Máster Universitario en Ingeniería Informática

de la Universidad de Granada

Práctica de algoritmos evolutivos

Problema de optimización combinatoria

QAP

Inteligencia Computacional (IC)

Autora

Marina Jun Carranza Sánchez

Índice

Índice de figuras

1. Introducción

Esta práctica aborda el problema de la asignación cuadrática (QAP, por sus siglas en inglés) mediante el uso de algoritmos evolutivos. Este problema pertenece a la categoría de optimización combinatoria, con múltiples aplicaciones en ingeniería y computación. El objetivo principal es minimizar los costes de transporte asignando eficientemente n instalaciones a n ubicaciones posibles, considerando distancias y flujos entre ellas.

1.1. Herramientas y entorno

El desarrollo se ha realizado en *Google Colab*, una plataforma que permite trabajar con Python y aprovechar recursos de hardware avanzados de manera gratuita.

Además, se ha optado por utilizar bibliotecas disponibles públicamente como `matplotlib`, `numpy` y `time`.

Esta elección se debe a su acceso gratuito a potentes recursos de computación, la disponibilidad de un entorno preconfigurado y una integración fluida con Google Drive, lo que simplifica el almacenamiento de los datos.

1.2. Preparación de los datos

En esta sección se detalla el proceso de lectura y preparación de los datos almacenados en el archivo `tai256c.dat`, que contiene toda la información necesaria para resolver el problema de asignación cuadrática (QAP).

El archivo `tai256c.dat` sigue un formato estándar para problemas QAP:

- **Primera línea**: contiene un número entero (n) que indica el tamaño del problema, es decir, el número de instalaciones y ubicaciones disponibles. En este caso, $n = 256$.
- **Matrices de datos**: se encuentran dos matrices cuadradas de tamaño 256×256:
  - La primera matriz (*flow*, w): Representa los flujos de materiales entre las instalaciones.
  - La segunda matriz (*distances*, d): Representa las distancias entre las ubicaciones.

```
[2]  import numpy as np
     import random

     # Lectura de datos
     data = np.loadtxt('tai256c.dat', skiprows=1)
     flow = np.int32(data[:256])
     distances = np.int32(data[256:])

     print(flow)
     print(distances)
```

```
[[1 1 1 ... 0 0 0]
 [1 1 1 ... 0 0 0]
 [1 1 1 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
[[     0 100000  25000 ...  10000  20000  50000]
 [100000      0 100000 ...   5882  10000  20000]
 [ 25000 100000      0 ...   3846   5882  10000]
 ...
 [ 10000   5882   3846 ...      0 100000  25000]
 [ 20000  10000   5882 ... 100000      0 100000]
 [ 50000  20000  10000 ...  25000 100000      0]]
```

Figura 1: Preparación y lectura de datos (`tai256c`).

El código anterior (Figura 1) permite realizar la preparación de datos:

1. **Lectura del archivo:** se utiliza la función `np.loadtxt` de la biblioteca NumPy para leer el archivo. Esta función permite ignorar la primera línea (que contiene n) y cargar el resto de los datos en una matriz unidimensional.

   El resultado es una matriz que contiene todos los valores de las dos matrices (flujos y distancias) de forma consecutiva.

2. **Separación de matrices:** dado que ambas matrices están almacenadas de forma contigua, se dividen en dos partes:

   - La primera mitad corresponde a la matriz de flujos (w).
   - La segunda mitad corresponde a la matriz de distancias (d).

   La separación se realiza de manera eficiente usando *slicing*.

3. **Conversión de tipos:** para mejorar la eficiencia en memoria y cálculo, las matrices se convierten a tipo entero (`int32`).

Con este procedimiento, las matrices w (flujos) y d (distancias) quedan listas para su uso en las siguientes etapas de la práctica, pudiendo comprobar las tres primeras y últimas filas de ambas en la Figura 1.

1.3. Metodología

Un algoritmo genético (AG) es una técnica de optimización inspirada en la evolución natural. En este caso, se busca minimizar la función de fitness, utilizando una población inicial generada de manera controlada con una semilla fija. A continuación, se describe la metodología seguida, estructurada en fases:

1. **Inicialización de la población:**

   - La población inicial está formada por 1000 permutaciones de números del 0 al 255, generadas de manera reproducible utilizando una semilla fija (`seed = 2024`):

```python
# Configuración de la semilla
seed = 2024
random.seed(seed)
np.random.seed(seed)

# Generación de la población inicial
population = np.array([np.random.permutation(np.arange(256)) for _ in range(1000)])
```

Figura 2: Definición de la población inicial con una semilla.

2. **Evaluación de la función de fitness:**

   - Cada individuo es evaluado mediante una función de fitness que mide el costo asociado a su solución. La función de fitness, diseñada para minimizar el objetivo, se define tal y como aparece en la siguiente figura:

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

Figura 3: Definición de la función de evaluación *fitness*.

3. **Evolución mediante iteraciones:**

   - Se ejecuta un bucle evolutivo hasta cumplir un criterio de parada (número de generaciones, tiempo máximo, o convergencia). Dentro de cada iteración:

     a) **Selección de padres**: se seleccionan individuos de la población actual para reproducirse, basándose en su fitness. Incluye técnicas como selección por torneo o ranking.

b) **Recombinación o cruce**: se combinan los genes de los padres seleccionados para generar nuevos individuos (descendientes). Técnicas comunes de cruce incluyen cruce de un punto, de dos puntos y PMX.

c) **Mutación**: se aplican modificaciones aleatorias a los descendientes para introducir diversidad genética en la población. La probabilidad de mutación puede ser constante o adaptativa.

d) **Evaluación del fitness**: Los descendientes generados son evaluados utilizando la misma función de fitness.

e) **Selección de supervivientes**: se eligen los individuos que conformarán la población de la siguiente generación. Métodos comunes incluyen elitismo (preservar los mejores individuos) y remplazo generacional completo o parcial.

4. **Criterio de parada:**

   - El proceso iterativo se detiene cuando se cumple el criterio de parada definido (por ejemplo, alcanzar un número máximo de generaciones o encontrar una solución satisfactoria).

Esta metodología permite desarrollar algoritmos genéticos robustos y adaptados al caso de minimizar costos en un espacio de búsqueda de permutaciones.

2. Algoritmo genético estándar (GA-1)

2.1. Fundamentos teóricos

El algoritmo genético estándar (GA-1) emplea diversas técnicas fundamentales inspiradas en procesos evolutivos naturales. A continuación, se describen los fundamentos teóricos de cada una de ellas:

- **Selección por torneo** (Goldberg & Deb, 1991)
  - Este método (ver Figura 4) selecciona padres de una población comparando el fitness de individuos seleccionados aleatoriamente. En cada torneo, el individuo con el mejor fitness (el de menor valor en problemas de minimización) es elegido como padre.
  - La selección por torneo equilibra la explotación de los mejores individuos y la exploración de nuevas combinaciones. Esto se logra ajustando el tamaño del torneo, lo que permite controlar la presión selectiva.
  - **Ventajas:** es simple de implementar, eficiente computacionalmente y robusto frente a problemas de escalado de fitness.
- **Recombinación por segmento aleatorio** (Goldberg, 1989)
  - En este método (ver Figura 4) , se selecciona un segmento aleatorio de genes de uno de los padres y se completa la solución utilizando los genes restantes del segundo padre, manteniendo el orden y la validez de la permutación.
  - Esta técnica es adecuada para problemas de optimización combinatoria, como el flujo de trabajo o el problema del viajante.
  - **Ventajas:** garantiza la validez de las soluciones generadas y preserva estructuras importantes de los padres.
- **Mutación por intercambio** (Michalewicz, 1996)
  - La mutación introduce diversidad genética intercambiando dos posiciones aleatorias en un individuo. Esto ayuda a explorar nuevas regiones del espacio de búsqueda y a evitar el estancamiento en óptimos locales (ver Figura 4).
  - **Ventajas:** es sencillo de implementar y muy eficaz en representaciones de permutaciones donde el orden de los elementos es crítico.

2.2. Implementación

```python
import numpy as np

# Selección por torneo
def tournament_selection(population, fitness_vals):
    selected = []
    for _ in range(len(population)):
        i, j = np.random.choice(len(population), 2, replace=False)
        selected.append(population[i] if fitness_vals[i] < fitness_vals[j] else population[j])
    return np.array(selected)

# Cruce: Intercambio aleatorio de un segmento
def crossover(parent1, parent2):
    size = len(parent1)
    start, end = sorted(np.random.choice(size, 2, replace=False))
    offspring = -np.ones(size, dtype=int)
    offspring[start:end+1] = parent1[start:end+1]
    for val in parent2:
        if val not in offspring:
            offspring[np.where(offspring == -1)[0][0]] = val
    return offspring

# Mutación: Intercambio de dos posiciones
def mutate(individual):
    i, j = np.random.choice(len(individual), 2, replace=False)
    individual[i], individual[j] = individual[j], individual[i]
    return individual
```

Figura 4: Funciones de selección por torneo, crossover y mutate.

El algoritmo genético estándar (GA-1) se implementa en un flujo estructurado que comprende las siguientes etapas principales:

1. **Inicialización**: se genera una población inicial de permutaciones de forma aleatoria, asegurando la diversidad inicial en el espacio de búsqueda.

