UNIVERSIDAD DE GRANADA

Práctica 2: Memoria Práctica 2 Algoritmos Evolutivos

INTELIGENCIA COMPUTACIONAL curso:2024/2025

Adriano García - Giralda Milena
DNI:77944452-M adrianoggm@correo.ugr.es Grupo 2

Universidad de Granada
España

26 de enero de 2025

Índice

1. Introducción

El objetivo de esta práctica es resolver un problema de optimización típico utilizando técnicas de computación evolutiva. Para ello, se implementarán varias variantes de algoritmos evolutivos con el fin de abordar el problema de la asignación cuadrática (QAP, por sus siglas en inglés), incluyendo, como mínimo, las variantes descritas en el guion de prácticas. Estas variantes incluirán un algoritmo genético estándar, así como variantes baldwiniana y lamarckiana que incorporan técnicas de optimización local.

2. El problema de la asignación cuadrática

El problema de la asignación cuadrática o QAP (*Quadratic Assignment Problem*) es un problema fundamental de optimización combinatoria con numerosas aplicaciones. Este problema se puede describir de la siguiente forma:

Supongamos que queremos decidir dónde construir n instalaciones (por ejemplo, fábricas) y que disponemos de n posibles localizaciones donde construir dichas instalaciones. Conocemos las distancias que existen entre cada par de localizaciones y el flujo de materiales que debe existir entre las distintas instalaciones (por ejemplo, la cantidad de suministros que deben transportarse de una fábrica a otra). El objetivo es determinar la asignación de instalaciones a localizaciones que minimice el coste total de transporte de materiales.

Formalmente, si denotamos $d(i, j)$ como la distancia entre la localización i y la localización j, y $w(i, j)$ como el peso asociado al flujo de materiales que debe transportarse entre la instalación i y la instalación j, el problema consiste en encontrar la permutación p que minimice la siguiente función de coste:

$$\text{Coste} = \sum_{i,j} w(i, j) \cdot d(p(i), p(j)), \tag{1}$$

donde $p()$ representa una permutación del conjunto de instalaciones.

Al igual que en el problema del viajante de comercio (TSP, *Traveling Salesman Problem*), que puede considerarse un caso particular del QAP, una solución para este problema consiste en una permutación del conjunto de instalaciones que indica dónde construir cada una.

2.1. Relación con otros problemas de optimización

El problema del viajante de comercio o TSP puede interpretarse como un caso particular del QAP si se asume que los flujos conectan todas las instalaciones formando un único y exclusivo anillo, teniendo todos los flujos el mismo peso (una constante distinta de cero). Otros problemas de optimización combinatoria pueden plantearse de forma similar.

El problema de la asignación cuadrática es habitual en el ámbito de la Investigación Operativa y, además de emplearse para decidir la ubicación de plantas de producción, también se utiliza como modelo para problemas como:

- La colocación de componentes electrónicos en una placa de circuito impreso.
- La disposición de módulos de un circuito integrado en la superficie de un microchip.

Debido a su interés teórico y práctico, existe una amplia variedad de algoritmos que abordan la resolución del QAP. Sin embargo, al tratarse de un problema NP-completo, el diseño y aplicación de

algoritmos exactos no es viable cuando n es grande. Por ello, en esta práctica nos centraremos en el diseño de algoritmos evolutivos y evaluaremos su rendimiento en instancias concretas del problema.

3. Implementación y análisis de resultados

Se implementará un algoritmo genético estándar para resolver el problema de la asignación cuadrática. Además de su implementación, se incluirá una descripción detallada del algoritmo genético utilizado, que abarcará:

- Técnica de representación.
- Mecanismo de selección.
- Operadores de cruce y mutación.

Asimismo, se proporcionará la permutación obtenida y el coste asociado a la misma para los distintos conjuntos de datos proporcionados. Se documentarán los parámetros concretos utilizados en la ejecución del algoritmo genético, incluyendo:

- Tamaño de la población.
- Número de generaciones necesario para obtener la solución encontrada.
- Probabilidades de cruce y mutación.

Además, se implementarán dos variantes adicionales del algoritmo genético estándar:

3.1. Variante baldwiniana

Esta variante incorpora técnicas de optimización local, como heurísticas *greedy*, que permiten a los individuos de la población .ªprender"durante la búsqueda. En esta variante, el fitness de cada individuo se evalúa utilizando técnicas de búsqueda local (por ejemplo, ascensión de colinas) para alcanzar un óptimo local. Sin embargo, el material genético transmitido a los descendientes proviene del individuo original, sin incluir las mejoras aprendidas.

