451	223.7868587087	56.8336053260

Enfatizar nuevamente dos puntos, la tan pronta convergencia que se tiene y que aunque tiendan al mismo resultado, los fitness son procesados de manera distinta



En lienas generales casi todas las funciones se comportaron igual que en las variantes anterirores. Ahora es cuando entra rosenbrock la cual tuvo valores peores, concentrandose entre 10 y 40, y con una gran cantidad que llegaron hasta casi los 80.



2.f) Comentarios sobre los resultados

Lo primero y mas evidente a notar es la gran diferencia que se tiene entre la búsqueda aleatoria y el algoritmo genético, no importa la variante de este ultimo, sus resultados son mucho mejores que los encontrados buscando aleatoriamente. Lo siguiente a notar es la similitud de los resultados a pesar de la varianza de métodos, todas tienden hacia el mismo conjunto de resultados óptimo, si no es que incluso llegan al optimo global; es cierto que hay algunos que pueden tener ciertas descomposturas como fue el caso de rosebrock al final, sin embargo, llegó a tener un resultado igualmente muy aceptable como mejor resultado. Dadas las distintas funciones, es importante observar a su vez que en funciones muy sencillas es casi seguro obtener el óptimo global y que en funciones más complejas con al menos una variante se puede obtener un resultado más que aceptable.