2. **Evaluación inicial**: cada individuo de la población es evaluado utilizando la función de fitness, que calcula el costo total asociado a la permutación considerando las matrices `flow` y `distances`.

3. **Bucle evolutivo**: Durante cada generación:

   - **Selección de padres:** se utiliza selección por torneo para elegir los individuos que participarán en la reproducción. Este enfoque asegura que los individuos con mejor fitness tienen una mayor probabilidad de ser seleccionados.
   - **Recombinación**: a partir de los padres seleccionados, se generan descendientes mediante un cruce de segmento aleatorio. Este proceso combina características de ambos padres mientras preserva la validez de las permutaciones.
   - **Mutación**: con una probabilidad definida, se realiza una mutación sobre los descendientes, intercambiando dos posiciones en la permutación para aumentar la diversidad genética y evitar el estancamiento en óptimos locales.
   - **Evaluación de fitness**: los descendientes generados son evaluados nuevamente con la función de fitness.

- **Actualización de la población**: la población de la siguiente generación se forma reemplazando completamente a la actual con los descendientes generados.

4. **Criterio de parada**: el proceso iterativo se detiene tras un número predefinido de generaciones, en este caso 100. Al finalizar, se selecciona el mejor individuo de la población final como la solución óptima.

```python
# Algoritmo genético
def genetic_algorithm(flow, distances, population, generations=100, crossover_rate=0.8, mutation_rate=0.1):
    best_fitness_per_gen = []

    for gen in range(generations):
        print(f"Generación {gen + 1}/{generations}: Evaluando fitness...")
        fitness_vals = fitness(population, flow, distances)
        best_fitness = np.min(fitness_vals)
        best_fitness_per_gen.append(best_fitness)
        print(f"Generación {gen + 1}: Mejor fitness = {best_fitness}")

        # Selección por torneo
        print(f"Generación {gen + 1}: Realizando selección por torneo...")
        population = tournament_selection(population, fitness_vals)
        next_generation = []

        for i in range(0, len(population), 2):
            if i + 1 >= len(population):
                break
            if np.random.rand() < crossover_rate:
                offspring1 = crossover(population[i], population[i + 1])
                offspring2 = crossover(population[i + 1], population[i])
            else:
                offspring1, offspring2 = population[i], population[i + 1]
            if np.random.rand() < mutation_rate:
                offspring1 = mutate(offspring1)
            if np.random.rand() < mutation_rate:
                offspring2 = mutate(offspring2)
            next_generation.extend([offspring1, offspring2])

        population = np.array(next_generation)
```

Figura 5: Función de GA-1.

Parámetros:

El algoritmo genético estándar (GA-1) fue ejecutado con los siguientes parámetros: una población inicial de 1000 individuos, un total de 100 generaciones, una probabilidad de cruce del 80 % y una probabilidad de mutación del 10 %.

2.3. Análisis de resultados

En los resultados de la ejecución, el mejor costo asociado al finalizar las 100 generaciones fue de **49406340**. Este valor corresponde a la solución óptima encontrada por el algoritmo, representada por una permutación específica (ver Figura 6). El tiempo total de ejecución fue de aproximadamente **3 min y 30 segundos**, lo que refleja un desempeño eficiente en términos computacionales considerando la complejidad del problema y el tamaño de la población (1000 individuos).

La evaluación del mejor fitness alcanzado confirma que el algoritmo fue capaz de identificar una solución medianamente decente teniendo en cuenta el tiempo empleado. Sin embargo, las grandes

fluctuaciones en el fitness indican que sería conveniente implementar técnicas más avanzadas, como la optimización local y otros métodos de algoritmos genéticos.

```
Generación 100/100: Evaluando fitness...
Generación 100: Mejor fitness = 49406340
Generación 100: Realizando selección por torneo...
Tiempo de ejecución: 197.92 segundos
Mejor permutación encontrada:
[ 12 145 208  72 229 133 173 204 135  53 105 217 211 241 152 182 215  15
 103  68  48 174 219 237 190  34 223 234 160 164  86 131  10  83 236 113
 169  94   1  40  91  30 178 143  17  36  77  81  73 243 124  95 100  44
  21 115  62 216 199 144 147 196 197  49  78 137 151 210 188  24   5  88
  38 187 166 125 249 205 171 248   6  98 158 123  71 179 109  23  43 186
 162  14 150   0  66 127 231 140  58 146 138 149 185  79 118 200 128 198
  70  25  28  29 251  89 209  67  35 130  99  19 139   8 193 168 120  39
 126 121  33 202 226  90  84 148   9 225 232 157  41 213  63 175  50 247
 114  52 246  59 218 153 134 170 122  65  64  80 220 191 240  85 181 172
 224 201 235 227 254  57  76 104 154 221  32  92 106 230 183 101  37 111
 142   2  20  26  27  11 167  97 141   7 116 253 233  45 165 222 255 102
 159 136  75 176  47 189 180  18 155  46 108 163 206  54 252 239 112 161
 177  16  87 212  42 238 192   4 228 156 242 110 207 194 244 132  56  93
  96 195 250 119 184  60  31  22  55  51 129   3 214  74 107 117 245  13
  69  82 203  61]
Costo asociado:
49406340
```

Figura 6: Resultados de GA-1.

La gráfica de la Figura 7 muestra cómo evoluciona el fitness a lo largo de las 100 generaciones del algoritmo genético estándar (GA-1). Durante las primeras generaciones, se observa un comportamiento fluctuante con valores de fitness que oscilan significativamente. Esto indica que el algoritmo está explorando activamente el espacio de búsqueda en esta etapa inicial, favoreciendo la diversidad genética en la población.

A partir de la mitad de las generaciones, se observa un descenso en el coste. Esto sugiere que el algoritmo comienza a explotar regiones más prometedoras del espacio de búsqueda. Sin embargo, incluso en las últimas generaciones, las oscilaciones siguen siendo evidentes, lo que indica que una alta variabilidad en la calidad de las soluciones generadas.

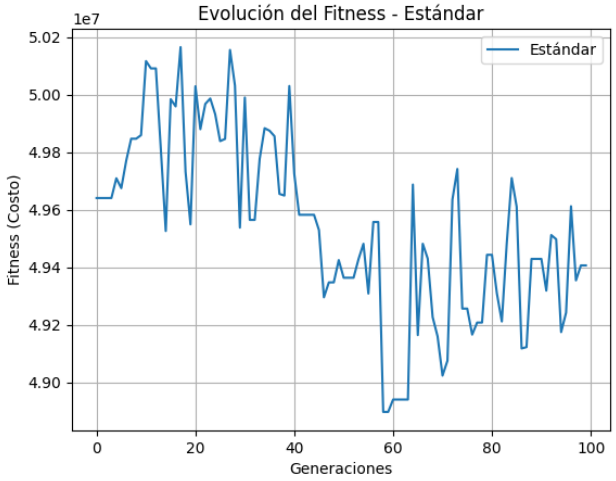

Figura 7: Gráfica de evolución del fitness de GA-1.

3. Variante baldwiniana (GA-2)

3.1. Fundamentos teóricos

El algoritmo genético con variante baldwiniana (GA-2) introduce un concepto adicional al proceso evolutivo estándar: la optimización local. Este enfoque, inspirado en la variante baldwiniana de la evolución, incorpora técnicas de búsqueda local en el cálculo del fitness. Sin embargo, a diferencia de la variante lamarckiana, los descendientes no heredan las mejoras adquiridas por sus padres mediante optimización local, ya que el material genético utilizado para formar descendientes proviene de los individuos originales.

El propósito de este enfoque es enriquecer el proceso de selección utilizando soluciones mejoradas mediante búsqueda local para evaluar el fitness, sin alterar directamente el espacio genético explorado. Esto permite que el algoritmo mantenga la diversidad en la población, al tiempo que mejora la precisión del fitness asociado a cada individuo.

En este caso, se utiliza la técnica de *hill climbing* (ascenso de colinas) para optimizar localmente los individuos seleccionados antes de evaluar su fitness. Esta técnica garantiza que cada individuo parte de su solución inicial y mejora de forma iterativa hasta alcanzar un óptimo local en su vecindad (Hinton & Nowlan, 1987).

3.2. Implementación

El GA-2 mantiene los pasos generales del algoritmo genético estándar descritos en GA-1, pero introduce las siguientes modificaciones clave:

- **Optimización local**: antes de evaluar el fitness, los primeros 500 individuos de cada generación se mejoran utilizando la técnica de *hill climbing*. Esta técnica se implementa como sigue:

```python
# Función de optimización local: Hill Climbing
def hill_climbing(individual, flow, distances, max_iters=100):
    best_individual = individual.copy()
    best_cost = fitness(best_individual[np.newaxis, :], flow, distances)[0]

    for _ in range(max_iters):
        # Generar un vecino intercambiando dos posiciones
        i, j = np.random.choice(len(individual), 2, replace=False)
        neighbor = best_individual.copy()
        neighbor[i], neighbor[j] = neighbor[j], neighbor[i]

        # Calcular el costo del vecino
        neighbor_cost = fitness(neighbor[np.newaxis, :], flow, distances)[0]

        # Si el vecino es mejor, actualizar
        if neighbor_cost < best_cost:
            best_individual, best_cost = neighbor, neighbor_cost

    return best_individual, best_cost
```

Figura 8: Función de *hill climbing*.

