udp Escuela de Informática y Telecomunicaciones

Facultad de Ingeniería y Ciencias
Escuela de Informática y Telecomunicaciones

Algoritmos Exactos y Metaheurísticas
Tarea 03

Jazmín Almendra Jassive Cuitiño Mendoza
Felipe Nicolás Gutierrez Lazo
jazmin.cuitino@mail.udp.cl
felipe.gutierrez_l@mail.udp.com

30 de junio de 2024

Índice general

Capítulo 1

Introducción

En esta actividad, se desarrolla e implementa una metaheurística de población, con el fin de demostrar que, en funciones multimodales, tienen un mejor rendimiento que algoritmos como el gradiente descendiente. Este tipo de funciones *(multimodales)* poseen numerosos óptimos locales en los que frecuentemente muchos algoritmos de optimización quedan atrapados. Esta metaheurística de población a implementar se pone a prueba con tres de estas funciones, buscando minimizar el valor objetivo. Se realizan gráficos de convergencia que revelan el mejor valor en el mejor tiempo de procesamiento, para finalmente realizar un análisis comparativo con otro tipo de técnicas.

Capítulo 2

Descripción Actividades

Las funciones multimodales son un tipo de funciones dentro del área de optimización que cuentan con numerosos extremos locales, atrapando con óptimos locales a muchos algoritmos de optimización. Un ejemplo de esto es el gradiente descendiente.

Las actividades que contempla este trabajo son las siguientes:

1. Diseñar e implementar una metaheurística de población, eligiendo entre Particle Swarm Optimization (***PSO***), Ant Colony Optimization (***ACO***) y Genetic Algorithm (***GA***).
2. Realizar 10 ejecuciones para cada función con la metaheurística implementada. En este caso, se elige el algoritmo genético. Se muestran los resultados obtenidos. Con estos resultados se generan gráficos de convergencia, para poder analizar y entregar una justificación de la elección de los parámetros del algoritmo.
3. Finalmente, realizar una conclusión en base a los análisis realizados con los resultados obtenidos

Capítulo 3

Desarrollo actividades

3.1. Justificación de la metaheurística elegida

Tanto **PSO** como **ACO** son algoritmos muy poderosos para ciertos tipos de problemas de optimización, sin embargo, los **Algoritmos Genéticos** son generalmente los preferidos para problemas multimodales, esto debido a su capacidad para manejar múltiples soluciones óptimas, no quedándose atrapado en óptimos locales gracias a su exploración global del espacio de búsqueda.

Además de lo mencionado, **GA** trabaja con menos parámetros configurables que PSO y ACO, de modo que su ejecución es mas rápida y mas fácil de testear modificando los parámetros necesarios hasta encontrar la mejor configuración de los mismos.

De este modo, la metaheurística escogida para resolver este problema multimodal es **Genetic Algorithm**.

3.2. Algoritmo Diseñado

Lo primero que se realiza antes de diseñar el algoritmo es la definición de las funciones de optimización en el código. A continuación, se adjuntan los códigos de las tres funciones.

Listing 3.1: Definición de las funciones de optimización

```
    def f1(x):
    return sum(xi**2 - 10 * np.cos(2 * np.pi * xi) + 10 for xi in x)

    def f2(x):
        return sum(100 * (x[i+1] - x[i]**2)**2 + (x[i] - 1)**2 for i in
           range(len(x)-1))

    def f3(x):
        return -sum(np.sin(xi) * (np.sin(i * xi**2 / np.pi))**20 for i, xi
            in enumerate(x, 1))
```

El algoritmo se descompone en las siguientes partes fundamentales:

1. **Inicialización de la población**: El primer paso de un algoritmo genético es generar la población inicial:

Listing 3.2: Inicializacion de la población

```
    def initialize_population(pop_size, dim, bounds):
    return np.random.rand(pop_size, dim) * (bounds[1] - bounds[0]) +
       bounds[0]
```

La definición de las variables se definen a continuación:

- **pop_size**: Tamaño de la población.
- **dim**: Dimensión de cada solución. Número de características o variables de cada individuo.
- **bounds**: Valores que indican límites inferiores y superiores.

2. **Evaluación de la población**: Una vez generada la población inicial, se debe evaluar el fitness aplicando la función objetivo. Esto transforma cada solución en un vector numérico que representa su calidad:

Listing 3.3: Evaluacion de la población

```
def evaluate_population(population, func):
return np.array([func(ind) for ind in population])
```

La definición de variables es la siguiente:

- **population:** Población de soluciones a evaluar.
- **func:** Función de evaluación que se aplicará a cada individuo de la población.

3. **Selección de los individuos para la reproducción:** Selecciona individuos para la próxima generación usando un torneo. Se eligen **k** individuos al azar y se selecciona el mejor de ellos *(el de menor valor de fitness)*. Este proceso se repite para llenar la nueva población.

Listing 3.4: Selección de la población para la reproducción

```
def tournament_selection(population, fitness, k=3):
selected = []
pop_size = len(population)
for _ in range(pop_size):
    aspirants = np.random.choice(pop_size, k, replace=False)
    best = aspirants[np.argmin(fitness[aspirants])]
    selected.append(population[best])
return np.array(selected)
```

Las variables se definen a continuación:

- **population**: Representa la población actual mediante un array bidimensional, donde cada fila es un individuo.
- **fitness**: Array 1D que contiene el valor fitness de cada individuo en la población.
- **k**: Tamaño de los individuos que competirán en cada torneo. En este caso, son 3 individuos.

4. **Nueva Generación**: El siguiente paso en el algoritmo genético es generar una nueva generación, para esto se realizan dos procesos:

- **Crossover o cruzamiento:** Esta técnica se encarga de combinar dos individuos (padres) para producir un nuevo individuo. En vista de este problema, se elige un punto de cruce al azar, y las partes de los padres se combinan, esto permite la recombinación de genes.

Listing 3.5: Crossover

```
def crossover(parent1, parent2, crossover_rate):
if np.random.rand() < crossover_rate:
    point = np.random.randint(1, len(parent1))
    return np.concatenate((parent1[:point], parent2[point
        :]))
return parent1
```

- **Mutación:** La mutación introduce una variación aleatoria en los individuos. Cada gen en un individuo tiene una probabilidad de ser alterado aleatoriamente dentro del dominio de las variables. En el código, esta probabilidad esta denotada por la variable **"mutation_rate"**.

Listing 3.6: Mutación

```
    def mutate(individual, mutation_rate, bounds):
    for i in range(len(individual)):
        if np.random.rand() < mutation_rate:
            individual[i] = np.random.uniform(bounds[0], bounds[1])
    return individual
```

La variable **mutation_rate** corresponde a un parámetro ajustable del algoritmo. En la sección ***Análisis***, se ve demostrada la influencia de esta.