3.2. Variante lamarckiana

En esta variante, los individuos no solo .ªprendenütilizando técnicas de búsqueda local, sino que también transmiten las mejoras adquiridas a sus descendientes. Esto significa que los rasgos adquiridos durante el aprendizaje se heredan directamente, lo que puede impactar significativamente en la convergencia del algoritmo.

3.3. Estudio comparativo

Se realizará un estudio comparativo de las tres variantes implementadas utilizando los conjuntos de datos proporcionados. Se analizarán:

- La calidad de las soluciones obtenidas.
- El tiempo necesario para obtener dichas soluciones.

Los resultados de las distintas ejecuciones se visualizarán gráficamente, mostrando la evolución del fitness de la mejor solución encontrada en cada generación.

4. Algoritmos

En esta sección se presentará el pseudocódigo de cada uno de los algoritmos implementados, los operadores utilizados y los fundamentos teóricos y prácticos que justifican las decisiones tomadas en su diseño.

4.1. Operadores

Los algoritmos desarrollados se basan en una serie de operadores fundamentales que se implementan en los siguientes módulos del directorio `/src`:

- `mutation.py`: Implementación de los operadores de mutación.
- `selection.py`: Implementación del operador de selección.
- `crossover.py`: Implementación de los operadores de cruce.
- `optimization.py`: Optimización local empleada en las variantes baldwiniana y lamarckiana.

4.1.1. Operador de Selección

El operador de selección, implementado en `selection.py`, utiliza el método de selección por torneo. Este operador selecciona un individuo de la población basándose en una competición entre un subconjunto de individuos seleccionados al azar.

Descripción del operador:

- **Entrada:**
  - `poblacion` (`numpy.ndarray`): Matriz que representa la población de individuos, con dimensiones (tamaño_población, n).
  - `fitness` (`numpy.ndarray`): Vector que contiene el fitness de cada individuo.
  - `k` (int, opcional): Tamaño del torneo, es decir, número de individuos seleccionados al azar para competir.
- **Salida:**
  - `numpy.ndarray`: Individuo seleccionado como ganador del torneo.

Pseudocódigo:

```
def seleccion_torneo(poblacion, fitness, k):
    seleccionados = np.random.choice(len(poblacion), size=k, replace=False)
    mejor_idx = seleccionados[np.argmin(fitness[seleccionados])]
    return poblacion[mejor_idx]
```

Este operador es fundamental para garantizar la diversidad genética dentro de la población y balancear la exploración y explotación del espacio de búsqueda.

4.1.2. Operador de Cruce

Revisé la Teoría y vi que PMX para el QAP era el mejor operador de cruce, por lo que decidí implementarlo.

Algorithm 1 Cruce PMX (Partially Mapped Crossover)

Require: *parent1*, *parent2*: Listas o arreglos de tamaño n.
Ensure: Dos hijos resultantes del cruce.

```
size ← longitud de parent1
hijo1 ← [−1] × size
hijo2 ← [−1] × size
punto1, punto2 ← seleccionar dos puntos al azar y ordenarlos
Copiar segmento [punto1 : punto2] de parent1 a hijo1
Copiar segmento [punto1 : punto2] de parent2 a hijo2
function COMPLETAR_HIJO(hijo, parent, parent_original)
    for i ← punto1 to punto2 do
        elemento ← parent[i]
        if elemento ∉ hijo then
            pos ← índice de parent_original[i] en parent
            while hijo[pos] ≠ −1 do
                pos ← índice de parent[pos] en parent_original
            end while
            hijo[pos] ← elemento
        end if
    end for
    for i ← 0 to size − 1 do
        if hijo[i] = −1 then
            hijo[i] ← parent[i]
        end if
    end for
    return hijo
end function
hijo1 ← COMPLETAR_HIJO(hijo1, parent2, parent1)
hijo2 ← COMPLETAR_HIJO(hijo2, parent1, parent2)
return hijo1, hijo2
```

El operador PMX (Partially Mapped Crossover) es una técnica de recombinación utilizada en algoritmos genéticos para problemas de permutaciones, como el problema del viajante de comercio. Este método asegura que los hijos hereden características válidas de los padres y mantiene la unicidad de los elementos en la solución.

- Selección de puntos de cruce: Se eligen dos puntos al azar dentro de los padres. Estos puntos delimitan un segmento en cada padre que será intercambiado.
- Copia inicial: Los elementos dentro del segmento seleccionado del primer padre se copian al primer hijo, y lo mismo ocurre con el segundo padre hacia el segundo hijo.
- Mapeo y completado: Los elementos restantes de cada padre se mapean en el otro hijo utilizando las relaciones de posición creadas por el segmento intercambiado. Esto asegura que no haya duplicados en los hijos.
- Relleno: Las posiciones que no se llenaron durante el mapeo se completan con los elementos restantes del otro padre.

