
# UNIVERSIDAD DE GRANADA

Inteligencia Computacional

## Reconocimiento Óptico de Caracteres

**Autores**

Antonio José Muriel Gálvez


Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, octubre de 2024

# Índice

**Introducción** **2**

**Implementación** **2**

Conjunto de Datos y Preparación . . . . . 2

Configuración inicial . . . . . 3

Algoritmo Genético: Estándar . . . . . 3

Algoritmo Genético: Variante Baldwiniana . . . . . 8

Algoritmo Genético: Variante Lamarckiana . . . . . 11

Algoritmo Génetico: Variante Lamarckiana v2 . . . . . 13

Algoritmo Genético: Variante Lamarckiana v3 . . . . . 16

**Experimentación y Resultados** **19**

Resultados del algoritmo genético estándar . . . . . 19

Resultados de la variante baldwiniana . . . . . 21

Resultados de la Variante Lamarckiana . . . . . 23

Resultados de la Variante Lamarckiana v2 . . . . . 24

Resultados de la Variante Lamarckiana v3 . . . . . 25

Resultados de Comparativa de los Algoritmos Genéticos . . . . . 27

**Conclusiones** **29**

# Introducción

En este trabajo se aborda la resolución del Problema de Asignación Cuadrática, un problema de optimización combinatoria ampliamente estudiado debido a su complejidad computacional y su aplicabilidad en diversas áreas. El QAP es un problema NP-difícil, lo que implica que encontrar soluciones exactas para instancias grandes resulta inviable en tiempos razonables. Por ello, el uso de algoritmos metaheurísticos, como los algoritmos genéticos, es una estrategia efectiva para aproximar soluciones de alta calidad.

El objetivo de este proyecto es implementar y optimizar un algoritmo genético avanzado que incorpore estrategias modernas como selección con nichos, cruce basado en ciclos, mutación adaptativa y búsqueda local, entre otras. Estas mejoras están diseñadas para enfrentar la complejidad de instancias del QAP de gran tamaño, como la instancia tai256c, que consta de 256 instalaciones y 256 ubicaciones. A través de esta implementación, se busca no solo encontrar soluciones competitivas, sino también explorar y analizar cómo las diferentes configuraciones de parámetros y operadores influyen en el rendimiento del algoritmo.

A lo largo de este trabajo se documentan las estrategias empleadas, los resultados obtenidos y los desafíos enfrentados, destacando la importancia de los algoritmos metaheurísticos en problemas de optimización complejos y su capacidad para adaptarse a escenarios de gran escala.

# Implementación

## Conjunto de Datos y Preparación

La preparación de los datos es esencial en este tipo de problemas porque las matrices de flujos y distancias determinan cómo se calcula el costo de una solución (es decir, el fitness). En este caso, los datos se leen desde un archivo y se procesan para separarlos en estas dos matrices.

```
1 data = np.loadtxt('qap.datos/tai256c.dat', skiprows=1)
```

- Carga el contenido del archivo tai256c.dat como un arreglo de NumPy.
- Omite la primera fila del archivo, ya que contiene metadatos o encabezados que no forman parte de las matrices.

```
flow = np.int32(data[:256])
distances = np.int32(data[256:])
```

Aunque np.loadtxt puede cargar los datos como flotantes, en este caso se convierten explícitamente a enteros con np.int32. Esto se debe a que tanto las distancias como los flujos son cantidades discretas.

- Las primeras 256 filas del archivo (data[:256]) se asignan a la matriz de flujos, flow.

  Cada entrada $flow[i][j]$ representa la cantidad de flujo entre la instalación i y la instalación j.

- Las siguientes 256 filas (data[256:]) se asignan a la matriz de distancias, distances.

  Cada entrada $distances[i][j]$ representa la distancia entre la ubicación i y la ubicación j.

## Configuración inicial

Con esto conseguimos proporcionar una base de soluciones iniciales para el algoritmo

- Se fija una semilla para garantizar reproducibilidad.
- Se genera una población inicial de 5000 individuos, cada uno representado como una permutación de 256 elementos, correspondientes a las asignaciones de instalaciones a ubicaciones.

```
seed = 2024
random.seed(seed)
np.random.seed(seed)

population = np.array([np.random.permutation(np.arange(256))
    for _ in range(5000)])
```

## Algoritmo Genético: Estándar

Un algoritmo genético (GA) es una técnica de optimización inspirada en la evolución natural. En este caso, el GA estándar resuelve un problema cuadrático

de asignación (QAP), donde el objetivo es encontrar la mejor permutación de ubicaciones para minimizar el costo total asociado a flujos y distancias.

Función de fitness:

Calcula el costo total de cada individuo en la población, usando operaciones matriciales para evitar bucles innecesarios.

Determina qué soluciones son más adecuadas para el problema según los costos calculados.

```
def fitness_pop(population, flow, distances):
    return np.sum(flow[np.newaxis, :, :] * distances[
        population[:,:,np.newaxis], population[:, np.newaxis,
         :]], axis=(1,2))
```

Selección por torneo:

Selecciona individuos para reproducirse utilizando un torneo, donde se elige al mejor entre k individuos seleccionados aleatoriamente. El uso de pesos inversamente proporcionales al fitness favorece a individuos con menor costo.

Permite garantizar que las mejores soluciones tengan más probabilidades de reproducirse.

```
def selection_tournament(population, fitness_vals, k=3):
    selected_indices = np.array([random.choices(np.arange(
        len(fitness_vals)), k=k, weights=1/fitness_vals)[0]
        for _ in range(len(population))])
    return population[selected_indices]
```

Cruce en la población:

Genera una nueva población a partir de la población actual mediante el operador de cruce.

1. Se eligen dos individuos de la población de manera aleatoria como candidatos a cruce.
2. Se realiza el cruce solo si un número aleatorio generado está por debajo de crossover_prob.
3. El operador crossover_order genera dos hijos, cada uno con información genética combinada de ambos padres.
4. Si no ocurre el cruce, los hijos son copias exactas de los padres

5. Los nuevos individuos generados (o copias de los padres) se añaden a la nueva población.

6. La nueva población se devuelve como un numpy.array para facilitar las operaciones posteriores.

Cruce de orden (OX):

Implementa el operador de cruce de orden (Order Crossover, OX). Una porción del padre 1 se transfiere directamente al hijo, y las posiciones restantes se llenan siguiendo el orden en el que aparecen en el padre 2.