- **Evaluación del fitness**: la evaluación del fitness utiliza los resultados de la optimización local, pero la población se actualiza con los individuos originales, no con los optimizados. Esto asegura que las mejoras no se transmitan genéticamente.
- **Proceso de selección**: similar a GA-1, se utiliza selección por torneo para elegir individuos que se reproducirán en la siguiente generación.
- **Parámetros ajustados**: en comparación con GA-1, el GA-2 utiliza una mayor probabilidad de cruce (**90 %**) y una mayor probabilidad de mutación (**20 %**) para incrementar la diversidad en las generaciones posteriores.

El GA-2 equilibra la explotación local mediante optimización con la exploración global, permitiendo que las soluciones mejoradas influyan en el proceso evolutivo sin comprometer la diversidad genética.

```python
# Algoritmo genético con variante baldwiniana
def genetic_algorithm_baldwinian(flow, distances, population, generations=100, crossover_rate=0.9, mutation_rate=0.2):
    best_fitness_per_gen = []

    for gen in range(generations):
        print(f"Generación {gen + 1}/{generations}: Evaluando fitness con optimización local...")
        fitness_vals = []
        optimized_individuals = []

        for idx, individual in enumerate(population[:500]):
            if idx % 100 == 0:
                print(f"  Procesando individuo {idx + 1}/500...")
            optimized_individual, local_cost = hill_climbing(individual, flow, distances)
            fitness_vals.append(local_cost)
            optimized_individuals.append(individual)

        fitness_vals = np.array(fitness_vals)
        population = np.array(optimized_individuals)
        best_fitness_per_gen.append(np.min(fitness_vals))

        print(f"Generación {gen + 1}: Mejor fitness hasta ahora = {np.min(fitness_vals)}")

        print(f"Generación {gen + 1}: Realizando selección por torneo...")
        population = tournament_selection(population, fitness_vals)
        next_generation = []

        for i in range(0, len(population), 2):
            if i + 1 >= len(population):
                break
            if np.random.rand() < crossover_rate:
                offspring1 = crossover(population[i], population[i + 1])
                offspring2 = crossover(population[i + 1], population[i])
            else:
                offspring1, offspring2 = population[i], population[i + 1]
            if np.random.rand() < mutation_rate:
                offspring1 = mutate(offspring1)
            if np.random.rand() < mutation_rate:
                offspring2 = mutate(offspring2)
            next_generation.extend([offspring1, offspring2])

        population = np.array(next_generation[:1000])

    best_idx = np.argmin(fitness(population, flow, distances))
    return population[best_idx], fitness(population[best_idx:best_idx+1], flow, distances)[0], best_fitness_per_gen
```

Figura 9: Función de GA-2.

3.3. Análisis de resultados

En los resultados de ejecución, el GA-2 alcanzó un costo asociado final de **49962266**. El tiempo total de ejecución fue significativamente mayor (aproximadamente **23 minutos**) debido a la incorporación de la optimización local, que añade un paso computacionalmente intensivo para los 500 mejores individuos de cada generación.

Aunque el GA-2 supera al GA-1 en términos de calidad de la solución alcanzada, el incremento en el tiempo de ejecución puede ser una limitación en problemas donde el tiempo de cómputo es crítico. Esto refleja un balance entre calidad de la solución y tiempo computacional que debe ser considerado según las necesidades del problema.

```
Generación 50/50: Evaluando fitness con optimización local...
  Procesando individuo 1/500...
  Procesando individuo 101/500...
  Procesando individuo 201/500...
  Procesando individuo 301/500...
  Procesando individuo 401/500...
Generación 50: Mejor fitness hasta ahora = 48096944
Generación 50: Realizando selección por torneo...
Tiempo de ejecución: 1355.08 segundos
Mejor permutación encontrada:
[186  12 123 224 189  40 139 106 131  83  62  10 110  35 125  21 183  41
 113 226  86 107 149  99 112 252 170 184 179  56 177  43 220  16  23  34
  94 198 155 157  72 223 221 201 108 133   1 100 233 213 160  74  68   4
  71 169 188  44 203  89 250  53  79 197 248 216 105 150 211 254 209 126
 145 135  25  77  33 153  31   7  26  27 120  97 240  39  65 162 148  45
 242 230 124  15  11 136 192 147 222 117  93 191 207  42 119 134  59 138
  88   5 122 237 245 166   8 172  48  60 238   6 137 140 243 167 128 196
 121 218  63 231 235  90 232  55 130 214 142 187 185  50 174  37 152  17
  61 175  96  38 159 144  91 176  57 253 190 200   0  20 212  24  76 181
 143  84 199 154 132 171  32 193 247 215  92  82 103 161 195 101 114  14
 104 225 229 129 158  80 249   2 217 111 151 244 116 241  49 227  66 246
 109  58  22 164 239 236  46  78  85 205 165 168  95   9  52 118 178 251
  18  73 141 180   3  98  19 163  75  51 234  69 210  70  64 228 219  54
 127 115  29 182 206  13 255 146 194  30  36  87 156 202 204 208  67 102
  28  81 173  47]
Costo asociado:
49962266
```

Figura 10: Resultados de GA-2.

La gráfica de la Figura 11 muestra la evolución del fitness a lo largo de las 50 generaciones del GA-2. En las primeras generaciones, se observa una disminución del fitness, con fluctuaciones en las generaciones iniciales y medias. Esto sugiere que la combinación de optimización local con el proceso evolutivo permite al algoritmo explorar el espacio de búsqueda de manera más eficiente en estas etapas. Sin embargo, las fluctuaciones persisten incluso en las generaciones finales, indicando que, aunque el algoritmo converge hacia soluciones de mejor calidad, no logra una estabilización completa.

En comparación con el primero, GA-2 contempla una menor cantidad de generaciones (50) pero logra conseguir un mejor valor de fitness final. Esto sugiere que la incorporación de la optimización local acelera el proceso de convergencia hacia soluciones más cercanas al óptimo.

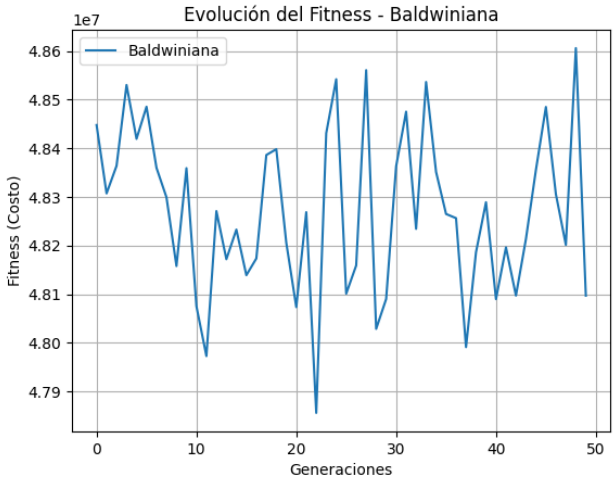

Figura 11: Gráfica de evolución del fitness de GA-2.

En comparación con el algoritmo anterior, GA-2 logró un mejor fitness (**48096944** frente a **49406340** del GA-1) en menos generaciones, gracias a la optimización local que potencia la búsqueda del óptimo. Sin embargo, el tiempo de ejecución fue significativamente mayor (unos **23 minutos** frente a los **3:30** anteriores), reflejando el costo computacional adicional de este enfoque. Además, el GA-2 mostró mayor estabilidad en el fitness hacia las últimas generaciones, lo que indica una convergencia más dirigida.

En conclusión, el GA-2 mejora la calidad de la solución respecto al GA-1, sacrificando el tiempo de ejecución debido a la optimización local. Este enfoque es ideal para problemas donde la calidad es prioritaria, aunque en escenarios con restricciones temporales podrían considerarse variantes más eficientes.

4. Variante lamarckiana (GA-3)

4.1. Fundamentos teóricos

El algoritmo genético con variante lamarckiana (GA-3) se basa en la idea de que los **descendientes pueden heredar directamente las mejoras** adquiridas por sus padres a través de procesos de optimización local. Este enfoque, inspirado en el lamarckismo, asume que los cambios adquiridos en el fitness de un individuo, gracias a procesos de aprendizaje o adaptación local, modifican su material genético, y estos cambios son transmitidos a la siguiente generación.

En comparación con la variante baldwiniana (GA-2), donde la optimización local no afecta directamente el material genético de los descendientes, el GA-3 actualiza la población con las soluciones optimizadas, lo que permite una convergencia más rápida hacia regiones de alto rendimiento en el espacio de búsqueda. Sin embargo, esta técnica puede reducir la diversidad genética más rápidamente, incrementando el riesgo de estancamiento en óptimos locales.

Este enfoque está respaldado por el trabajo de Ackley y Littman (1987), quienes destacaron cómo los métodos de optimización local pueden mejorar directamente la calidad de las soluciones genéticas cuando se integran con técnicas evolutivas (Ackley & Littman, 1987).