El resultado son dos hijos que combinan características de ambos padres y respetan las restricciones de unicidad del problema.

4.1.3. Operador de Mutación

El operador de mutación es una herramienta crucial en algoritmos genéticos, ya que permite explorar nuevas áreas del espacio de búsqueda y evitar caer en óptimos locales, manteniendo al mismo tiempo la diversidad de la población.

Algorithm 2 Mutación por Intercambio (Swap Mutation)

Require: *individuo*: Lista o arreglo a mutar de tamaño n.
Require: *tasa_mutacion*: Probabilidad de aplicar la mutación.
Ensure: *individuo_mutado*: Individuo resultante tras la mutación.
1: *individuo_mutado* ← copia de *individuo*
2: **if** random() ¡ *tasa_mutacion* **then**
3: Seleccionar dos posiciones i, j al azar, con $i \neq j$
4: Intercambiar *individuo_mutado*[i] con *individuo_mutado*[j]
5: **end if**
6: **return** *individuo_mutado*

La mutación por intercambio (*swap mutation*) es un operador que realiza un pequeño cambio en un individuo para garantizar la diversidad de la población en algoritmos genéticos. Este operador funciona de la siguiente manera:

1. **Evaluación de la probabilidad de mutación**: Se compara un número aleatorio con la tasa de mutación. Si el número es menor, se procede con la mutación; en caso contrario, no se realiza ningún cambio.

2. **Selección de posiciones**: Se eligen dos índices aleatorios distintos dentro del individuo.

3. **Intercambio**: Los valores en las posiciones seleccionadas se intercambian, lo que altera ligeramente el individuo sin modificar su longitud ni introducir duplicados.

4. **Devolución del individuo mutado**: El resultado es un individuo ligeramente modificado, que puede explorar una nueva región del espacio de soluciones.

Este operador es especialmente útil para problemas de permutación, como el problema del viajante de comercio, donde un intercambio mantiene la validez del individuo y permite explorar diferentes configuraciones.

4.2. Optimización

La optimización es una parte fundamental en los algoritmos genéticos y heurísticos, ya que permite mejorar la calidad de las soluciones encontradas. Durante el desarrollo, se probaron múltiples enfoques para implementar la optimización en los algoritmos lamarckiano y baldwiniano. Sin embargo, los costes computacionales asociados llevaron a descartar métodos de búsqueda local tradicionales como *2-opt* y *greedy*. Finalmente, se propuso un enfoque de **búsqueda local con máscaras**, que se detalla a continuación.

4.2.1. Búsqueda Local con Máscaras

Este enfoque utiliza dos tipos de máscaras booleanas para mejorar la eficiencia en la búsqueda local:

1. **Máscara de combinaciones prohibidas**: Una matriz booleana que evita la evaluación repetida de combinaciones ya probadas.
2. **Máscara de exclusión**: Una máscara unidimensional que bloquea índices específicos en la permutación, reduciendo la cantidad de combinaciones evaluadas.

El algoritmo sigue los siguientes pasos principales:

1. **Inicialización**:
   - Se genera una permutación inicial que representa la solución candidata.
   - Se calculan las matrices de flujos y distancias asociadas al problema.
   - Se inicializan las máscaras booleanas.
2. **Evaluación de vecinos**:
   - Se genera un subconjunto aleatorio de combinaciones (r, s) de índices, respetando las máscaras.
   - Para cada par (r, s), se calcula el coste incremental utilizando un modelo optimizado con `Numba`, lo que reduce significativamente los tiempos de cómputo.
3. **Actualización de la solución**:
   - Si se encuentra una mejora, se actualiza la permutación, el coste asociado y las máscaras.
   - Si no hay mejoras, se bloquean los índices afectados y el proceso se detiene.
4. **Iteración**: El proceso se repite hasta alcanzar el máximo de iteraciones permitido o hasta que no se encuentren más mejoras.

4.2.2. Implementación del Algoritmo

A continuación, se presentan las funciones principales utilizadas en el enfoque de búsqueda local con máscaras:

- `calcular_delta_coste_numba`: Calcula de manera eficiente la diferencia en el coste al intercambiar dos posiciones en la permutación. Este cálculo se realiza utilizando una fórmula incremental basada en las matrices de flujos y distancias, lo que elimina la necesidad de recalcular el coste total del individuo de forma completa, optimizando significativamente el rendimiento del algoritmo.
- `actualizar_coste_incremental`: Actualiza el coste total de forma incremental tras un intercambio, aprovechando el valor de Δ calculado previamente. Este enfoque permite reducir el coste computacional al recalcular únicamente la diferencia asociada a las posiciones intercambiadas, en lugar de recalcular el coste completo desde cero.