5. **Evolución de la población:** Ya definidas las partes anteriores, se aplican estas funciones (selección, cruce y mutación) para generar una nueva población.

Listing 3.7: Mutación

```
def evolve_population(population, fitness, mutation_rate,
    crossover_rate, bounds):
    new_population = []
    selected_population = tournament_selection(population, fitness)
    for i in range(0, len(population), 2):
        parent1 = selected_population[i]
        parent2 = selected_population[i + 1]
        child1 = crossover(parent1, parent2, crossover_rate)
        child2 = crossover(parent2, parent1, crossover_rate)
        new_population.append(mutate(child1, mutation_rate, bounds))
        new_population.append(mutate(child2, mutation_rate, bounds))
    return np.array(new_population)
```

Cabe destacar que en esta función es donde se producen o crean los dos hijos provenientes del crossover entre los padres.

Luego de generar las funciones principales para el algoritmo genético, se crea la función *run_ga*, que coordina el proceso evolutivo. Esta función junta todas las funciones anteriores y guarda la mejor solución y su fitness. A continuación, se adjunta el código de la función en cuestión.

Listing 3.8: Función *run_ga* que ejecuta el algoritmo genético.

```
    def run_ga(func, dim, bounds, pop_size, max_generations, mutation_rate
        , crossover_rate):
    population = initialize_population(pop_size, dim, bounds)
    fitness = evaluate_population(population, func)
    best_fitness = np.min(fitness)
    best_index = np.argmin(fitness)
    best_solution = population[best_index]
    convergence = [(0, best_fitness, 0)]
    start_time = time.time()
    for generation in range(1, max_generations + 1):
        population = evolve_population(population, fitness, mutation_rate,
            crossover_rate, bounds)
        fitness = evaluate_population(population, func)
        current_best_fitness = np.min(fitness)
        if current_best_fitness < best_fitness:
```

```
            best_fitness = current_best_fitness
            best_index = np.argmin(fitness)
            best_solution = population[best_index]
        elapsed_time = time.time() - start_time
        convergence.append((elapsed_time, best_fitness, generation))
    return best_solution, best_fitness, generation, convergence
```

Finalmente, se crea la función *execute_ga_multiple_times()*, que es la que se encarga de realizar las 10 ejecuciones por cada función de optimización. Se adjunta el código a continuación.

Listing 3.9: Función *execute_ga_multiple_times()*.

```
def execute_ga_multiple_times(func, dim, bounds, pop_size,
    max_generations, num_executions=10):
results = []
convergence_data = []
for _ in range(num_executions):
    best_solution, best_fitness, generation, convergence = run_ga(func
        , dim, bounds, pop_size, max_generations, mutation_rate,
        crossover_rate)
    cpu_time = convergence[-1][0]  # Ultimo tiempo de CPU registrado
    results.append((best_solution, best_fitness, generation, cpu_time,
         convergence))
    convergence_data.append(convergence)
return results, convergence_data
```

3.3. Consideraciones importantes

Además del código expuesto previamente, cabe destacar el reconocimiento y aplicación de los siguientes conceptos:

1. **Operador de cruzamiento:** Se define como cruzamiento en un punto. El punto de cruzamiento es escogido al azar, concatenando una parte de cada padre para dar origen a 2 hijos. Estos son hijos son generados en la función ***evolve_population()***. A modo de ejemplo, se adjunta la demostración de las funciones *crossover()* y *evolve_population()*, respectivamente:

   - Suponga que se tienen los siguientes padres, con un *crossover_rate* de 0.8.
     - parent1 = [1, 2, 3, 4, 5]
     - parent1 = [1, 2, 3, 4, 5]

     Si la probabilidad de cruce resulta positiva para el crossover_rate (es decir, *crossover_rate* > *np.random.rand*()) se procede con el cruce. Hay que considerar ahora el valor de retorno que entrega la función *np.random.randint(1, 5)*. A modo ilustrativo, se establece un valor igual a 3, es decir, el punto donde se realiza el cruce es después del tercer elemento de cada individuo. Así, el hijo resulta de la concatenación de los dos padres en ese punto. En el caso ilustrativo, el hijo resulta ser **[1, 2, 3, 2, 1]**. El proceso recién descrito corresponde al realizado por la función **crossover()**. La generación de los dos hijos es gracias a la función **evolve_population()**.

2. **Operador de mutación:** El operador de mutación es definido como **mutación uniforme**, pues cada uno de los genes del individuo se puede alterar según una cierta probabilidad aleatoria llamada *"mutation_rate"*. Se ilustra con el siguiente ejemplo:

   - Suponga que tiene un individuo y los siguientes parámetros:
     a) **individual** = [1.0, 2.0, 3.0, 4.0, 5.0]
     b) **mutation_rate** = 0.1

c) **bounds** = [0.0, 10.0]

- **Iteraciones en el Proceso de Mutación**

  a) **Para el gen 0 - Individual[0] = 1.0**

  - Se genera un número aleatorio, digamos 0.05. Como $0.05 < 0.1$, se muta.
  - Nuevo valor: 7.2 (por ejemplo, generado aleatoriamente entre 0.0 y 10.0).
  - Individuo después de mutar el gen 0: [7.2, 2.0, 3.0, 4.0, 5.0].

  b) **Para el gen 1 - Individual[1] = 2.0**

  - Se genera un número aleatorio, digamos 0.15. Como $0.15 > 0.1$, no se muta.
  - Individuo permanece: [7.2, 2.0, 3.0, 4.0, 5.0].

  De la misma forma, se van mutando (o no) cada uno de los genes para dar origen a un nuevo individuo.

3. **Estrategia de reemplazo:** En el código actual la estrategia de reemplazo adoptada es **reemplazo generacional**, lo que implica que todos los padres son reemplazados por los hijos. La efectividad de esta estrategia se verá en el siguiente capítulo, sin embargo, debido a pruebas experimentales, se demuestra que es la mas rápida, ya que no evalúa el fitness de los nuevos sujetos para contrastarlos con el fitness de los padres. Todos los resultados se adjuntan en el capítulo siguiente.

Capítulo 4

Análisis y Resultados

Ya diseñado el algoritmo y habiendo explicado cada uno de sus componentes, se realizan distintas ejecuciones variando los parámetros involucrados en el algoritmo. A continuación se ofrece un análisis teórico sobre la variación de cada uno de ellos

- **pop_size:** Es el número de individuos de cada generación. La variación de este parámetro resulta fundamental, pues aumentarlo puede ayudar a mejorar la diversidad genética, reduciendo el riesgo de convergencia prematura. Sin embargo, aumentar mucho este valor también puede ser perjudicial, pues aumenta el costo computacional del algoritmo.