4.2. Implementación

El GA-3 introduce las siguientes diferencias clave respecto al GA-2:

- **Optimización local heredable**: cada individuo de la población es optimizado mediante *hill climbing*, y los resultados de esta optimización reemplazan a los individuos originales en la población. Esto asegura que las mejoras obtenidas localmente sean heredadas por los descendientes.

- **Procesamiento completo de la población**: a diferencia de GA-2, donde solo un subconjunto (500 individuos) era sometido a optimización local, en GA-3 se optimiza toda la población en cada generación, mejorando significativamente la calidad general de las soluciones, aunque a costa de un mayor tiempo de ejecución.
- **Menor riesgo de ruido**: debido a la herencia directa de las mejoras, los valores de fitness reflejan de manera más precisa la calidad de las soluciones optimizadas.

El flujo del GA-3 prioriza la explotación de soluciones locales óptimas en cada generación, lo que acelera la convergencia hacia soluciones de alta calidad. Sin embargo, este enfoque requiere un balance cuidadoso para evitar la pérdida de diversidad genética y posibles estancamientos.

```python
# Algoritmo genético con variante lamarckiana
def genetic_algorithm_lamarckian(flow, distances, population, generations=100, crossover_rate=0.9, mutation_rate=0.2):
    best_fitness_per_gen = []

    for gen in range(generations):
        print(f"Generación {gen + 1}/{generations}: Evaluando fitness con optimización local...")
        fitness_vals = []
        optimized_population = []

        for individual in population:
            optimized_individual, local_cost = hill_climbing(individual, flow, distances)
            fitness_vals.append(local_cost)
            optimized_population.append(optimized_individual)

        fitness_vals = np.array(fitness_vals)
        population = np.array(optimized_population)
        best_fitness_per_gen.append(np.min(fitness_vals))

        print(f"Generación {gen + 1}: Mejor fitness hasta ahora = {np.min(fitness_vals)}")

        print(f"Generación {gen + 1}: Realizando selección por torneo...")
        population = tournament_selection(population, fitness_vals)
        next_generation = []

        for i in range(0, len(population), 2):
            if i + 1 >= len(population):
                break
            if np.random.rand() < crossover_rate:
                offspring1 = crossover(population[i], population[i + 1])
                offspring2 = crossover(population[i + 1], population[i])
            else:
                offspring1, offspring2 = population[i], population[i + 1]
            if np.random.rand() < mutation_rate:
                offspring1 = mutate(offspring1)
            if np.random.rand() < mutation_rate:
                offspring2 = mutate(offspring2)
            next_generation.extend([offspring1, offspring2])

        population = np.array(next_generation[:1000])

    best_idx = np.argmin(fitness(population, flow, distances))
    return population[best_idx], fitness(population[best_idx:best_idx+1], flow, distances)[0], best_fitness_per_gen
```

Figura 12: Función de GA-3.

4.3. Análisis de resultados

Al finalizar las 100 generaciones, el GA-3 obtuvo un costo final asociado de **46902326**. Este resultado supera a las dos variantes implementadas previamente, lo que confirma la efectividad de permitir la herencia directa de mejoras locales.

El tiempo de ejecución total fue de aproximadamente **1.49 horas**, significativamente mayor que en las variantes anteriores (GA-1 y GA-2). Este incremento en el tiempo computacional es una

consecuencia directa de optimizar toda la población mediante *hill climbing* en cada generación, lo que eleva el costo computacional pero garantiza un ajuste constante hacia mejores soluciones.

```
Generación 100/100: Evaluando fitness con optimización local...
Generación 100: Mejor fitness hasta ahora = 46568244
Generación 100: Realizando selección por torneo...
Tiempo de ejecución: 5372.15 segundos
Mejor permutación encontrada:
[ 53  30  40 202 178  32  65 119 229 185   4  13  11 174  77 141 191 197
 143 209  97 148   0 115  41  26 127  85 173  60 205  89   2 161  95 232
 102 145 150 147 226  38 219  55 130 235 255  84 211 135 222  67  33  52
 105  50 243 179  35 208 192 124  28 167  22 214 113  87 250 247 168  75
 107  63 109  73 200  78 241 187 246  58 156   8 253  92 216 149 138 137
 117  45 228 144   9  54  83 136  24  57 190 152 154  64  34 166 231 237
 220  25 218 245 121 212   1 120 171 104 217 176  81  79 195 162 224  44
  49 252 155 170 112 236  74  69 234  29  27  23 160 126  19  62  91 132
 158 181 134  18  86 196 210 163 248 188 146 111  68 251 157 240  16 213
  31  20 139 201 184 225 233 142 118  98 172  15  93 203 128 206  56  71
 215 110  17 180  90 122 100 254  43 175 182 177 106 194 123  21 189 249
 227  72  14 186 129 207 125  70  61 103 199 153  59  36  47 133  80  66
 159 230   6 116   5  48 140 223  39  51 101   3  96 114 221 183  46 193
  10  88 151  12 108 238  82 169 164  76  94 239  37 198  42 244  99 204
 131   7 165 242]
Costo asociado:
46902326
```

Figura 13: Resultados de GA-3.

La gráfica muestra que el GA-3 tiene una evolución del fitness marcada por una fuerte disminución inicial durante las primeras generaciones. Esto se debe a la herencia directa de las mejoras obtenidas mediante optimización local, lo que permite un rápido ajuste hacia soluciones de mejor calidad. Sin embargo, a partir de este descenso inicial se observa que las fluctuaciones se mantienen relativamente constantes alrededor de valores cercanos al óptimo encontrado.

En comparación con las variantes anteriores, la lamarckiana converge de manera más rápida y consistente hacia soluciones de alta calidad, aunque con fluctuaciones más leves en generaciones avanzadas. Esto sugiere que la reducción de diversidad genética es compensada por el impacto positivo de las mejoras heredadas.

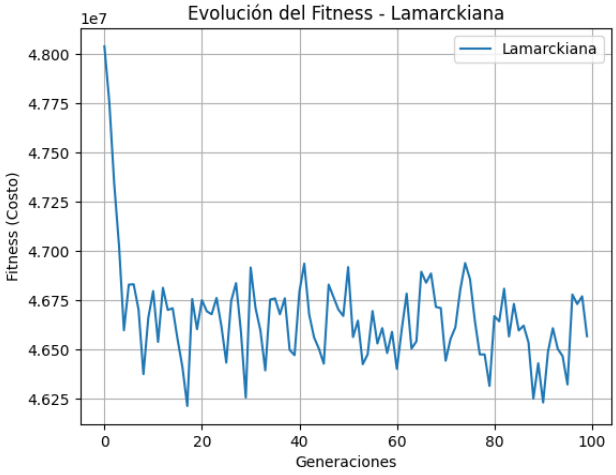

Figura 14: Gráfica de evolución del fitness de GA-3.

En comparación con GA-2, el GA-3 presenta ventajas claras en la calidad de las soluciones alcanzadas. Sin embargo, este beneficio viene acompañado de un incremento sustancial en el tiempo de ejecución (casi 4 veces más). Adicionalmente, el GA-3 muestra una mayor estabilidad, con menores fluctuaciones en comparación con GA-2, gracias a la transmisión directa de las mejoras genéticas.

En resumen, esta variante resulta más efectiva que la baldwidiana en términos de calidad de las soluciones, aunque a costa de un tiempo computacional bastante mayor. Por ello, es necesario combinar la optimización lamarckiana con técnicas que reduzcan el costo computacional.

5. Mejora de la variante lamarckiana (GA-4)

5.1. Fundamentos teóricos

El algoritmo genético con variante lamarckiana mejorada (GA-4) incorpora ajustes y mejoras específicas para abordar las limitaciones observadas en el GA-3. Aunque mantiene el principio básico del lamarckismo, donde las mejoras locales son heredadas directamente por los descendientes, introduce técnicas adicionales para equilibrar la explotación de soluciones óptimas con la exploración del espacio de búsqueda.

Una de las principales mejoras del GA-4 es la inclusión de la **selección basada en ranking**, que prioriza individuos con mejor fitness sin eliminar completamente la diversidad genética, evitando que la población se concentre excesivamente en un único óptimo (Baker, 1985). Además, se utiliza un operador de cruce más avanzado, conocido como ***Order Crossover (OX)***, que asegura una mejor preservación de la estructura genética de los padres en los descendientes (Syswerda, 1991). Estas modificaciones buscan optimizar la convergencia y la calidad de las soluciones, especialmente en problemas de alta complejidad.

Por otro lado, el GA-4 emplea un ***hill climbing*** **más exhaustivo** con un mayor número de iteraciones (**150**) durante la optimización local, lo que permite explorar más profundamente las vecindades de los individuos y encontrar mejores soluciones locales. Esto lo convierte en una versión más robusta para problemas donde la calidad de la solución es crítica Goldberg, 1989.