Permite combinar información genética de dos padres para generar nuevos individuos.

```
def crossover_order(parent1, parent2):
    n = len(parent1)
    start, end = sorted(random.sample(range(n), 2))
    child = -np.ones(n, dtype=int)
    child[start:end+1] = parent1[start:end+1]

    position = 0
    for gene in parent2:
        if gene not in child:
            while child[position] != -1:
                position += 1
            child[position] = gene
    return child
```

Mutación por intercambio:

Cambia aleatoriamente dos elementos de un individuo con cierta probabilidad.

Introduce diversidad genética para evitar el estancamiento en óptimos locales.

```
def mutate_population(population, mutation_prob=0.3):
    for individual in population:
        if random.random() < mutation_prob:
            i, j = random.sample(range(len(individual)), 2)
            individual[i], individual[j] = individual[j],
                individual[i]
```

Búsqueda local (2-opt limitada):

Optimiza localmente un individuo intercambiando segmentos de la permutación (2-opt) durante un número limitado de iteraciones.

Mejora las soluciones individuales tras cada generación.

```
def local_search_2opt_limited(permutation, flow, distances,
    max_iter=20):
    n = len(permutation)
    best_cost = calculate_cost(permutation, flow, distances)
    for _ in range(max_iter):
        i, j = sorted(random.sample(range(n), 2))
        new_perm = permutation.copy()
        new_perm[i:j+1] = list(reversed(new_perm[i:j+1]))
        new_cost = calculate_cost(new_perm, flow, distances)
        if new_cost < best_cost:
            permutation = new_perm
            best_cost = new_cost
    return permutation
```

Costo asociado:

Esta función calcula el costo asociado a una única permutación (individuo), que representa una solución al problema.

- permutation: Representa el orden de asignación de instalaciones a ubicaciones.
- indexed_distances: Es una matriz que reorganiza las distancias de acuerdo a la permutación proporcionada.

Es una alternativa al cálculo de fitness para evaluar soluciones individuales, y se utiliza específicamente en la búsqueda local.

```
def calculate_cost(permutation, flow, distances):
    indexed_distances = distances[permutation][:,
        permutation]
    cost = np.sum(flow * indexed_distances)
    return cost
```

Aplicar búsqueda local a la élite:

Se aplica la búsqueda local solo a los mejores individuos de la población, seleccionados según el fitness.

Maximizando el rendimiento sin afectar significativamente el tiempo de ejecución.

```python
def apply_2opt_to_elite(population, fitness_vals, flow,
    distances, elite_ratio=0.2):
    elite_size = int(len(population) * elite_ratio)
    elite_indices = np.argsort(fitness_vals)[:elite_size]
    for idx in elite_indices:
        population[idx] = local_search_2opt_limited(
            population[idx], flow, distances)
    return population
```

Configuración del ciclo evolutivo:

- Generaciones: Es el número de iteraciones que el algoritmo ejecutará como máximo. Cada generación representa un ciclo completo de selección, cruce, mutación y optimización.
- Probabilidad de cruce: Indica la probabilidad de que dos padres seleccionados se crucen para generar nuevos hijos. Si no ocurre el cruce, los hijos serán copias exactas de los padres.
- Probabilidad de mutación: Define la probabilidad de que ocurra una mutación en un individuo durante cada generación. Una mayor probabilidad aumenta la diversidad genética, pero puede ralentizar la convergencia.
- Variables para rastrear la mejor solución: best_solution almacena el mejor individuo encontrado en todo el proceso evolutivo best_fitness guarda el valor del fitness asociado a esa mejor solución.

  Inicialmente, best_solution es None y best_fitness se define como infinito para garantizar que cualquier solución válida sea mejor.
- Se calcula el fitness de la población inicial, utilizando la función de evaluación definida previamente.

```python
generations = 250
crossover_prob = 0.9
mutation_prob = 0.3
stagnation_limit = 100

best_solution = None
best_fitness = float('inf')

fitness_vals = fitness_pop(population, flow, distances)
```

Ciclo evolutivo:

Consiste en iterar el proceso evolutivo para mejorar progresivamente las soluciones.

1. Selección: Se eligen los padres usando selección por torneo.
2. Cruce: Se genera una nueva población cruzando a los padres.
3. Mutación: Se aplica el operador de mutación para introducir diversidad.
4. Búsqueda local: Se optimizan las mejores soluciones.
5. Evaluación: Se calcula el fitness de la nueva población.
6. Se actualiza el mejor individuo encontrado.

```python
for generation in range(generations):
    # Selección
    selected_population = selection_tournament(population,
        fitness_vals)

    # Cruce
    offspring = crossover_population(selected_population,
        crossover_prob)

    # Mutación
    population = mutate_population(offspring, mutation_prob)

    # Aplicar búsqueda local a la élite
    population = apply_2opt_to_elite(population,
        fitness_vals, flow, distances, elite_ratio=0.2)

    # Recalcular fitness
    fitness_vals = fitness_pop(population, flow, distances)

    # Guardar el mejor individuo
    gen_best_fitness = fitness_vals.min()
    if gen_best_fitness < best_fitness:
        best_fitness = gen_best_fitness
        best_solution = population[np.argmin(fitness_vals)]

    print(f"Generación {generation + 1}: Mejor Fitness = {
        best_fitness}")
    if best_fitness <= 44759294:
        print("Objetivo alcanzado, deteniendo el proceso.")
        break
```

## Algoritmo Genético: Variante Baldwiniana

La Variante Baldwiniana introduce un cambio clave al aplicar la búsqueda local a cada individuo de la población antes de la selección. Esto modifica el flujo

del algoritmo genético estándar al aprovechar el refinamiento local para guiar el proceso evolutivo. A continuación, se explican en profundidad las partes nuevas o modificadas

Búsqueda Baldwiniana

Aplica una mejora local a todos los individuos de la población antes de realizar la selección.

En la variante Baldwiniana, la búsqueda local no modifica directamente el individuo en la población, sino que genera una versión "mejorada". Esto significa que la capacidad adaptativa (fitness) se utiliza para influir en la selección sin alterar la representación genética.

Proceso:

1. Itera sobre cada individuo en la población.
2. Aplica la búsqueda local limitada mediante la función local_search_2opt_limited.
3. Agrega la versión mejorada del individuo a una nueva lista llamada improved_population.

```
def apply_baldwinian_search(population, flow, distances,
    max_iter=20):
    improved_population = []
    for individual in population:
        improved_individual = local_search_2opt_limited(
            individual, flow, distances, max_iter)
        improved_population.append(improved_individual)
    return np.array(improved_population)
```

- population: Conjunto de individuos que representan soluciones al problema.
- flow y distances: Matrices del problema QAP que se utilizan para calcular el costo.
- max_iter: Número máximo de iteraciones para la búsqueda local.