- **max_generations:** Es el número máximo de iteraciones del algoritmo, mientras mas iteraciones, se puede mejorar la calidad final de la solución, pues se crearán mas generaciones. Al reducir este número, reduce el tiempo de ejecución del algoritmo, sin embargo, puede perjudicar la calidad de la población final.

- **mutation_rate:** Este parámetro representa la probabilidad de que un gen de un individuo sea mutado. Aumentarlo hace que el algoritmo explore más el espacio de búsqueda, lo que ayuda a evitar óptimos locales. No obstante, aumentar mucho este parámetro puede desestabilizar la población. Por el contrario, reducirlo mucho hará que la población sea estable, sin embargo, aumenta la probabilidad de quedar atrapado en óptimos locales.

- **crossover_rate:** Representa la probabilidad de que dos individuos se crucen para generar nueva descendencia. Se observa que aumentarlo genera una mayor recombinación de los genes, lo que ayuda a explorar de mejor manera el espacio de búsqueda. Por otro lado, disminuirlo puede llevar a una convergencia mas rápida, pero menos robusta.

De esta forma, la variación de estos parámetros ayuda a configurar y balancear de manera apropiada la exploración y explotación en el algoritmo, sin dejar de lado las consideraciones respectivas para no aumentar demasiado el costo computacional del algoritmo, y no afectar la diversidad genética.

A continuación, se ofrece una tabla para cada una de las funciones con los resultados obtenidos en las iteraciones mas relevantes:

Parámetros	**Best overall fitness**	**CPU time**
p_size = 100, gen=500, mut_rate=0.03, cruz_rate=0.7	0.01919480036325183	1.8950 s
p_size = 100, gen=500, mut_rate=0.03, cruz_rate=0.7	0.028222585731253247	2.0007 s
p_size = 100, gen=500, mut_rate=0.05, cruz_rate=0.7	0.0550741358434248	2.0157 s
p_size = 100, gen=500, mut_rate=0.03, cruz_rate=0.5	0.028222585731253247	1.8682 s
p_size = 100, gen=500, mut_rate=0.03, cruz_rate=0.9	0.035129251527054706	.9632 s
p_size = 300, gen=500, mut_rate=0.03, cruz_rate=0.7	0.0029910782976276806	6.2943 s
p_size = 500, gen=500, mut_rate=0.03, cruz_rate=0.7	0.0038416430807313162	11.6517 s
p_size = 300, gen=300, mut_rate=0.03, cruz_rate=0.7	0.018002964199087756	4.1372 s
p_size = 300, gen=700, mut_rate=0.03, cruz_rate=0.7	0.0018276225150444247	8.7857 s
p_size = 300, gen=1000, mut_rate=0.03, cruz_rate=0.7	0.00044655167535623264	13.9170 s
p_size = 300, gen=1500, mut_rate=0.03, cruz_rate=0.7	0.00015539434090605653	17.8265 s
p_size = 300, gen=3000, mut_rate=0.03, cruz_rate=0.7	0.00021805612034597743	38.6774 s

Tabla 4.1: Resultados obtenidos para la función 1.

Parámetros	**Best overall fitness**	**CPU time**
p_size = 100, gen=500, mut_rate=0.03, cruz_rate=0.7	131.30862412342194	2.5526 s
p_size = 100, gen=500, mut_rate=0.03, cruz_rate=0.7	120.38017124009708	2.6491 s
p_size = 100, gen=500, mut_rate=0.05, cruz_rate=0.7	1512.6969639586812	2.3636 s
p_size = 100, gen=500, mut_rate=0.03, cruz_rate=0.5	91.2017559966194	2.2173 s
p_size = 100, gen=500, mut_rate=0.03, cruz_rate=0.9	115.67539706999808	2.3481 s
p_size = 300, gen=500, mut_rate=0.03, cruz_rate=0.7	81.11819284113945	7.5206 s
p_size = 500, gen=500, mut_rate=0.03, cruz_rate=0.7	15.739235113446334	13.133 s
p_size = 300, gen=300, mut_rate=0.03, cruz_rate=0.7	86.0288807364289	4.5259 s
p_size = 300, gen=700, mut_rate=0.03, cruz_rate=0.7	27.805779829333023	10.3932 s
p_size = 300, gen=1000, mut_rate=0.03, cruz_rate=0.7	16.92759275443952	14.7254 s
p_size = 300, gen=1500, mut_rate=0.03, cruz_rate=0.7	3.9346249489044713	20.7289 s
p_size = 300, gen=3000, mut_rate=0.03, cruz_rate=0.7	1.9573303890038776	46.8946 s

Tabla 4.2: Resultados obtenidos para la función 2.

Parámetros	**Best overall fitness**	**CPU time**
p_size = 100, gen=500, mut_rate=0.03, cruz_rate=0.7	-4.687572037430689	1.7613 s
p_size = 100, gen=500, mut_rate=0.03, cruz_rate=0.7	-4.6865345047981615	1.7165 s
p_size = 100, gen=500, mut_rate=0.05, cruz_rate=0.7	-4.687599289746917	1.7961 s
p_size = 100, gen=500, mut_rate=0.03, cruz_rate=0.5	-4.687465021425695	1.747 s
p_size = 100, gen=500, mut_rate=0.03, cruz_rate=0.9	-4.687600143496979	1.7711 s
p_size = 300, gen=500, mut_rate=0.03, cruz_rate=0.7	-4.687643190355216	5.73 s
p_size = 500, gen=500, mut_rate=0.03, cruz_rate=0.7	-4.687653943276551	10.8504 s
p_size = 300, gen=300, mut_rate=0.03, cruz_rate=0.7	-4.687538216476796	3.4203 s
p_size = 300, gen=700, mut_rate=0.03, cruz_rate=0.7	-4.687635602144833	7.922 s
p_size = 300, gen=1000, mut_rate=0.03, cruz_rate=0.7	-4.687647271189949	11.2587 s
p_size = 300, gen=1500, mut_rate=0.03, cruz_rate=0.7	-4.687652130143529	15.91 s
p_size = 300, gen=3000, mut_rate=0.03, cruz_rate=0.7	-4.687657481788897	36.8916 s

Tabla 4.3: Resultados obtenidos para la función 3.