5.2. Implementación

```python
# Selección basada en ranking
def rank_selection(population, fitness_vals):
    sorted_indices = np.argsort(fitness_vals)
    probabilities = np.linspace(1, 2, len(fitness_vals))[::-1]
    probabilities /= probabilities.sum()
    selected_indices = np.random.choice(sorted_indices, size=len(population), p=probabilities, replace=True)
    return population[selected_indices]

# Operador de cruce mejorado (Order Crossover - OX)
def order_crossover(parent1, parent2):
    size = len(parent1)
    start, end = sorted(np.random.choice(size, 2, replace=False))
    offspring = -np.ones(size, dtype=int)
    offspring[start:end+1] = parent1[start:end+1]
    pointer = 0

    for gene in parent2:
        if gene not in offspring:
            while offspring[pointer] != -1:
                pointer += 1
            offspring[pointer] = gene

    return offspring
```

Figura 15: Función de selección por ranking y *Ordered Crossover*

Las diferencias principales del GA-4 respecto a la variante lamarckiana inicial son las siguientes:

- **Selección basada en ranking**: a diferencia de la selección por torneo utilizada en los anteriores, el GA-4 implementa un esquema de selección basado en ranking. Este método asigna mayores probabilidades de selección a los individuos con mejor fitness, reduciendo el sesgo extremo y promoviendo una distribución más balanceada de la reproducción (ver Figura 15).
- **Cruce mejorado**: el operador de cruce utilizado en el GA-4 es *Order Crossover* (OX), que garantiza que los descendientes hereden la secuencia relativa de genes de los padres, preservando mejor las características estructurales que podrían ser beneficiosas (ver Figura 15).
- **Optimización local más exhaustiva**: se aumenta el número de iteraciones en el *hill climbing* a 150, en comparación con las 100 iteraciones del GA-3. Esto permite una exploración más profunda de las vecindades locales, mejorando la calidad de las soluciones en cada generación.
- **Control de la diversidad genética**: aunque el GA-4 optimiza toda la población, la inclusión de selección por ranking ayuda a preservar cierto nivel de diversidad genética, mitigando el riesgo de estancamiento en óptimos locales.
- **Parámetros ajustados**: el GA-4 utiliza una mayor probabilidad de mutación (**30 %**) para fomentar la exploración, especialmente en generaciones avanzadas, equilibrando la explotación intensiva del espacio de búsqueda promovida por el lamarckismo.

```python
# Algoritmo genético con variante lamarckiana mejorada
def genetic_algorithm_lamarckian_v2(flow, distances, population, generations=50, crossover_rate=0.9, mutation_rate=0.3):
    best_fitness_per_gen = []

    for gen in range(generations):
        print(f"Generación {gen + 1}/{generations}: Evaluando fitness con optimización local...")
        fitness_vals = []
        optimized_population = []

        for individual in population:
            optimized_individual, local_cost = hill_climbing_2(individual, flow, distances)
            fitness_vals.append(local_cost)
            optimized_population.append(optimized_individual)

        fitness_vals = np.array(fitness_vals)
        population = np.array(optimized_population)
        best_fitness_per_gen.append(np.min(fitness_vals))

        print(f"Generación {gen + 1}: Mejor fitness hasta ahora = {np.min(fitness_vals)}")

        print(f"Generación {gen + 1}: Realizando selección basada en ranking...")
        population = rank_selection(population, fitness_vals)
        next_generation = []

        for i in range(0, len(population), 2):
            if i + 1 >= len(population):
                break
            if np.random.rand() < crossover_rate:
                offspring1 = order_crossover(population[i], population[i + 1])
                offspring2 = order_crossover(population[i + 1], population[i])
            else:
                offspring1, offspring2 = population[i], population[i + 1]
            if np.random.rand() < mutation_rate:
                offspring1 = mutate(offspring1)
            if np.random.rand() < mutation_rate:
                offspring2 = mutate(offspring2)
            next_generation.extend([offspring1, offspring2])

        population = np.array(next_generation[:1000])

    best_idx = np.argmin(fitness(population, flow, distances))
    return population[best_idx], fitness(population[best_idx:best_idx+1], flow, distances)[0], best_fitness_per_gen
```

Figura 16: Función de GA-4.

Estas mejoras hacen que el GA-4 sea más eficiente y efectivo para resolver problemas complejos, aunque su tiempo de ejecución podría ser mayor debido a la mayor cantidad de cómputo en la optimización local y los operadores avanzados.

5.3. Análisis de resultados

El GA-4 alcanzó un costo asociado final de **46451920**, lo que representa una mejora respecto a los anteriores. Además, el GA-4 logró esta mejora en un tiempo de ejecución significativamente menor (3945.57 segundos, aproximadamente **1.1 horas**) en comparación con el tiempo empleado por el GA-3 (1.49 horas).

Este ahorro de tiempo computacional es una consecuencia directa de la optimización de las técnicas utilizadas, como el aumento de eficiencia en el proceso de selección y cruce, y una optimización local ajustada. Sin embargo, el tiempo sigue siendo considerablemente mayor que en variantes anteriores (GA-1 y GA-2), lo que se justifica por la calidad de las soluciones obtenidas.

```
Generación 50/50: Evaluando fitness con optimización local...
Generación 50: Mejor fitness hasta ahora = 46360134
Generación 50: Realizando selección basada en ranking...
Tiempo de ejecución: 3945.57 segundos
Mejor permutación encontrada:
[218  20 145 155 205 153 242 209  27 194 238   6 143 192  23 138 232 141
  49 124 187 105 253 164 112  38 148 172  73 223 122  18 201  71 104 228
 119 126   8  99 108  25  79 111 196 177 234  59 199 220 227  14  80   1
  47 160  82 151  90 113 162 131 102  45  93  51  62 182  56  85  53 251
 231 204   4 213  16 224  28  65 245  54  15 117 174  57 181 134  35  76
 184  67 252  86 120 250  48 173 233 202 166 200  26  12 128 101 158 168
 193 255  96 167 235  30 247  66  21 123 210  50  31  29 106 115 127  95
 176  98  81   2 161 230 203 149  61 144 211  72 189 147  92 165 163  41
 103 243 152 107 198 236  13  87 159  63  24 185  32 222 246 229  22  46
   9 169 239  75 248  34  58  36 237 118  33 191 214 217 129 142  91 195
 156 175 146  77 170  68 244 215  52 132  64  60  70  88 121 130  40 212
 180  39 110  10 139  69  78  89 221 190 197 116 241 179 157  84 133 216
  55 188 135 225 125 208 206 109 178  37   5 249   0 183 154 171 100  74
   7 114 254  94 226 136  19  43   3 186  44 240 140 150  97 219  83  42
  11 207  17 137]
Costo asociado:
46451920
```

Figura 17: Resultados de GA-4.

La gráfica muestra una evolución del fitness con una rápida mejora en las primeras generaciones, seguida de fluctuaciones moderadas alrededor de valores óptimos en las generaciones finales. Este comportamiento, muy similar al observado en GA-3, refleja la efectividad del método lamarckiano para explotar regiones de alto rendimiento en el espacio de búsqueda, al tiempo que mantiene un nivel aceptable de diversidad genética gracias al uso del operador de cruce *Order Crossover* (OX) y la selección basada en ranking, que introduce una presión selectiva más controlada.

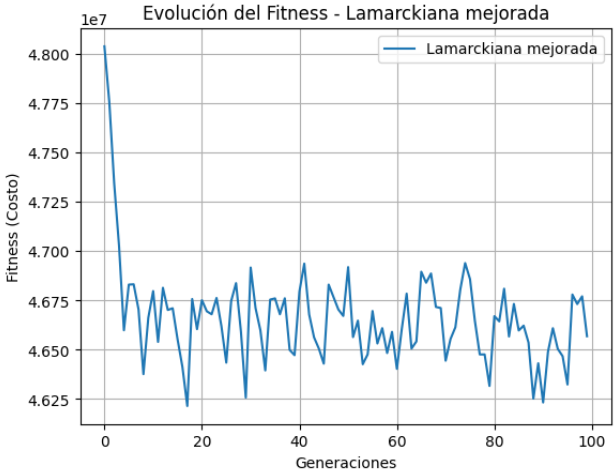

Figura 18: Gráfica de evolución del fitness de GA-4.

GA-4 logró superar a la mejor solución previa (GA-3), con una reducción del tiempo de ejecución del **26.5 %**, destacándose por la eficacia de técnicas avanzadas como la selección basada en ranking y el operador de cruce OX.

En conclusión, el GA-4 mejora tanto la calidad de las soluciones como la eficiencia computacional de GA-3. Estas mejoras lo convierten en una variante altamente efectiva, aunque el tiempo de ejecución sigue siendo algo alto.

6. Algoritmo genético mejorado (GA-5)

6.1. Fundamentos teóricos

El algoritmo genético mejorado (GA-5) introduce nuevas técnicas para mejorar tanto la calidad de las soluciones como la eficiencia del proceso evolutivo.

Este emplea un **enfoque basado en elitismo**, asegurando que los mejores individuos de cada generación sean preservados directamente en la siguiente, sin someterse a procesos de cruce o mutación. Este método reduce el riesgo de pérdida de las mejores soluciones, fomentando una convergencia más robusta hacia el óptimo global.

Otra mejora relevante es la implementación del operador de cruce ***Partially Mapped Crossover* (PMX)**. Este método garantiza una mayor preservación de la información genética al respetar la relación entre genes de los padres, lo que es particularmente beneficioso en problemas donde el orden relativo de los elementos es importante, como en problemas de optimización combinatoria (Goldberg & Lingle Jr., 1985).