Devuelve una nueva población que refleja las mejoras obtenidas mediante la búsqueda local.

Modificaciones en el ciclo evolutivo

La variante Baldwiniana afecta directamente el flujo del ciclo evolutivo. A continuación, se describen los cambios específicos:

En el algoritmo genético estándar, la selección se realiza directamente sobre la población generada. En la variante Baldwiniana, antes de la selección, cada individuo pasa por un proceso de búsqueda local para producir una versión mejorada. Esto cambia las probabilidades de ser seleccionado, ya que los individuos con mejor capacidad adaptativa tienen una ventaja, sin alterar su estructura genética original.

```
improved_population = apply_baldwinian_search(population,
    flow, distances)
```

Flujo general

1. Búsqueda local: Se genera una población "mejorada" con la función apply_baldwinian_search.
2. Selección basada en la población mejorada:La selección se realiza considerando los individuos mejorados. Esto incrementa las probabilidades de que las mejores soluciones sean elegidas como padres.

```
        selected_population = selection_tournament(
            improved_population, fitness_vals)
```

   Restante igual al estándar:

3. Cruce: Genera descendencia combinando características genéticas de los padres.
4. Mutación: Introduce variaciones en la población para mantener la diversidad genética.
5. Cálculo del fitness: Evalúa la nueva población.

Comparación con el algoritmo genético estándar

Tabla 1: Comparación entre Algoritmo Genético Estándar y Variante Baldwiniana

| **Aspecto** | **Algoritmo Estándar** | **Variante Baldwiniana** |
|---|---|---|
| Selección | Directa en la población | Basada en población mejorada |
| Búsqueda local | Solo en élite | En toda la población antes de selección |
| Costo computacional | Moderado | Elevado por búsqueda local masiva |
| Convergencia | Más lenta | Más rápida |
| Diversidad genética | Mejor mantenida | Menor diversidad por dominancia de mejores soluciones |

## Algoritmo Genético: Variante Lamarckiana

La variante Lamarckiana del algoritmo genético se basa en la idea de que los individuos pueden mejorar sus características (soluciones) a través de un proceso de aprendizaje local, y estas mejoras se reflejan directamente en la población. En comparación con el algoritmo estándar y la variante Baldwiniana, esta estrategia tiene las siguientes diferencias clave:

Búsqueda Lamarckiana

En el enfoque Lamarckiano, se aplica la búsqueda local (local_search_2opt_limited) a cada individuo de la población, y los resultados mejorados se actualizan directamente en la población. Esto significa que los cambios realizados en los individuos son permanentes y se transmiten a la siguiente generación.

```
def apply_lamarckian_search(population, flow, distances,
    max_iter=20):
    # Mejorar cada individuo y actualizar permanentemente la
        población
    for i in range(len(population)):
        population[i] = local_search_2opt_limited(population
            [i], flow, distances, max_iter)
    return population
```

- Diferencia clave con Baldwiniana: En la variante Baldwiniana, los individuos mejorados por la búsqueda local no actualizan permanentemente la población. En la Lamarckiana, los cambios se aplican directamente y se heredan.
- Costo computacional: Es más alto en Lamarckiana, ya que todos los individuos pasan por un proceso de búsqueda local intensiva, y el resultado se almacena directamente en la población.
- Impacto: El fitness de los individuos mejora antes de la selección, lo que puede acelerar la convergencia del algoritmo.

Ciclo evolutivo

En la variante Lamarckiana, el ciclo evolutivo se modifica para incluir la actualización permanente de los individuos antes de cualquier otra operación genética.

```
for generation in range(generations):
    # Aplicar la variante Lamarckiana: mejora de la població
        n de forma permanente
```

```
    population = apply_lamarckian_search(population, flow,
        distances)

    # Selección
    selected_population = selection_tournament(population,
        fitness_vals)

    # Cruce
    offspring = crossover_population(selected_population,
        crossover_prob)

    # Mutación
    population = mutate_population(offspring, mutation_prob)

    # Recalcular fitness
    fitness_vals = fitness_pop(population, flow, distances)
```

La mejora Lamarckiana (apply_lamarckian_search) se realiza antes de la selección. Esto asegura que la población seleccionada para la reproducción esté compuesta por individuos con mejor fitness.

- Selección: Se basa en una población ya mejorada, lo que aumenta la probabilidad de seleccionar individuos de alta calidad.
- Diversidad: La fuerte mejora puede reducir la diversidad genética, ya que las mejores soluciones tienden a dominar más rápidamente.

Comparación con Baldwiniana y Algoritmo Estándar

| Aspecto | Algoritmo Estándar | Variante Baldwiniana | Variante Lamarckiana |
|---|---|---|---|
| **Búsqueda local** | No aplica | Aplicada a todos, pero no actualiza la población | Aplicada a todos y actualiza la población |
| **Actualización** | Ninguna | Fitness mejorado, pero no heredado | Fitness mejorado y heredado |
| **Costo computacional** | Moderado | Elevado (búsqueda local masiva) | Muy elevado (búsqueda local + actualizaciones) |
| **Convergencia** | Más lenta | Más rápida debido a la mejora de la población | Muy rápida, mejoras permanentes |
| **Diversidad genética** | Mejor mantenida | Riesgo de menor diversidad | Alto riesgo de reducción de diversidad |
| **Fitness inicial** | Sin mejoras | Mejora antes de selección | Mejora antes y después de selección |

Tabla 2: Comparación entre el algoritmo genético estándar, la variante Baldwiniana y la variante Lamarckiana.

## Algoritmo Génetico: Variante Lamarckiana v2

Esta variante es más sofisticada que las versiones estándar y Baldwiniana, ya que introduce técnicas avanzadas como el elitismo, la mutación adaptativa, la búsqueda local paralelizada y estrategias para contrarrestar el estancamiento, lo que la hace más robusta y eficiente en la búsqueda de soluciones óptimas para problemas complejos como el QAP.