Se aprecia entonces que a medida que aumentan las iteraciones, se consiguen valores mas cercanos a los óptimos globales. En este caso, se sacrifica tiempo de ejecución (velocidad de ejecución), pues cada iteración del algoritmo es mas costosa computacionalmente.

A continuación, se exponen algunos gráficos de convergencia generados para distintos valores de estos parámetros:

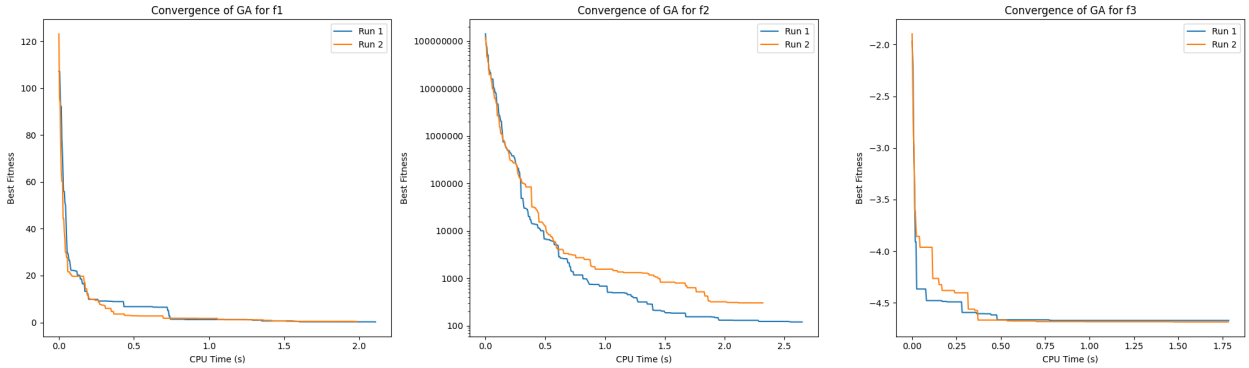

Figura 4.1: Gráfico de convergencia para F1, F2 y F3, con parámetros pop_size=100, max_gen=500, mut_rate=0.01 y cross_rate=0.7.

Cabe resaltar que en la función 2, los valores obtenidos correspondían inicialmente a valores demasiado altos, por lo que se decidió cambiar la escala de fitness (eje y) en el gráfico, y dejar una escala logarítmica, para así poder apreciar la variación de los resultados en el tiempo de cómputo.

Se puede observar mediante los gráficos de convergencia, que el algoritmo tiende a encontrar soluciones bastante buenas, incluso alcanzando los óptimos globales en tiempos considerablemente cortos. Por ejemplo, en *F1*, se demora aproximadamente 2 segundos en encontrar el óptimo global, mientras que en *F3* se demora incluso menos de 2 segundos para llegar al óptimo; *F2* fue una función que, por más iteraciones se realizaran, no permitía llegar al óptimo, aunque se obtuvieron valores bastante cercanos a él (1.18 fue el valor más bajo obtenido).

De esto se puede extraer que los parámetros influyen en la exploración y explotación que realiza el algoritmo, balanceando así estas técnicas, permitiendo realizar un trabajo robusto en cortos lapsos de tiempo.

De esta forma, se aprecia la evolución de los parámetros. En total se hicieron mas de 20 variaciones, las que pueden ser encontradas en el git del presente documento. A raíz del análisis inicial en esta sección, los valores para los parámetros que entregaron mejores resultados fueron:

- **Población inicial:** 300
- **Generaciones máximas:** 3000
- **Tasa de mutación:** 0.03
- **Tasa de cruzamiento:** 0.7

Una vez estabilizados estos valores (manualmente, iteración tras iteración, se localizaron los valores óptimos para estos parámetros), se fue aumentando el número de iteraciones (generaciones) para así mejorar el fitness de los individuos finales. Como se mencionó anteriormente, aumentar la cantidad de generaciones hace que el algoritmo sea mas costoso computacionalmente, pero puede mejorar la calidad de la población final.

4.1. Mejoras Propuestas

El algoritmo diseñado, implementado y analizado anteriormente cumple a cabalidad el objetivo planteado, entrega resultados relativamente buenos para cada una de las funciones evaluadas, y en un buen tiempo. Sin embargo, existen mejoras que pueden complementar de buena forma la metaheurística seleccionada.

4.1.1. Cambio en estrategia de reemplazo: Elitismo

La estrategia de reemplazo adaptada en el algoritmo anterior aseguraba que la nueva generación estaría compuesta en un 100 % por los hijos generados, dejando a los padres atrás. Esto ayuda a tener una mayor diversidad genética, además de explorar mas ampliamente el espacio de búsqueda. Sin embargo, tiene un problema: Los mejores individuos de cada generación se van perdiendo, y junto con ellos, sus genes. De esta forma, se implementó una nueva estrategia de selección de la nueva generación: **el elitismo.**

Esta estrategia de reemplazo consiste en hacer pasar a la siguiente generación un cierto número de los mejores individuos, es decir, si se considera un **elite_size** de 10, entonces los 10 mejores individuos del total de los individuos pasara automáticamente a la nueva generación. Esto preserva los genes de los mejores individuos, y permite que sigan cruzándose con otros para así dar lugar a una población con un mejor fitness.

En particular, el elitismo se implementa de la siguiente manera:

Listing 4.1: Función *Implementacion elitísmo.*

```
def evolve_population(population, fitness, mutation_rate, crossover_rate,
    bounds, elite_size=1):
     new_population = []
     # Seleccion de los mejores individuos (elitismo)
     elite_indices = np.argsort(fitness)[:elite_size]
     for index in elite_indices:
         new_population.append(population[index])

     # Seleccion de los individuos para la reproduccion
     selected_population = tournament_selection(population, fitness)

     # Crear nueva poblacion mediante cruzamiento y mutacion
     for i in range(0, len(population) - elite_size, 2):
         parent1 = selected_population[i]
         parent2 = selected_population[i + 1]
         child1 = crossover_two_point(parent1, parent2, crossover_rate)
         child2 = crossover_two_point(parent2, parent1, crossover_rate)
         new_population.append(mutate(child1, mutation_rate, bounds))
         new_population.append(mutate(child2, mutation_rate, bounds))

     return np.array(new_population)
```

Es importante destacar que la elección de la cantidad de individuos que pasarán por elitismo debe ser cuidadosamente calculada, pues un porcentaje muy alto de elitismo hará que **no se explore** lo suficiente el espacio de búsqueda, detonando una convergencia prematura (y no óptima totalmente). Se recomienda un porcentaje de elitismo de entre un 5 y un 20 %. Las pruebas son expuestas a continuación.