Además, el GA-5 introduce una **estrategia dinámica para la tasa de mutación**, reduciéndola progresivamente a lo largo de las generaciones. Esto permite que el algoritmo favorezca la exploración en las primeras etapas y la explotación en las etapas finales, optimizando el balance entre diversidad y convergencia.

Finalmente, se mantiene el uso de optimización local (*hill climbing*) como en el GA-4, pero se aplica únicamente a una fracción de los mejores individuos seleccionados mediante elitismo, lo que reduce significativamente el costo computacional mientras se maximiza el impacto de la optimización local.

6.2. Implementación

```python
# Selección basada en elitismo
def elitism_selection(population, fitness_vals, retain_rate=0.2):
    retain_length = int(len(population) * retain_rate)
    sorted_indices = np.argsort(fitness_vals)
    return population[sorted_indices[:retain_length]]

# Operador de cruce PMX (Partially Mapped Crossover)
def pmx_crossover(parent1, parent2):
    size = len(parent1)
    start, end = sorted(np.random.choice(size, 2, replace=False))
    offspring = -np.ones(size, dtype=int)
    offspring[start:end+1] = parent1[start:end+1]

    # Mapeo entre los genes en la región cruzada
    mapping = {parent1[i]: parent2[i] for i in range(start, end+1)}

    # Llenar el resto del offspring respetando el mapeo
    for i in range(size):
        if offspring[i] == -1:
            gene = parent2[i]
            while gene in mapping:
                gene = mapping[gene]
            offspring[i] = gene

    return offspring
```

Figura 19: Función de selección por elitismo y cruce PMX

El GA-5 presenta las siguientes diferencias con respecto a los anteriores:

- **Selección basada en elitismo**: los mejores individuos (20 % de la población) son preservados directamente en la siguiente generación, asegurando que las soluciones de mayor calidad no se pierdan durante las operaciones genéticas (ver Figura 19).
- **Operador de cruce PMX:** el cruce *Partially Mapped Crossover* reemplaza al *Order Crossover* (OX) utilizado en el GA-4. Este operador preserva la relación genética entre los elementos de los padres, mejorando la transmisión de información estructuralmente relevante a los descendientes (ver Figura 19).
- **Tasa de mutación adaptativa**: la mutación se inicia con una probabilidad del 30 % y se reduce gradualmente en un 5 % por generación. Esto promueve una mayor diversidad genética en las primeras generaciones, mientras que en las últimas se prioriza la estabilidad y la explotación de soluciones óptimas.
- **Optimización local dirigida**: a diferencia del GA-4, donde se optimizaba una mayor proporción de individuos, el GA-5 aplica la optimización local únicamente al 10 % de los mejores individuos seleccionados mediante elitismo, reduciendo el costo computacional sin comprometer la calidad de las soluciones.

```python
# Algoritmo genético mejorado
def genetic_algorithm_v5(flow, distances, population, generations=120, crossover_rate=0.9, initial_mutation_rate=0.3):
    best_fitness_per_gen = []
    mutation_rate = initial_mutation_rate

    for gen in range(generations):
        print(f"Generación {gen + 1}/{generations}: Evaluando fitness...")
        fitness_vals = fitness(population, flow, distances)
        best_fitness = np.min(fitness_vals)
        best_fitness_per_gen.append(best_fitness)

        print(f"Generación {gen + 1}: Mejor fitness = {best_fitness}")

        print(f"Generación {gen + 1}: Aplicando elitismo...")
        elites = elitism_selection(population, fitness_vals, retain_rate=0.2)

        next_generation = elites.tolist()
        while len(next_generation) < len(population):
            parents = population[np.random.choice(len(population), 2, replace=False)]
            if np.random.rand() < crossover_rate:
                offspring1 = pmx_crossover(parents[0], parents[1])
                offspring2 = pmx_crossover(parents[1], parents[0])
            else:
                offspring1, offspring2 = parents[0], parents[1]
            if np.random.rand() < mutation_rate:
                offspring1 = mutate(offspring1)
            if np.random.rand() < mutation_rate:
                offspring2 = mutate(offspring2)
            next_generation.extend([offspring1, offspring2])

        population = np.array(next_generation[:len(population)])

        print(f"Generación {gen + 1}: Aplicando optimización local...")
        top_individuals = elitism_selection(population, fitness_vals, retain_rate=0.1)
        for i, individual in enumerate(top_individuals):
            optimized_individual, _ = hill_climbing_3(individual, flow, distances)
            population[i] = optimized_individual

        mutation_rate *= 0.95

    best_idx = np.argmin(fitness(population, flow, distances))
    return population[best_idx], fitness(population[best_idx:best_idx+1], flow, distances)[0], best_fitness_per_gen
```

Figura 20: Función de GA-5.

Estas mejoras hacen que el GA-5 sea una versión más avanzada, enfocada en la preservación de la calidad genética y la eficiencia computacional, lo que lo convierte en una herramienta adecuada para problemas de optimización de alta complejidad.

Se realizaron dos pruebas de ejecución, una donde el número de generaciones es 50, y otra donde se aumenta a 120, extendiendo el tiempo disponible para explorar y explotar el espacio de búsqueda.

6.3. Primera versión (GA-5.1)

El GA-5.1 alcanzó un costo mínimo de **45193052** en la generación 50, con un tiempo de ejecución total de 433.18 segundos (aproximadamente **7.22 minutos**). Este resultado destaca por ser el mejor fitness registrado entre todas las variantes, evidenciando la efectividad de los ajustes realizados en esta versión. Además, el tiempo de ejecución es considerablemente más bajo en comparación con GA-4 y otras variantes previas, lo que lo convierte en una opción bastante más eficiente.

```
Generación 50/50: Evaluando fitness...
Generación 50: Mejor fitness = 45193052
Generación 50: Aplicando elitismo...
Generación 50: Aplicando optimización local...
Tiempo de ejecución: 433.18 segundos
Mejor permutación encontrada:
[ 60 174 137  43 192 124 114  81   4 126  23  19  14 224 113 243 135  45
  21 183  64 186 212  82 154 152 203 166  86 191  69  28  71 103  94 221
 143 206 118 160 255  58  34 178  77 249  67  54   1 181 195 216  32 218
 131 229 157 169 235 238  26 145  63  31  96 171 116  36   8 109 253 163
 148  90   6 188 201 149 107 139 226  11  49  88  92  41 214  84 231  56
 105 209 167 227 211 129 223 225 161 190   9 132 156 153 237 230  37  53
 199  18  76  73 202 172 220 176  46 141 117  30  57  29 247 119  16 110
 241  48 150 170 151 197 111 194 180  87  40  12  44 245  10 248 246  27
  13  95  33 128 123  80 136  59  24 251 205 140  74 158 173   0 213 179
  78 184  65 177 240  70 252  75 125 242  55  61  50  62  39 100 142 196
 159 144  38 127  22  17  68  35  51 138 254 134 234 222 210 189 233 122
 208   7 115  66 102   5 200 146   3  72 185 120 168 215 121 164  97 108
 219 236 198  52   2 217  85 104 239  25  83 204 193  98 175 133 228 130
 101  99 232 244 250  79 207  93  20  91  42 162 165 182  15 155 187 106
 147  89  47 112]
Costo asociado:
45193052
```

Figura 21: Resultados de GA-5.1.

La gráfica muestra una rápida disminución del fitness durante las primeras generaciones, seguida de una estabilización progresiva alrededor de valores óptimos en las generaciones finales. Este comportamiento es característico de un algoritmo que combina una explotación intensiva del espacio de búsqueda con estrategias de preservación de las mejores soluciones. Comparado con variantes anteriores, el GA-5.1 presenta una mayor velocidad de convergencia, alcanzando un fitness competitivo en un menor número de generaciones, y muchas más estabilidad, lo que se refleja en la ausencia de grandes oscilaciones como sucedía en los primeros algoritmos.

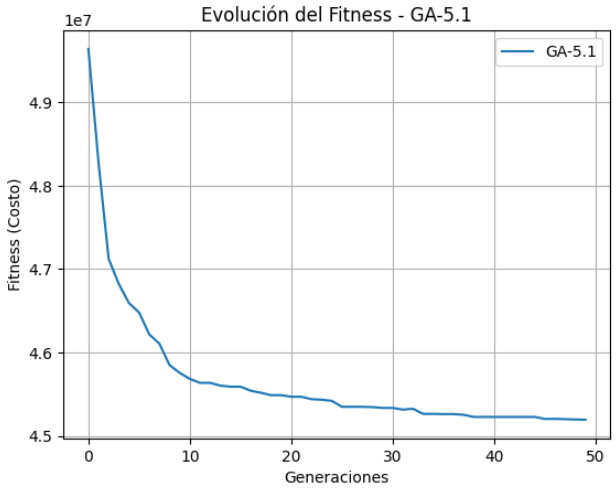

Figura 22: Gráfica de evolución del fitness de GA-5.1.

El GA-5.1 incorpora mejoras adicionales, principalmente en la eficiencia de las operaciones de optimización local y la reducción de generaciones necesarias para alcanzar una solución óptima. A pesar de utilizar un menor número de generaciones (50 en lugar de 100), logra superar en calidad

de solución a los anteriores, demostrando que los ajustes realizados optimizan tanto el tiempo de cómputo como la calidad de las soluciones obtenidas.