Selección con Elitismo: Se incorpora elitismo en la selección. El elitismo asegura que una fracción de los mejores individuos (determinado por elite_ratio, en este caso el 10 %) no se vea afectada por el proceso de cruce y mutación, protegiéndolos de posibles deterioros. El resto de la población se selecciona de acuerdo a la selección por torneo, con un factor de pesos invertidos basado en el fitness.

```
1 def selection_tournament_with_elitism(population,
      fitness_vals, k=3, elite_ratio=0.1):
2     elite_size = int(len(population) * elite_ratio)
```

```
    elite_indices = np.argsort(fitness_vals)[:elite_size]
    elites = population[elite_indices]

    selected_indices = np.array([random.choices(np.arange(
        len(fitness_vals)), k=k, weights=1/fitness_vals)[0]
        for _ in range(len(population) - elite_size)])
    selected = population[selected_indices]

    return np.vstack((elites, selected))
```

Cruce basado en ciclos (Cycle Crossover, CX):

Se utiliza Cycle Crossover (CX), un cruce especializado en problemas de permutación. Este tipo de cruce mantiene la estructura de los padres, respetando los ciclos de permutación y asegura que los hijos sean soluciones válidas sin duplicados.

```
def crossover_cycle(parent1, parent2):
    n = len(parent1)
    child = -np.ones(n, dtype=int)
    start = 0
    indices = set()

    while start not in indices:
        indices.add(start)
        start = np.where(parent2 == parent1[start])[0][0]

    for idx in indices:
        child[idx] = parent1[idx]

    for idx in range(n):
        if child[idx] == -1:
            child[idx] = parent2[idx]

    return child
```

Mutación Adaptativa:

La probabilidad de mutación depende del número de generaciones sin mejora (stagnation_counter). Si el algoritmo está atascado en un óptimo local y no mejora, se aumenta la probabilidad de mutación para promover la diversidad genética y escapar de esa trampa. La probabilidad de mutación es ajustada dinámicamente y tiene un máximo de 50 %.

```
def mutate_population_adaptive(population, mutation_prob,
    stagnation_counter, max_stagnation):
```

```
    adaptive_prob = mutation_prob + (0.1 *
        stagnation_counter / max_stagnation)
    adaptive_prob = min(adaptive_prob, 0.5)  # Limitar la
        mutación adaptativa a un máximo de 50%

    for individual in population:
        if random.random() < adaptive_prob:
            i, j = random.sample(range(len(individual)), 2)
            individual[i], individual[j] = individual[j],
                individual[i]
    return population
```

Búsqueda Local Paralelizada (2-opt):

Solo los mejores individuos (élite) se someten a búsqueda local, lo cual reduce el costo computacional. La búsqueda local es paralelizada utilizando joblib, lo que mejora la eficiencia y velocidad de mejora de los individuos élite. Los individuos mejorados se reinsertan en la población para que su progreso sea aprovechado en futuras generaciones.

```
def parallel_local_search(population, flow, distances,
    max_iter=20, elite_ratio=0.1):
    elite_size = int(len(population) * elite_ratio)
    elite_indices = np.argsort(fitness_pop(population, flow,
        distances))[:elite_size]
    elite_population = population[elite_indices]

    improved_elites = Parallel(n_jobs=-1)(
        delayed(local_search_2opt_limited)(individual, flow,
            distances, max_iter)
        for individual in elite_population
    )

    population[elite_indices] = improved_elites
    return population
```

Estrategia contra el Estancamiento

Estrategia contra el estancamiento. Si el número de generaciones sin mejora supera un umbral (stagnation_limit), se agregan nuevos individuos generados aleatoriamente a la población. Esto introduce diversidad en la población, evitando que se quede atrapada en óptimos locales durante demasiado tiempo.

```
if no_improvement_counter >= stagnation_limit // 2:
    new_individuals = np.array([np.random.permutation(np.
        arange(256)) for _ in range(len(population) // 10)])
```

```
    population = np.vstack((population, new_individuals))
```

## Algoritmo Genético: Variante Lamarckiana v3

Esta versión del algoritmo genético para el Problema de Asignación Cuadrática (QAP) representa una mejora significativa basada en la incorporación de estrategias avanzadas para optimizar el rendimiento y superar las limitaciones de versiones previas. A continuación, se explican cada uno de los componentes y mejoras clave:

Búsqueda Local Mejorada: 3-opt

La búsqueda local 3-opt explora soluciones vecinas modificando tres segmentos de una permutación.

Es más efectiva que 2-opt, ya que evalúa un espacio de búsqueda más amplio, permitiendo escapar de óptimos locales.

```
def mutate_population_advanced(population, mutation_prob,
    stagnation_counter, max_stagnation):
    adaptive_prob = mutation_prob + (0.1 *
        stagnation_counter / max_stagnation)
    adaptive_prob = min(adaptive_prob, 0.5)

    for individual in population:
        if random.random() < adaptive_prob:
            if random.random() < 0.5:
                # Scramble Mutation
                i, j = sorted(random.sample(range(len(
                    individual)), 2))
                np.random.shuffle(individual[i:j+1])
            else:
                # Swap Mutation
                i, j = random.sample(range(len(individual)),
                     2)
                individual[i], individual[j] = individual[j
                    ], individual[i]
    return population
```

- Aplica un número limitado de iteraciones (max_iter), equilibrando la calidad de la mejora y el tiempo de ejecución.
- Paralelización con joblib: Cada individuo en la élite se optimiza en paralelo, reduciendo el tiempo de ejecución total.

Mutación Avanzada con Adaptación

Incrementa dinámicamente si el algoritmo no mejora durante varias generaciones y ayuda a superar el estancamiento en óptimos locales.

```
def mutate_population_advanced(population, mutation_prob, stagnation_counter,
     max_stagnation):
    adaptive_prob = mutation_prob + (0.1 * stagnation_counter /
        max_stagnation)
    adaptive_prob = min(adaptive_prob, 0.5)

    for individual in population:
        if random.random() < adaptive_prob:
            if random.random() < 0.5:
                # Scramble Mutation
                i, j = sorted(random.sample(range(len(individual)), 2))
                np.random.shuffle(individual[i:j+1])
            else:
                # Swap Mutation
                i, j = random.sample(range(len(individual)), 2)
                individual[i], individual[j] = individual[j], individual[i]
    return population
```

Combina dos tipos de mutaciones:

- Mutación Scramble: Reorganiza aleatoriamente un segmento de la permutación y explora nuevas áreas del espacio de búsqueda.
- Mutación Swap: Intercambia dos elementos aleatorios en la permutación y realiza pequeños ajustes, útiles para la explotación local.

Selección con Nichos y Elitismo

En la versión anterior, se utilizaba un torneo estándar sin nichos, lo que podía llevar a una pérdida de diversidad genética y convergencia prematura.