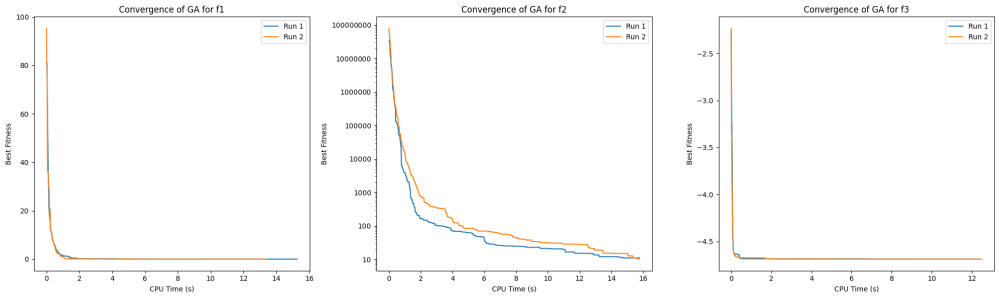

Figura 4.2: Gráfico de convergencia para F1, F2 y F3, con parámetros pop_size=300, max_gen=3000, mut_rate=0.03, cross_rate=0.7 y elite_size=5

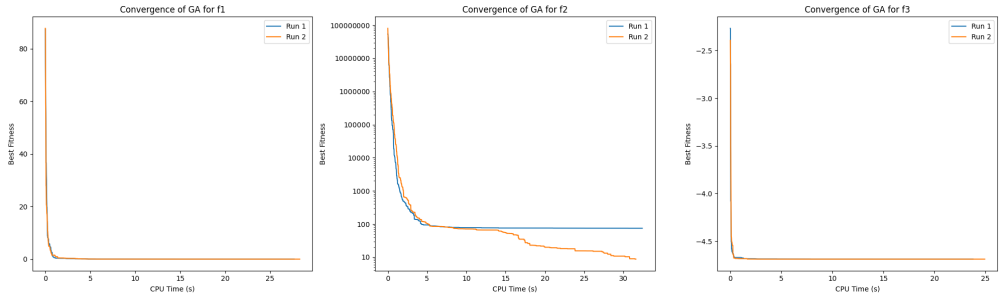

Figura 4.3: Gráfico de convergencia para F1, F2 y F3, con parámetros pop_size=300, max_gen=3000, mut_rate=0.03, cross_rate=0.7 y elite_size=5

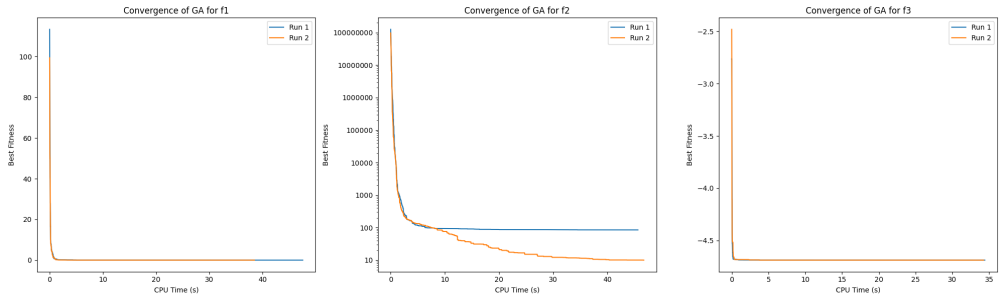

Figura 4.4: Gráfico de convergencia para F1, F2 y F3, con parámetros pop_size=300, max_gen=3000, mut_rate=0.03, cross_rate=0.7 y elite_size=5.

Para cada una de las pruebas anteriores, se optó por un elite_size = 5, lo que representa a un 1.66 % de la población.

A continuación, se realiza una iteración para un elite_size = 33, equivalente a un 11 % de la población, y finalmente un elite_size = 100, lo que equivale a un 33,3 % de la misma. Los valores de los otros parámetros se mantuvieron estables:

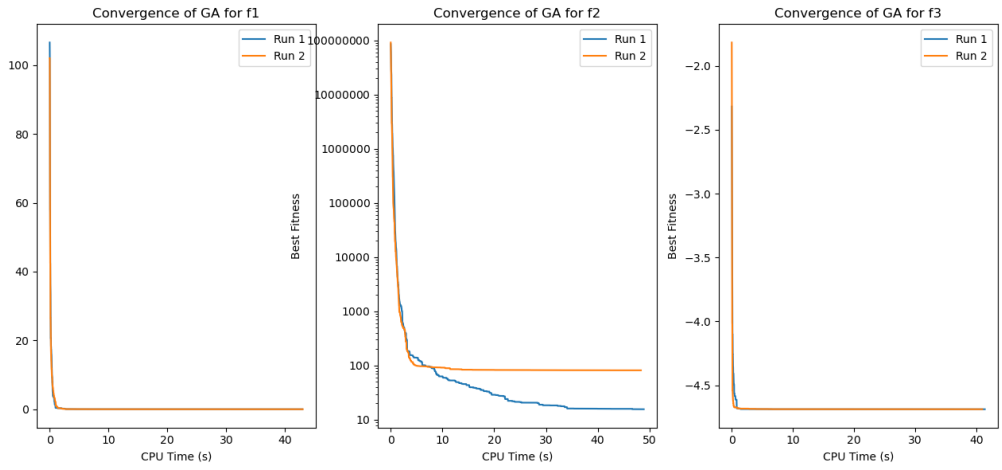

Figura 4.5: 11 % de elitismo

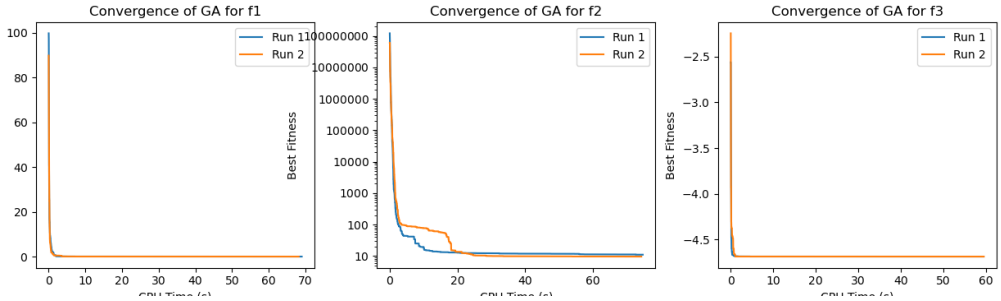

Figura 4.6: 33,3 % de elitismo

4.1.2. Cambio en operador de cruzamiento: 2 Puntos

Otra forma en la que pueden existir mejoras en los resultados es variando el operador de cruzamiento. En el código originalmente implementado, el operador de cruzamiento constaba de un punto, elegido al azar, para poder mezclar la genética entre los dos padres que se seleccionasen. Ahora, se cambia a un cruzamiento por 2 puntos, permitiendo que la variación genética sea mayor entre los descendientes.