En resumen, GA-5.1 representa un avance significativo en la familia de algoritmos genéticos analizados, logrando un balance ideal entre calidad de la solución y tiempo de ejecución. Es particularmente adecuado para aplicaciones donde se requiere una convergencia rápida hacia soluciones de alta calidad sin comprometer la eficiencia computacional.

A continuación, se ha optado por aumentar el número de generaciones, para ver si se puede reducir aún más el coste.

6.4. Segunda versión (GA-5.2)

El GA-5.2 logró un costo mínimo de **45004702** tras 120 generaciones, superando al GA-5.1. Sin embargo, el tiempo de ejecución fue de 1006.79 segundos (aproximadamente **16.78 minutos**), más del doble que los 7.22 minutos requeridos por el GA-5.1.

```
Generación 120/120: Evaluando fitness...
Generación 120: Mejor fitness = 45004702
Generación 120: Aplicando elitismo...
Generación 120: Aplicando optimización local...
Tiempo de ejecución: 1006.79 segundos
Mejor permutación encontrada:
[101 203 114  24  65  70  59 150 116 245  32  12 222 144 104  53 248 173
 226  93  46 190 133  76 163 175  39 196 177 148  63  42  61 213 216  29
  72 122 141 239 218   7 228 188 209 241  89 184  36  74 181  67 146 194
 220 231  84 170  31 129  50 153 111 119 233 167 107  80   0 237  97 192
  27 124 136 198  22 143 126  87  19 201  41 243 254   4 235 155  10  99
  17  78   3 199  26  25 134 219  95 149 140   2 183 223 159 118  77   1
   8 193 208 127 251 112  20  73 123  13 125 162  28 108 236 166  62 180
   6 179 132  15 186 206 234 176  96 187 117  88  69 102 185 165 214 244
 128 227 247  11  85 229 164  79 212  30 120 168 156 217  86 238  55 182
  54  34 252 105  60 131 210 207 250 138  35  21  51 113 174 145 151 215
 240 142  83 205  94 246 221 158  57  71 197 211 202  68 232  56  48 195
  45  82 171 255  75 225 160 139 157 178  64 249  37 189 137  47  66  44
 100  90  40 135 230  92  18 242  33 110  98  14 172 191 161 130  23 169
  16  91 200 106 109  49 147  81 121 103 204  38 152   9 154  52 115  58
   5 224  43 253]
Costo asociado:
45004702
```

Figura 23: Resultados de GA-5.2.

La gráfica del GA-5.2 muestra una convergencia consistente hacia soluciones de alta calidad, con una rápida mejora en las primeras generaciones y una estabilización progresiva a partir de la generación 40. Este comportamiento es similar al observado en el GA-5.1, pero la mayor cantidad de generaciones (120 frente a 50 en GA-5.1) permite al GA-5.2 alcanzar un fitness ligeramente mejor, aprovechando más tiempo para explorar y explotar el espacio de búsqueda.

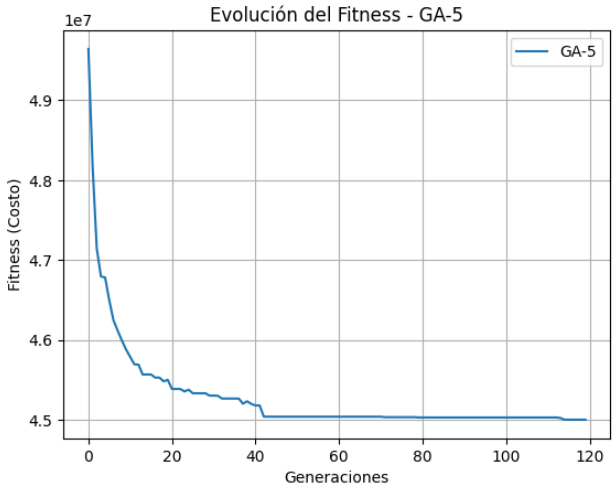

Figura 24: Gráfica de evolución del fitness de GA-5.2.

En comparación con la ejecución anterior, el GA-5.2 logró una mejor calidad de solución (45004702 frente a 45193052) gracias a un mayor número de generaciones (120 frente a 50). Sin embargo, este incremento en generaciones duplicó el tiempo de ejecución, haciéndolo menos eficiente en términos computacionales. Ambos algoritmos muestran estabilización en las generaciones finales, pero el GA-5.2 presenta una convergencia más sostenida debido al tiempo adicional para explorar y explotar el espacio de búsqueda.

En conclusión, el GA-5.2 mejora ligeramente la calidad de las soluciones respecto al GA-5.1, pero a costa de un tiempo de ejecución bastante mayor. Por tanto, es más adecuado para escenarios donde la calidad es prioritaria y el tiempo computacional no es una limitación crítica. En cambio, podría preferirse reducir el número de generaciones si se buscan resultados rápidos con una calidad aceptable.

6.5. Tercera versión (GA-5.3)

El GA-5.3 alcanzó un costo mínimo de **44911050** tras 250 generaciones, siendo el mejor resultado registrado entre todas las variantes evaluadas hasta ahora. Sin embargo, esta mejora en la calidad de la solución conlleva un tiempo de ejecución considerablemente alto, con un total de 3109.32 segundos (aproximadamente **52 minutos**). Esto muestra la necesidad de alcanzar un compromiso entre la calidad de la solución y la eficiencia computacional.

```
Generación 250/250: Evaluando fitness...
Generación 250: Mejor fitness = 44911050
Generación 250: Aplicando elitismo...
Generación 250: Aplicando optimización local...
Tiempo de ejecución: 3109.32 segundos
Mejor permutación encontrada:
[242 188  88   3  53 255 223 174   9 236 145  73  40 217 114 206  68 246
  26 221 103  91 128  82 212 248 152  43 160 210 127 219 123  46 148  49
  31 135 195  70  28 100 238  63 250 171 106 169 227  51 191  23 108 110
 182   5 225 154  93  13  11 184   0 137  76 202  17  95 162 157  85 180
  34 231  97 229  38 214 142  20 120 117  64 193 150  61 140 199  58  55
 165 131 226 254  33 222  10 130  89 159 189  67  74 149 119  80  48 185
 156 204 133  54  98 166 196   4  19 126 215 168 136 113 187 147 139 163
 132 253 116 240 183 144  66 121  44 181 192  75 232 233  52  42 158  12
 155  60  90 138  14   2   1  77 141 190  65 201 153  87  35 186 177   7
 209 134  94  24  62 207  47  18 170 101 161  37 124 245  99 208 230 179
 146  29  41  57 213  15 237 244  84 175  27 203  69 205 220 164 241 176
  25 198   6  56 224 129  36 243  22  50  86 234  39 228 216  30  32 239
 247 173 118  72  78 172 125 194 197  59  83  21  16  45  79 218 167 143
  71 107 211   8 151 102  96 249 178 115 122 109  92 105 112 111  81 104
 235 251 200 252]
Costo asociado:
44911050
```

Figura 25: Resultados de GA-5.3.

La gráfica de la Figura 26 muestra una rápida mejora del fitness en las primeras generaciones, seguida de una estabilización progresiva a partir de generaciones más avanzadas, y una ligera mejora continua hasta alcanzar las últimas generaciones. Este comportamiento indica que el GA-5.3 logra refinar los resultados, explorando las mejores regiones del espacio de búsqueda, pero a partir de cierto tiempo, las mejoras son muy graduales.

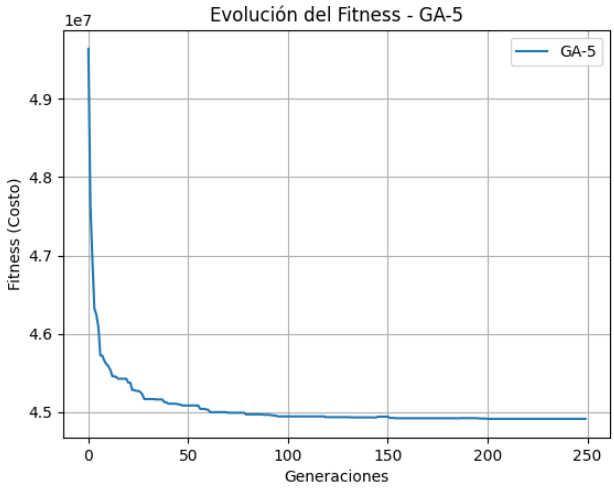

Figura 26: Gráfica de evolución del fitness de GA-5.3.

Si se compara con las dos ejecuciones anteriores de GA-5, se puede confirma que un mayor número de generaciones e iteraciones máximas para *Hill Climbing* permite encontrar soluciones más óptimas; no obstante, el tiempo requerido es considerablemente más alto que el de las variantes anteriores, lo que lo hace menos adecuado para aplicaciones con restricciones de tiempo. Por tanto, se necesita implementar un mecanismo para poder mejorar la eficiencia de búsqueda.

Por el motivo anterior, se ha decidido crear una nueva versión de GA-5 que incluya un criterio de parada que permita finalizar la ejecución del algoritmo en caso de que no se encuentren mejoras en un número determinado de generaciones (la misma utilidad del parámetro *patience*).