Evita que toda la población se concentre alrededor de un único óptimo e incrementa la probabilidad de encontrar soluciones globalmente óptimas.

```
def mutate_population_advanced(population, mutation_prob,
    stagnation_counter, max_stagnation):
    adaptive_prob = mutation_prob + (0.1 *
        stagnation_counter / max_stagnation)
    adaptive_prob = min(adaptive_prob, 0.5)

    for individual in population:
        if random.random() < adaptive_prob:
            if random.random() < 0.5:
                # Scramble Mutation
                i, j = sorted(random.sample(range(len(
                    individual)), 2))
```

```
                np.random.shuffle(individual[i:j+1])
            else:
                # Swap Mutation
                i, j = random.sample(range(len(individual)),
                    2)
                individual[i], individual[j] = individual[j
                    ], individual[i]
    return population
```

- Elitismo: Preserva los mejores individuos (10 %) de la población y garantiza que las mejores soluciones no se pierdan en las generaciones futuras.
- Control de Nichos: Divide la población en pequeños subgrupos (nichos) y selecciona el mejor individuo de cada nicho, manteniendo la diversidad genética.

Generación de Nuevos Individuos en Caso de Estancamiento

Cuando el algoritmo detecta estancamiento (sin mejoras durante varias generaciones), se generan nuevos individuos aleatorios. Estos nuevos individuos introducen diversidad genética para revitalizar la población.

La versión anterior no contaba con un mecanismo para detectar ni contrarrestar el estancamiento.

```
if no_improvement_counter >= stagnation_limit // 2:
    new_individuals = np.array([np.random.permutation(np.
        arange(256)) for _ in range(len(population) // 10)])
    population = np.vstack((population, new_individuals))
```

Enfriamiento Simulado

Combina exploración global y local mediante la probabilidad de aceptar soluciones peores basada en una temperatura decreciente. Explora soluciones vecinas al reordenar segmentos de la permutación, aceptando cambios si son mejores o con cierta probabilidad si son peores.

La versión anterior no incluía esta técnica adicional de optimización local.

```
def simulated_annealing(permutation, flow, distances,
    initial_temp=1000, cooling_rate=0.99, max_iter=50):
    def calculate_cost(permutation):
        indexed_distances = distances[permutation][:,
            permutation]
        return np.sum(flow * indexed_distances)
```

```
5
6       current_temp = initial_temp
7       best_perm = permutation.copy()
8       best_cost = calculate_cost(permutation)
9
10      for _ in range(max_iter):
11          i, j = sorted(random.sample(range(len(permutation)),
                2))
12          new_perm = permutation.copy()
13          new_perm[i:j+1] = list(reversed(new_perm[i:j+1]))
14          new_cost = calculate_cost(new_perm)
15          if new_cost < best_cost or random.random() < np.exp
                (-(new_cost - best_cost) / current_temp):
16              permutation = new_perm
17              best_cost = new_cost
18              best_perm = permutation
19          current_temp *= cooling_rate
20      return best_perm
```

Paralelización

Utiliza joblib para paralelizar el proceso de búsqueda local 3-opt en los mejores individuos.

La versión anterior no empleaba paralelización, lo que limitaba la escalabilidad.

```
improved_elites = Parallel(n_jobs=-1)(
    delayed(local_search_3opt)(individual, flow, distances,
        max_iter)
    for individual in elite_population
)
```

# Experimentación y Resultados

## Resultados del algoritmo genético estándar

El algoritmo converge rápidamente en las primeras generaciones (1-7), donde se produce el mayor descenso en el fitness. Luego, la convergencia es extremadamente lenta y prolongada, lo que sugiere un desequilibrio entre exploración (buscar soluciones nuevas) y explotación (refinar las soluciones actuales).

La estabilización en valores altos (49,216,524 y luego 49,161,486) por muchas generaciones indica que el algoritmo podría haber quedado atrapado en óptimos

locales durante largos períodos. Sin embargo, la mejora tardía muestra que el algoritmo finalmente logra escapar.

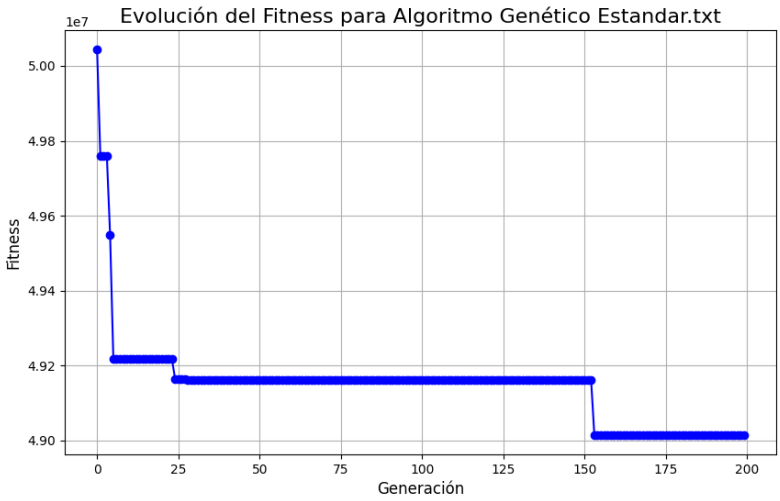


Figura 1: Algoritmo Genético Estandar

Evolución inicial (Generaciones 1-5):

Durante las primeras generaciones, se observa una mejora significativa en el fitness, comenzando desde 50,044,002 y descendiendo rápidamente a 49,549,784. Esto es común en los algoritmos genéticos, ya que al inicio hay más diversidad en la población y mayores oportunidades para explorar mejores soluciones.

Fase de estabilización (Generaciones 6-25):

A partir de la generación 6, el fitness mejora más lentamente. Se estabiliza en torno a 49,216,524 en la generación 7 y permanece constante hasta la generación 25, salvo una pequeña mejora en la generación 25, donde alcanza 49,164,536. Esta fase puede deberse a:

- Exploración limitada: La población podría haber convergido hacia un óptimo local, reduciendo la diversidad genética.
- Falta de presión selectiva: La selección podría estar favoreciendo soluciones similares, dificultando explorar regiones no visitadas del espacio de búsqueda.

Prolongada estancación (Generaciones 26-154):

La solución se mantiene casi constante en 49,161,486 durante un gran número de generaciones (más de 120). Esto indica que el algoritmo no encuentra nuevas soluciones óptimas, probablemente debido a:

- Baja tasa de mutación, lo que dificulta introducir diversidad genética.
- Algoritmo atrapado en un óptimo local.

Mejora tardía (Generación 154):

A partir de esta generación, el fitness desciende nuevamente a 49,014,404, que parece ser el mínimo alcanzado al final. Este comportamiento podría deberse a:

- Un evento de mutación afortunado que introdujo una solución más prometedora.
- Cruzamiento efectivo entre soluciones cercanas al óptimo global.

## Resultados de la variante baldwiniana

El uso de la variante baldwiniana en el algoritmo genético muestra un patrón diferente en términos de mejora del fitness en comparación con la versión estándar.