Además, este cambio tiene una significancia en la exploración del algoritmo, ya que permite que los genes se mezclen de mejor manera, recorriendo de mejor manera el espacio de búsqueda del problema, evitando también que pueda quedar estancado en algún óptimo local.

A continuación, se adjunta el código donde se implementa el cruzamiento en 2 puntos:

Listing 4.2: Cruzamiento en 2 puntos.

```
def crossover_two_point(parent1, parent2, crossover_rate):
if np.random.rand() < crossover_rate:
    point1 = np.random.randint(1, len(parent1) - 1)
    point2 = np.random.randint(point1, len(parent1))
    return np.concatenate((parent1[:point1], parent2[point1:point2],
       parent1[point2:]))
return parent1
```

En la siguiente función, se hace uso del cruzamiento en 2 puntos. Cabe resaltar que esta implementación se hizo en conjunto con la estrategia de reemplazo de elitismo.

Listing 4.3: Utilización del cruzamiento en dos puntos

```
def evolve_population(population, fitness, mutation_rate,
   crossover_rate, bounds, elite_size=1):
new_population = []
# Seleccion de los mejores individuos (elitismo)
elite_indices = np.argsort(fitness)[:elite_size]
for index in elite_indices:
    new_population.append(population[index])

# Seleccion de los individuos para la reproduccion
selected_population = tournament_selection(population, fitness)

# Crear nueva poblacion mediante cruzamiento y mutacion
for i in range(0, len(population) - elite_size, 2):
    parent1 = selected_population[i]
    parent2 = selected_population[i + 1]
    child1 = crossover_two_point(parent1, parent2, crossover_rate)
    child2 = crossover_two_point(parent2, parent1, crossover_rate)
```

```
            new_population.append(mutate(child1, mutation_rate, bounds))
            new_population.append(mutate(child2, mutation_rate, bounds))

    return np.array(new_population)
```

Finalmente, y a modo de comparar los resultados entre las técnicas, se adjunta una tabla resumen con los mejores resultados de cada una de ellas.

Función	Parámetros	B.O.F	B.O.F CPU Time	Avg. CPU Time
F1	p_size=300, gen=3000, mut_rate=0.03, cross_rate=0.7	0.00021	38.6774 s	41.6122 s
F2	p_size=300, gen=3000, mut_rate=0.03, cross_rate=0.7	1.95733	46.8946 s	46.8845 s
F3	p_size=300, gen=3000, mut_rate=0.03, cross_rate=0.7	-4.68765	36.8916 s	36.7793 s
F1 + Elitismo	p_size=300, gen=3000, mut_rate=0.03, cross_rate=0.7	0.00008	38.1127 s	39.0737 s
F2 + Elitismo	p_size=300, gen=3000, mut_rate=0.03, cross_rate=0.7	7.05690	44.4410 s	44.9722 s
F3 + Elitismo	p_size=300, gen=3000, mut_rate=0.03, cross_rate=0.7	-4.68765	34.3988 s	34.4758 s
F1 + Cross. 2 puntos	p_size=300, gen=3000, mut_rate=0.03, cross_rate=0.7	0.000001	96.7829 s	96.7884 s
F2 + Cross. 2 puntos	p_size=300, gen=3000, mut_rate=0.03, cross_rate=0.7	1.18168	49.7551 s	49.2010 s
F3 + Cross. 2 puntos	p_size=300, gen=3000, mut_rate=0.03, cross_rate=0.7	-4.68765	87.6863 s	88.0323 s

Tabla 4.4: Mejores resultados para cada función con los parámetros determinados.

Como se observa en la tabla anterior, los resultados con el algoritmo genético sin modificaciones entregan valores más que satisfactorios, encontrando el óptimo en 2 de las 3 funciones propuestas. Sin embargo, es cierto que las modificaciones logran mejorar el valor de las funciones, pero requiriendo de un tiempo de cómputo mucho mayor, sobre todo cuando se utiliza el cruzamiento en 2 puntos. **B.O.F** hace referencia al mejor valor encontrado entre todos *(best overall fitness)*.

Lo anterior deja en evidencia el peso que tiene el buen ajuste de los parámetros del algoritmo. Si bien los resultados son de calidad, se ve que empeora en los tiempos de cálculo que debe hacer el sistema para llegar a los resultados relativamente similares. Esto se puede deber a que

- Más exploración: Al tener una mayor diversidad genética, se requiere de más exploración en el espacio de búsqueda, debiendo recorrer más individuos, realizando todo el proceso con ellos.
- Estancamiento: También podría pasar que, al aplicar elitismo, las generaciones no vayan mejorando lo suficiente, y el algoritmo finalice con condiciones de término de no mejora.

De esta manera, se establece que los valores que se establecieron para los parámetros anteriormente logran un ajuste muy bueno con los requerimientos y condiciones de cada una de las funciones anteriores.

4.1.3. Explotación Inicial - Intento fallido

Inicialmente se planteó la idea de que, una vez generada la población inicial, hacer explotación en cada uno de los individuos de la población, para que de esta forma la totalidad de la población inicial sean individuos con un fitness muy bueno. La idea se basa en una variación de *Scatter Search.* Sin embargo, a pesar de que la población inicial era prometedora, no se consiguieron mejoras significativas:

Listing 4.4: Scatter Search con Tabu Search

```
def tabu_search(individual, func, bounds, max_iter=100, tabu_size=10):
    best_individual = individual.copy()
    best_fitness = func(best_individual)
    tabu_list = deque(maxlen=tabu_size)
    tabu_list.append(best_individual.copy())

    for _ in range(max_iter):
        neighborhood = [mutate(best_individual, 1.0, bounds) for _ in
            range(50)]
```

```
        neighborhood_fitness = evaluate_population(neighborhood, func)

        best_neighbor_index = np.argmin(neighborhood_fitness)
        best_neighbor = neighborhood[best_neighbor_index]
        best_neighbor_fitness = neighborhood_fitness[best_neighbor_index]

        if best_neighbor_fitness < best_fitness and not any(np.array_equal
           (best_neighbor, t) for t in tabu_list):
            best_individual = best_neighbor
            best_fitness = best_neighbor_fitness
            tabu_list.append(best_individual.copy())

    return best_individual

def scatter_search(population, func, bounds, max_iter=100, tabu_size=10):
    for i in range(len(population)):
        population[i] = tabu_search(population[i], func, bounds, max_iter,
            tabu_size)
    return population
```

Capítulo 5

Conclusiones

La actividad permitió al grupo obtener una comprensión profunda del funcionamiento de las metaheurísticas basadas en poblaciones y su comportamiento frente a diversas funciones multimodales, las cuales contienen múltiples óptimos locales. Esta comprensión se alcanzó mediante la implementación y experimentación con diferentes algoritmos de optimización, permitiendo a los participantes observar cómo estos algoritmos responden a los desafíos presentados por las funciones multimodales. Las funciones multimodales son especialmente complicadas debido a la presencia de varios picos y valles en el espacio de búsqueda, lo que puede llevar a que los algoritmos se queden atrapados en óptimos locales en lugar de encontrar el óptimo global.