6.6. Cuarta versión (GA-5.4)

La implementación del criterio de parada basado en un parámetro de paciencia viene reflejada en la Figura 27.

```python
def genetic_algorithm_v5_4(flow, distances, population, generations=500, crossover_rate=0.9, initial_mutation_rate=0.3, patience=30):
    best_fitness_per_gen = []
    mutation_rate = initial_mutation_rate
    stagnation_counter = 0
    last_best_fitness = None

    for gen in range(generations):
        print(f"Generación {gen + 1}/{generations}: Evaluando fitness...")
        fitness_vals = fitness(population, flow, distances)
        best_fitness = np.min(fitness_vals)
        best_fitness_per_gen.append(best_fitness)

        if last_best_fitness is not None and best_fitness == last_best_fitness:
            stagnation_counter += 1
        else:
            stagnation_counter = 0
            last_best_fitness = best_fitness

        print(f"Generación {gen + 1}: Mejor fitness = {best_fitness}")

        if stagnation_counter >= patience:
            print(f"El algoritmo se detiene después de {gen + 1} generaciones debido a falta de mejora en las últimas {patience} generaciones.")
            break

        print(f"Generación {gen + 1}: Aplicando elitismo...")
        elites = elitism_selection(population, fitness_vals, retain_rate=0.2)
```

Figura 27: Modificación del criterio de parada de GA-5.4.

Esta cuarta versión alcanzó su mejor solución con un costo asociado de **44905968**. El tiempo total de ejecución fue de 4676.26 segundos (unas **1.3 horas**). Aunque este tiempo es considerablemente alto, el uso del criterio de parada permitió reducir el número máximo de generaciones (se detuvo en la generación 194 en lugar de llegar hasta la 500), evitando un procesamiento innecesario una vez que las mejoras se estancan.

```
Generación 194: Mejor fitness = 44905968
El algoritmo se detiene después de 194 generaciones debido a falta de mejora en las últimas 30 generaciones.
Tiempo de ejecución: 4676.26 segundos
Mejor permutación encontrada:
[ 46 160 117  35  63 244 177 201 213 242 126 191 239 220  15  87 129 215
 147  18 114  55 196 171   1 254  28 123 205   7 153  13  16 119 233 248
 227 111 184 106   5 235  33 162  81 208 218   9 108 151  43 188  70 179
 194 132  11 143 166 174  60 138  40  99 157 186  84  91  22  53 252  67
 149  78 230 102 140  89  64 222 225 104  74 112  20  57  26  50 181 198
 136  93 178 154  68  59 173 247  62 231  48  42 249 229 180 202  12 216
  27  71 223 113  80 232 115 197 195 152 253 200  38  92 175  66  90 146
 110 159 155  85  17 211 246  54 148  32   0  34  69 164   4 172  51  24
  36 105  98  95 139 226 185  10  83 150 199 142  45 203  94  79 236 189
  29  23 128 161 212 210 165  97 137 121 214 209 141 145 250 255 251  72
 243 125 221 238 234 124 190 170 144  47 109 100 133 134 167  25 192  77
 219 120 228 176 131 241 204  75  52  82  41  96 207  37   2 217  73  58
 224  56 130  44 163 183 169  49 118  19 240   3 127   6 193 158 103  86
  30 182 206 187  61 116  88  76 245  21  39 107 101 156 168   8  65  31
 237  14 122 135]
Costo asociado:
44905968
```

Figura 28: Resultados de GA-5.4.

La gráfica de la Figura 29 muestra una rápida disminución del fitness en las primeras generaciones, seguida de una estabilización progresiva, especialmente alrededor de la generación 100. La variante GA-5.4 se detuvo en la generación 194 debido al criterio de parada por estancamiento en el fitness, logrando el costo mínimo. Este comportamiento sugiere que el algoritmo aprovecha eficientemente las primeras generaciones para explorar el espacio de búsqueda y luego refina las soluciones en generaciones posteriores.

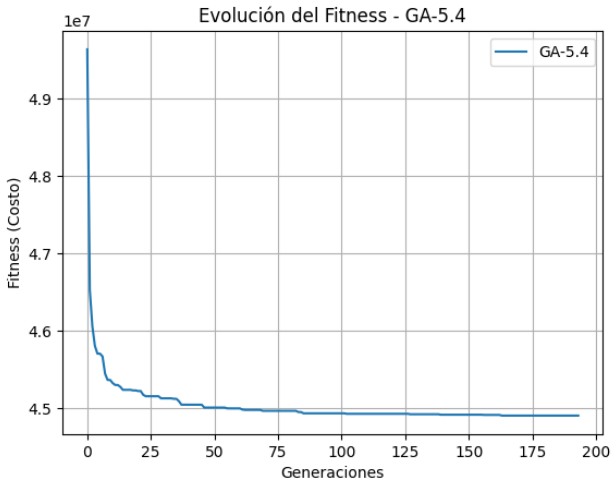

Figura 29: Gráfica de evolución del fitness de GA-5.4.

El GA-5.4 logra un costo mínimo de 44905968, superando ligeramente al GA-5.3 y demostrando que el criterio de parada adaptativo no compromete la calidad de la solución, sino que controla eficazmente la convergencia. Sin embargo, el tiempo de ejecución del GA-5.4 es mucho mayor, debido al aumento del máximo número de iteraciones en la ascensión de colinas.

Como conclusión, el GA-5.4 demuestra ser efectivo en términos de calidad de solución, logrando el mejor resultado entre las variantes evaluadas. Sin embargo, el tiempo de ejecución sigue siendo considerablemente alto incluso con el nuevo criterio de parada. Este mecanismo ayuda a evitar generaciones adicionales innecesarias, pero su impacto podría optimizarse incluso más, ajustando el umbral de generaciones sin mejora de forma dinámica, por ejemplo.

7. Conclusiones

El análisis de las distintas variantes de algoritmos genéticos aplicados al problema de asignación cuadrática (QAP) muestra la evolución en la calidad de las soluciones y eficiencia computacional a través de ajustes progresivos en las técnicas empleadas. Se explican los hallazgos principales:

- **Incremento en la calidad de las soluciones**: las variantes introdujeron mejoras significativas en el costo asociado al problema QAP, confirmando que la incorporación de técnicas avanzadas como optimización local y elitismo contribuyen a alcanzar soluciones más cercanas al óptimo. El GA-5.4, con un costo mínimo de 44.905.968, se posiciona como la variante más efectiva en términos de calidad de solución.

- **Eficiencia computacional**: esta varió considerablemente dependiendo de las técnicas utilizadas. Aunque el tiempo de ejecución aumentó en las variantes con mayor complejidad, estas lograron un balance adecuado entre tiempo computacional y calidad de las soluciones, especialmente mediante la reducción de generaciones innecesarias con criterios adaptativos.

- **Exploración vs. explotación**: se observó que las estrategias que combinan exploración en etapas iniciales (mediante tasas de mutación altas o selección por ranking) con explotación intensiva en etapas finales (como optimización local dirigida) permiten una convergencia más rápida y estable. Esto se reflejó en la disminución de las oscilaciones en las generaciones finales de variantes avanzadas como GA-5.

- **Uso del criterio de parada**: la implementación de un criterio de parada en GA-5.4 demostró ser efectiva para evitar generaciones adicionales innecesarias. Este mecanismo no comprometió la calidad de la solución, y además mejoró la eficiencia.

- **Impacto de los operadores genéticos**: probar diversos operadores de cruce, desde métodos básicos como por segmento aleatorio hasta técnicas más avanzadas como PMX, permitió una mejora en la preservación de la información genética. Asimismo, el uso de tasas de mutación adaptativas contribuyó a mantener un balance adecuado entre diversidad y estabilidad.

En conclusión, en esta práctica se puede observar la mejora incremental de los algoritmos genéticos, incorporando técnicas avanzadas y ajustando parámetros, para abordar el problema complejo del QAP. La clave está en encontrar un equilibrio entre calidad de solución, tiempo computacional y adaptabilidad de los algoritmos para satisfacer las necesidades específicas del problema.

8. Bibliografía

Ackley, D. H., & Littman, M. L. (1987). Connectionist Learning of Deterministic Finite-State Automata. *Proceedings of the National Conference on Artificial Intelligence*, 574-578.

Baker, J. E. (1985). Adaptive Selection Methods for Genetic Algorithms. *Proceedings of the International Conference on Genetic Algorithms*, 101-111.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

Goldberg, D. E., & Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms. En *Foundations of Genetic Algorithms* (pp. 69-93). Elsevier.

Goldberg, D. E., & Lingle Jr., R. (1985). Alleles, Loci, and the Traveling Salesman Problem. *Proceedings of the First International Conference on Genetic Algorithms*, 154-159.

Hinton, G. E., & Nowlan, S. J. (1987). How Learning Can Guide Evolution. *Complex Systems*, *1*(3), 495-502.

Michalewicz, Z. (1996). *Genetic Algorithms + Data Structures = Evolution Programs*. Springer Science & Business Media.

Syswerda, G. (1991). Schedule Optimization Using Genetic Algorithms. *Handbook of Genetic Algorithms*, 332-349.