En el esquema baldwiniano, las mejoras se dan a nivel de aprendizaje del fenotipo (sin modificar el genotipo directamente). Esto significa que los individuos pueden mejorar su desempeño durante su evaluación, pero estas mejoras no se transmiten genéticamente, a menos que sean seleccionadas en las operaciones de cruce o mutación.

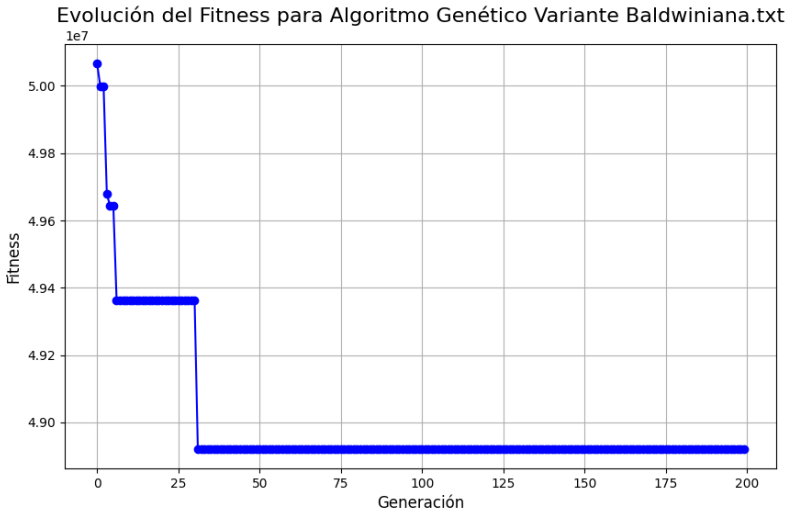


Figura 2: Algoritmo Genético Variante Baldwiniana

Inicio (Generaciones 1-7):

En las primeras generaciones, el fitness mejora notablemente, pasando de 50,066,808 a 49,363,856. Esto indica que el algoritmo encuentra soluciones mejores al principio, lo que es típico en algoritmos genéticos: las primeras iteraciones exploran amplias áreas del espacio de búsqueda y descartan soluciones muy malas.

Estancamiento intermedio (Generaciones 8-32):

Entre estas generaciones, el fitness permanece constante en 49,363,856. Este estancamiento puede deberse a que la población ha alcanzado un equilibrio local, donde la mayoría de los individuos son similares y las variaciones (mutaciones o cruces) no generan cambios significativos.

Mejora adicional (Generación 32):

En la generación 32, el fitness mejora a 48,920,496, lo que indica que alguna mutación o cruce resultó en una solución que escapó del óptimo local. Sin embargo, esta mejora es la última que se observa, ya que el valor de fitness se mantiene constante hasta la generación 200.

## Resultados de la Variante Lamarckiana

La variante lamarckiana en algoritmos genéticos introduce un cambio clave: permite que las mejoras locales en el fitness de un individuo se reflejen directamente en su genotipo. Esto implica que cualquier adaptación que ocurra durante el proceso de mejora local se transmite inmediatamente a la siguiente generación. Esto contrasta con la variante darwiniana, donde los individuos se seleccionan según su fitness, pero no se alteran directamente por adaptaciones locales antes de la reproducción.

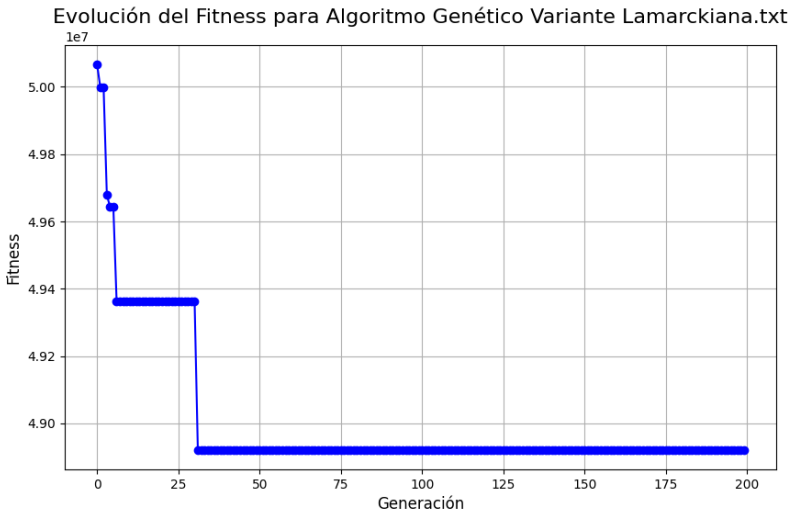


Figura 3: Algoritmo Genético Variante Lamarckiana

Evolución inicial (Generaciones 1-31):

Se observan varias mejoras significativas en el fitness inicial.

- Generación 1: 50,066,808
- Generación 2: 49,980,040
- Generación 4: 49,678,946
- Generación 7: 49,363,856

Esto indica que la estrategia lamarckiana es efectiva para obtener mejoras rápidas al principio, debido a la integración directa de las adaptaciones locales.

Estancamiento (Generaciones 32-200):

Desde la generación 32, el fitness se mantiene constante en 48,920,496. A pesar de tener 168 generaciones adicionales, el algoritmo no puede escapar de este valor.

## Resultados de la Variante Lamarckiana v2

La variante lamarckiana v2 de un algoritmo genético se centra en la combinación del enfoque evolutivo con modificaciones directas al genotipo de los individuos, basándose en los resultados de su desempeño. Este enfoque refleja el principio lamarckiano de la herencia de caracteres adquiridos, donde los ajustes individuales logrados durante la vida de un organismo se transfieren a su descendencia.