A través de los experimentos descritos, el grupo pudo evaluar la eficacia de varias técnicas de optimización y observar cómo diferentes ajustes y configuraciones impactan en el desempeño del algoritmo. Esto incluyó la comparación de resultados obtenidos con distintas configuraciones de parámetros, así como la implementación de técnicas específicas para mejorar la exploración y explotación del espacio de búsqueda. Se destacó la importancia de adaptar estas técnicas y metaheurísticas para optimizar su desempeño en contextos específicos, influyendo significativamente en la capacidad del algoritmo para escapar de óptimos locales y encontrar soluciones más cercanas al óptimo global.

Además, se logró comprender que los parámetros configurables dentro de los algoritmos genéticos juegan un papel crucial en su desempeño. Estos parámetros incluyen, pero no se limitan a, el tamaño de la población, la tasa de mutación, la tasa de cruzamiento, y el tamaño del elitismo. La correcta configuración de estos parámetros es esencial para balancear adecuadamente la exploración y la explotación dentro del algoritmo. La exploración se refiere a la capacidad del algoritmo para investigar nuevas áreas del espacio de búsqueda, mientras que la explotación se centra en refinar las soluciones existentes. Un balance adecuado entre ambos aspectos es fundamental para permitir la convergencia del algoritmo hacia una solución óptima.

En la implementación y uso del algoritmo genético, también se profundizó en las diferentes partes que componen este tipo de algoritmos. Se examinaron detalladamente las fases de inicialización, selección, cruzamiento, mutación y reemplazo:

- **Inicialización**: Consiste en la creación de una población inicial de individuos de manera aleatoria. Esta fase es crucial porque una buena diversidad inicial puede influir en la capacidad del algoritmo para explorar eficientemente el espacio de búsqueda.

- **Selección:** En esta fase, se seleccionan los individuos que participarán en la generación de la siguiente población. Técnicas como por ejemplo, la selección por torneo, se utilizan para dar preferencia a los individuos con mejor desempeño.

- **Cruzamiento:** Es el proceso mediante el cual se combinan los genes de dos padres para producir descendientes. El cruzamiento puede ser de un punto, dos puntos o de otros tipos, cada uno con sus propias ventajas y desventajas en términos de exploración y explotación.

- **Mutación:** Esta fase introduce variaciones aleatorias en los individuos, ayudando a mantener la diversidad genética dentro de la población y permitiendo al algoritmo explorar nuevas áreas del espacio de búsqueda.
- **Reemplazo:** Finalmente, esta fase determina cómo se forma la nueva población a partir de la actual. Estrategias como el reemplazo generacional completo o el elitismo (donde se preservan los mejores individuos) tienen un impacto significativo en la convergencia del algoritmo.

Los métodos y técnicas aplicados en cada una de estas fases influyen directamente en la eficacia del algoritmo para encontrar soluciones óptimas. Por ello, es crítico realizar un ajuste adecuado de los parámetros antes de intentar resolver una problemática específica. Un buen entendimiento y configuración de estos parámetros puede hacer la diferencia entre encontrar una solución aceptable o una verdaderamente óptima

Finalmente, la actividad no solo permitió una comprensión técnica de las metaheurísticas y sus componentes, sino que también subrayó la importancia de los parámetros configurables en el desempeño del algoritmo. A través de una serie de experimentos y ajustes, se pudo ver de primera mano cómo pequeños cambios en la configuración pueden tener un impacto significativo en los resultados, destacando la necesidad de un ajuste fino y una comprensión profunda de los mecanismos internos de los algoritmos genéticos para optimización.

Ahora, cerrando la actividad y entregando los mejores resultados, es que en el caso de la implementación del grupo, la configuración que mejor se adaptó a las 3 funciones fue utilizar estrategias de reemplazo generacionales, cruzamiento en un punto, con probabilidades de mutación de un 3 %, población inicial de 300 individuos, y una cantidad máxima de generaciones de 3.000.

Capítulo 6

Anexos

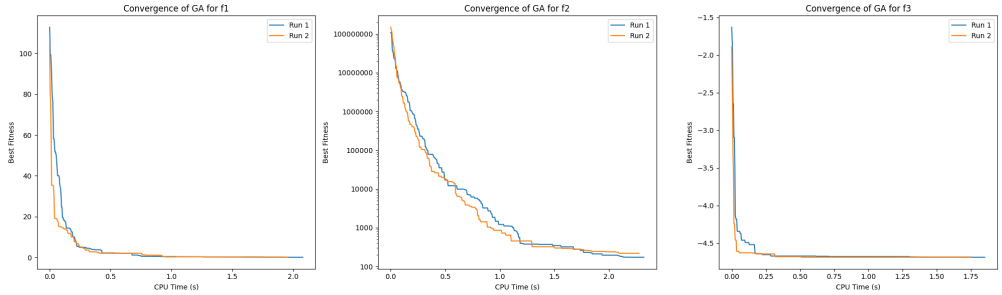

Figura 6.1: Gráfico de convergencia para F1, F2 y F3, con parámetros pop_size=100, max_gen=500, mut_rate=0.05 y cross_rate=0.5.