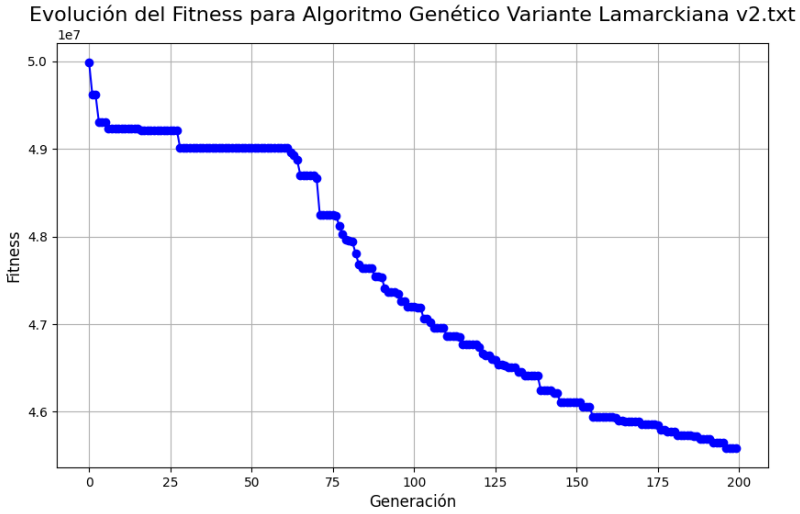


Figura 4: Algoritmo Genético Variante Lamarckiana v2

Progresión del fitness a lo largo de las generaciones

Al observar los valores de fitness, podemos notar una disminución continua en el costo asociado a la mejor solución encontrada a medida que avanza el proceso evolutivo. Esto indica que el algoritmo mejora constantemente las soluciones, lo que es consistente con la naturaleza exploratoria y de explotación de un algoritmo genético. Sin embargo, el progreso no es lineal, hay períodos en los que el fitness permanece constante (por ejemplo, Generación 29 a 63 o Generación 140 a 146), lo cual podría deberse a la convergencia parcial o a un equilibrio

entre exploración y explotación.

Puntos de estancamiento y mejoras significativas

Períodos como entre las generaciones 63 y 80, o entre las generaciones 140 y 146, muestran estancamiento en la mejora. Esto puede deberse a que el algoritmo ha encontrado un óptimo local en estas fases y le resulta más difícil escapar hacia regiones mejores del espacio de soluciones. Las mejoras significativas ocurren en momentos clave, como:

- Generación 29: disminución notable de 49217256 a 49012560.
- Generación 72: salto considerable de 48669440 a 48243140.
- Generación 77 a 79: se pasa de 48241066 a 48029522, lo que denota un ajuste rápido y efectivo de las soluciones.
- Generación 140 a 156: aunque el progreso es más lento, hay una transición constante hacia un menor costo.
- Generación 197 a 200: el algoritmo alcanza finalmente el costo mínimo (45585584), que parece ser el óptimo global dentro del tiempo y configuraciones dadas.

La mejor solución encontrada

La mejor solución tiene un fitness final de 45585584, obtenido en la Generación 200. Esto refleja un alto nivel de optimización, considerando la naturaleza del problema.

Limitaciones observadas

En las últimas generaciones (por ejemplo, después de la generación 150), la tasa de mejora se vuelve muy lenta. Esto sugiere que, si bien el enfoque lamarckiano permite una optimización acelerada en las primeras etapas, puede requerir ajustes adicionales.

## Resultados de la Variante Lamarckiana v3

La variante del algoritmo genético Lamarckiano V3 sigue un esquema de optimización que mejora progresivamente el fitness a lo largo de 273 generaciones. Al observar las métricas, se puede deducir que el algoritmo logra una convergencia, aunque lenta en las últimas etapas, lo que refleja un comportamiento típico de este tipo de estrategias cuando se acerca al óptimo.

El fitness inicial comienza en 49,640,646, y tras 273 generaciones alcanza 44,978,446, logrando una mejora significativa de 4,662,200 unidades en el fitness global. Esto implica que el algoritmo está ajustando consistentemente las soluciones hacia un óptimo en el espacio de búsqueda.

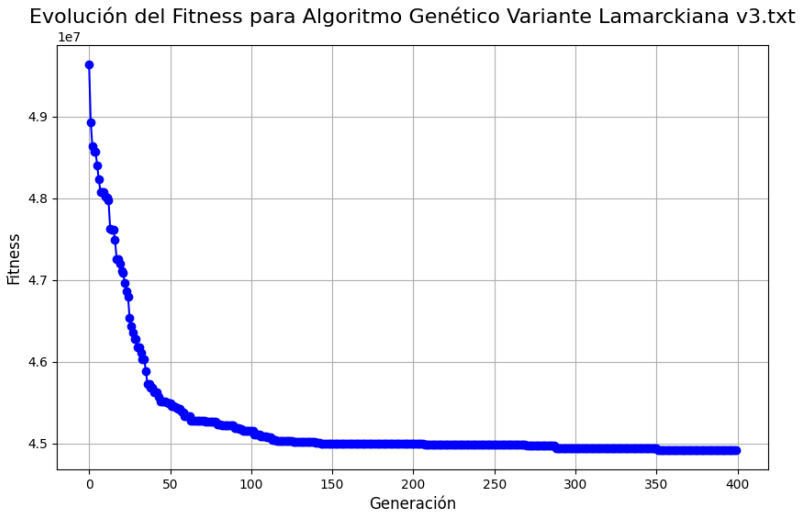


Figura 5: Algoritmo Genético Variante Lamarckiana v3

Fase Inicial (Generaciones 1-20):

Durante esta fase, el fitness mejora rápidamente, con una reducción de 7,583,912 unidades en tan solo 20 generaciones.

Este comportamiento sugiere una alta exploración del espacio de búsqueda, combinada con un balance adecuado entre la explotación de soluciones prometedoras.

La frecuencia de mejora es constante y notable, indicando que los operadores genéticos y las mutaciones están funcionando eficazmente en este rango.

Fase Intermedia (Generaciones 21-150):

La mejora comienza a desacelerarse, con intervalos de estancamiento en el fitness, como se observa entre las generaciones 30-35 y 60-80.

A partir de la generación 80, las mejoras se vuelven más espaciadas, lo que sugiere una exploración más detallada en un vecindario del espacio de soluciones.

La implementación del mecanismo lamarckiano (que adapta los individuos al

entorno a través de aprendizaje local) podría estar impulsando estas pequeñas pero consistentes mejoras en esta fase.

Fase Final (Generaciones 151-273):

A partir de la generación 150, el algoritmo experimenta una convergencia lenta y prolongada.

Las mejoras son mínimas y están separadas por largos intervalos sin cambios en el mejor fitness.

Este comportamiento puede indicar que el algoritmo ha alcanzado una región cercana al óptimo local/global, donde la diversidad genética restante es baja y los operadores genéticos no están explorando soluciones significativamente mejores.

## Resultados de Comparativa de los Algoritmos Genéticos

Los algoritmos evaluados presentan variaciones en su diseño y comportamiento, con diferencias notables en su capacidad para explorar y explotar el espacio de búsqueda, el tiempo de convergencia y la mejora global del fitness. A continuación, se analizan los aspectos clave que distinguen cada variante.