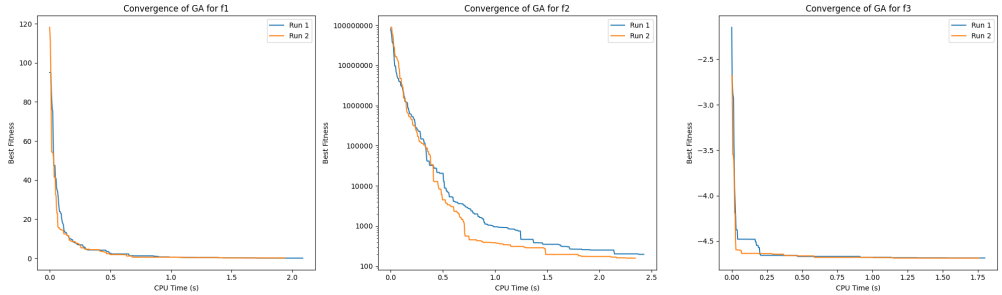

Figura 6.2: Gráfico de convergencia para F1, F2 y F3, con parámetros pop_size=100, max_gen=500, mut_rate=0.03 y cross_rate=0.7.

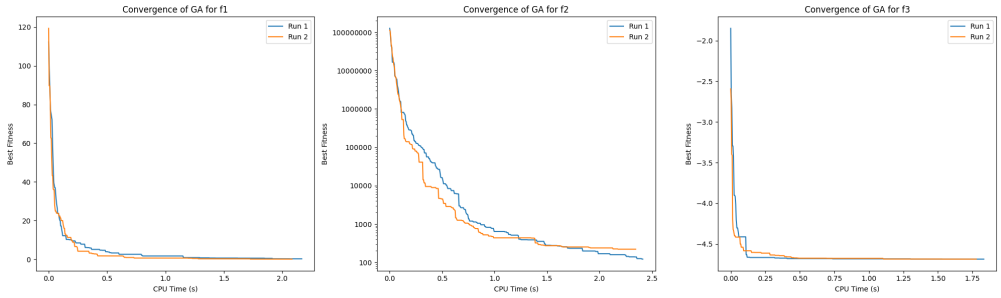

Figura 6.3: Gráfico de convergencia para F1, F2 y F3, con parámetros pop_size=100, max_gen=500, mut_rate=0.03 y cross_rate=0.9.

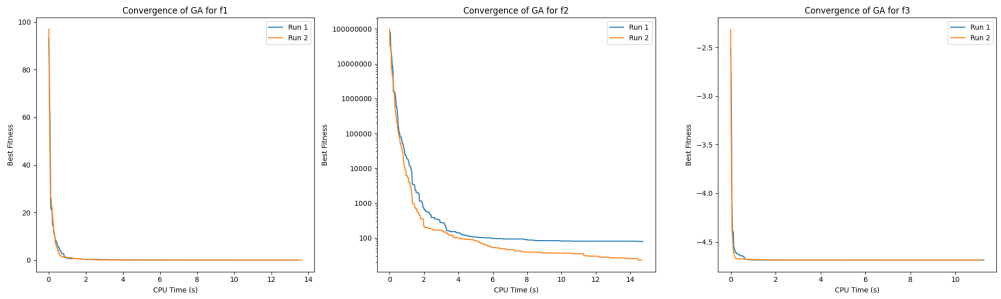

Figura 6.4: Gráfico de convergencia para F1, F2 y F3, con parámetros pop_size=300, max_gen=1000, mut_rate=0.03 y cross_rate=0.7.

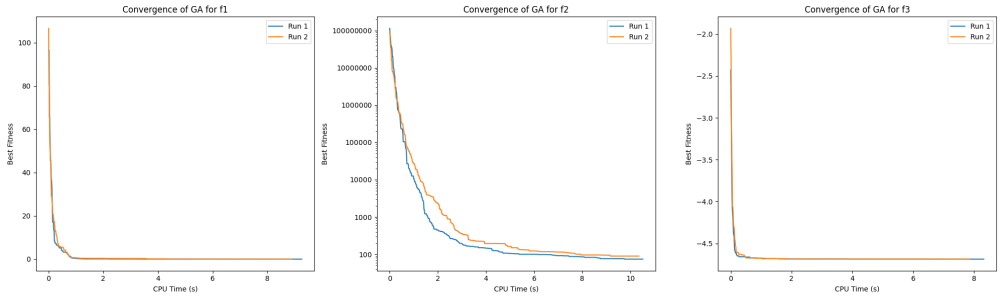

Figura 6.5: Gráfico de convergencia para F1, F2 y F3, con parámetros pop_size=300, max_gen=700, mut_rate=0.03 y cross_rate=0.7.

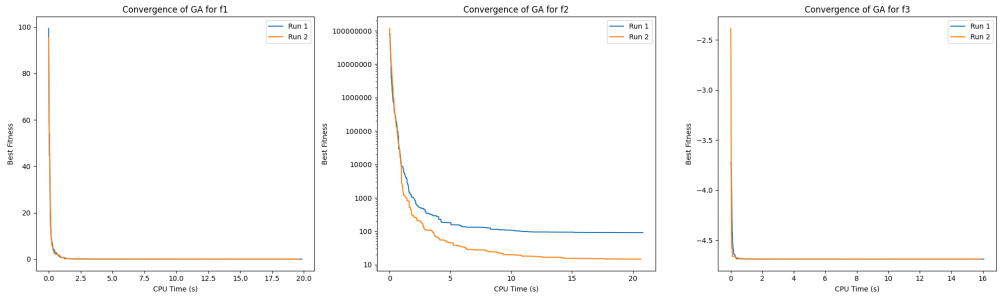

Figura 6.6: Gráfico de convergencia para F1, F2 y F3, con parámetros pop_size=300, max_gen=1500, mut_rate=0.03 y cross_rate=0.7.

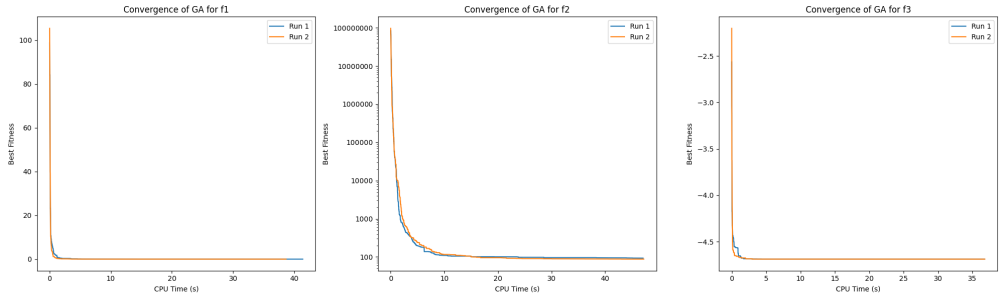

Figura 6.7: Gráfico de convergencia para F1, F2 y F3, con parámetros pop_size=300, max_gen=3000, mut_rate=0.03 y cross_rate=0.7.

Repositorio de la actividad: `https://github.com/iJass21/Algoritmos_exactos_y_metaheuristicas/`