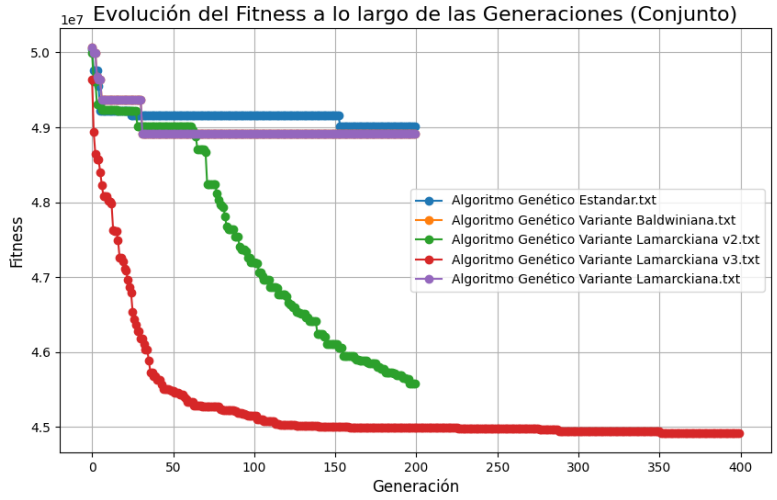


Figura 6: Comparación Algoritmos Genéticos

Algoritmo Genético Estándar

Mejora considerablemente al principio, pero converge rápidamente a un valor alto de fitness ( 4.9e7). Se estabiliza alrededor de las primeras 50 generaciones y mantiene un valor fijo sin más mejoras.

Es el peor de todos los algoritmos comparados, ya que no logra una optimización tan efectiva como las variantes Baldwiniana o Lamarckiana. Es el algoritmo menos eficiente para este problema. Aunque estable, no alcanza soluciones competitivas.

Variante baldwiniana

Presenta una mejora inicial más pronunciada que el estándar, pero su ritmo de mejora disminuye rápidamente después de las primeras generaciones. Se estabiliza más temprano que el estándar con un fitness final ligeramente mejor ( 4.8e7).

La introducción del aprendizaje Baldwiniano permite una mejora, pero no lo suficiente como para destacarse significativamente. Es mejor que el estándar, pero no ofrece una ventaja competitiva considerable respecto a las variantes Lamarckianas.

Variante Lamarckiana

Mejora notablemente más rápido que el estándar, alcanzando un valor cercano a 4.6e7 en pocas generaciones. Logra una estabilización temprana con un fitness más bajo, mostrando mejor capacidad de optimización global.

La estrategia Lamarckiana favorece la adaptación directa en los individuos, lo que permite encontrar mejores soluciones. Conclusión : Representa una mejora considerable respecto al estándar, destacando en rendimiento global.

Variante Lamarckiana v2

Similar a la variante original Lamarckiana, con mejoras rápidas y consistentes en las primeras generaciones. Se estabiliza cerca de 4.55e7, mostrando un equilibrio entre convergencia rápida y fitness final.

Ofrece resultados competitivos, situándose como una de las mejores alternativas para optimización general.

Variante Lamarckiana v3

Es el algoritmo más agresivo en términos de mejora inicial, alcanzando rápidamente un fitness cercano a 4.5e7. Converge a la solución más baja de todas ( 4.5e7) y lo hace de manera rápida, consolidándose como el mejor algoritmo del conjunto.

La versión optimizada del enfoque Lamarckiano maximiza la capacidad de ex-

ploración inicial y explotación posterior, logrando resultados óptimos. Es la variante más efectiva tanto en velocidad como en calidad del fitness final, superando ampliamente al estándar y al resto de variantes.

Comparativa general

Final de Convergencia y Fitness :

- La variante Lamarckiana v3 es la mejor, alcanzando el fitness más bajo de todos ( 4.5e7) con una convergencia rápida y consistente.
- La variante Lamarckiana v2 es una buena segunda opción, equilibrando resultados con estabilidad.
- El algoritmo estándar es el peor, quedándose muy lejos de las soluciones más óptimas ( 4.9e7).

Velocidad de convergencia : La Lamarckiana v3 lidera en velocidad, encontrando soluciones óptimas mucho más rápido que los demás. El algoritmo estándar y la variante Baldwiniana son lentos y no tan efectivos.

Exploración vs Explotación : Las variantes Lamarckianas muestran mejor equilibrio entre explorar nuevas soluciones y explotar las buenas, logrando un fitness significativamente mejor.

# Conclusiones

La práctica realizada permitió analizar y comparar en profundidad el comportamiento de distintos algoritmos genéticos aplicados a un problema de optimización a lo largo de varias generaciones. Los resultados obtenidos evidencian patrones interesantes en la evolución de las soluciones y destacan la importancia del diseño de los algoritmos para maximizar su desempeño.

En general, los algoritmos genéticos mostraron un comportamiento típico caracterizado por una mejora progresiva en el fitness de las soluciones. Inicialmente, se observó una disminución pronunciada en el costo (mejor fitness), reflejando la eficacia de los operadores genéticos en explorar soluciones de mayor calidad en las primeras generaciones. Sin embargo, con el transcurso de las generaciones, el progreso se estabilizó, lo que evidencia una fase de convergencia donde las mejoras son mínimas o inexistentes. Este fenómeno es común en algoritmos genéticos y está relacionado con la explotación intensiva de soluciones locales en detrimento de una exploración más amplia del espacio de búsqueda.

El análisis individual de los algoritmos puso en evidencia diferencias significativas en su rendimiento. Algunos algoritmos presentaron una reducción más

agresiva del costo durante las primeras generaciones, logrando converger rápidamente hacia una solución óptima. Por otro lado, otros algoritmos mostraron un progreso más gradual pero sostenido, alcanzando soluciones finales comparables o incluso mejores. Esto resalta la importancia de seleccionar cuidadosamente los parámetros y operadores genéticos (como las probabilidades de cruce y mutación, o el tamaño de la población) en función de las características del problema.

Un fenómeno recurrente en algunos algoritmos fue la convergencia prematura, donde la población alcanza soluciones subóptimas debido a la pérdida de diversidad genética. Esto impide que el algoritmo explore regiones potencialmente mejores del espacio de búsqueda. Estrategias como el ajuste de la probabilidad de mutación o la incorporación de técnicas que fomenten la diversidad podrían mitigar este problema y mejorar los resultados globales.

Por último, se destaca la importancia de encontrar un equilibrio adecuado entre la exploración y la explotación. La exploración permite al algoritmo descubrir nuevas regiones prometedoras del espacio de búsqueda, mientras que la explotación se centra en refinar las soluciones ya encontradas. Un balance inadecuado entre estas dos fases puede llevar a resultados subóptimos, ya sea por falta de progreso o por un enfoque excesivo en soluciones locales.