- `mejor_vecino_con_mascara`: Encuentra el mejor vecino evaluando únicamente combinaciones válidas definidas por las máscaras. Selecciona un subconjunto aleatorio de combinaciones (r, s) y calcula el delta para cada par. Este método utiliza una máscara booleana que permite evitar el cálculo innecesario de deltas o costes para pares (r, s) ya evaluados, reduciendo así el tiempo computacional. La máscara almacena qué pares de índices (r, s) han sido evaluados previamente durante la búsqueda local. Si una combinación (r, s) ha sido evaluada, se marca como prohibida (`máscara[r, s] = True`), garantizando que no vuelva a ser considerada en iteraciones futuras.
- `calcula_busqueda_local_con_mascara`: Ejecuta el proceso iterativo de búsqueda local, evaluando vecinos y actualizando la solución hasta alcanzar un máximo de iteraciones o no encontrar mejoras. En este proceso, se utiliza una máscara adicional que controla dinámicamente qué partes de la permutación son candidatas para ser alteradas. Esta máscara mejora la eficiencia al reducir el espacio de búsqueda al descartar índices que no contribuyen a mejorar la solución. Un índice se considera activo (`máscara_permutacion[i] = True`) si puede formar parte de un par (r, s); en caso contrario, se bloquea temporalmente (`máscara_permutacion[i] = False`). Si durante una iteración no se encuentra una mejora al alterar los índices r y s, estos se bloquean, restringiendo la búsqueda futura.

Algorithm 3 Cálculo del Delta de Coste

Require: *individuo*, *flujo_matrix*, *distancia_matrix*, r, s
Ensure: Δ (Cambio en el coste total)

$\Delta \leftarrow 0{,}0$
$p_r \leftarrow individuo[r]$
$p_s \leftarrow individuo[s]$
for $k \leftarrow 0$ **to** $n-1$ **do**
  if $k \neq r$ **and** $k \neq s$ **then**
    $p_k \leftarrow individuo[k]$
    $\Delta \leftarrow \Delta + flujo\_matrix[r,k] \cdot (distancia\_matrix[p_s, p_k] - distancia\_matrix[p_r, p_k]) +$
    $flujo\_matrix[s,k] \cdot (distancia\_matrix[p_r, p_k] - distancia\_matrix[p_s, p_k]) +$
    $flujo\_matrix[k,r] \cdot (distancia\_matrix[p_k, p_s] - distancia\_matrix[p_k, p_r]) +$
    $flujo\_matrix[k,s] \cdot (distancia\_matrix[p_k, p_r] - distancia\_matrix[p_k, p_s])$
  end if
end for
return Δ

Algorithm 4 Actualización del Coste Incremental

Require: *coste_actual*, *flujo_matrix*, *distancia_matrix*, *individuo*, r, s
Ensure: *nuevo_coste*

$\Delta \leftarrow$ CALCULAR_DELTA_COSTE_NUMBA(*individuo*, *flujo_matrix*, *distancia_matrix*, r, s)
nuevo_coste $\leftarrow$ *coste_actual* $+\Delta$
return *nuevo_coste*

Algorithm 5 Búsqueda del Mejor Vecino con Máscaras

Require: *individuo*, *flujo_matrix*, *distancia_matrix*, *mascara*, *mascara_permutacion*, *coste_actual*, *max_vecinos*
Ensure: *nuevo_individuo*, *nuevo_coste*, *mascara*, *mascara_permutacion*
Generar un subconjunto aleatorio de pares (r, s) donde $r \neq s$ y *mascara*[r, s] = `False`
Inicializar *mejor_delta* ← 0,0 y *mejor_swap* ← `None`
for (r, s) **in** *vecinos* **do**
    if *mascara_permutacion*[r] = `True` **and** *mascara_permutacion*[s] = `True` **then**
        $\Delta \leftarrow$ CALCULAR_DELTA_COSTE_NUMBA(*individuo*, *flujo_matrix*, *distancia_matrix*, r, s)
        if $\Delta <$ *mejor_delta* **then**
            mejor_delta ← Δ
            mejor_swap ← (r, s)
        end if
    end if
end for
if *mejor_swap* ≠ `None` **then**
    Realizar el intercambio r, s en *individuo*
    Actualizar *mascara* y *mascara_permutacion*
    return *nuevo_individuo*, *nuevo_coste*, *mascara*, *mascara_permutacion*
else
    Bloquear índices en *mascara_permutacion*
    return *individuo*, *coste_actual*, *mascara*, *mascara_permutacion*
end if

Algorithm 6 Búsqueda Local con Máscaras

Require: *individuo*, *flujo_matrix*, *distancia_matrix*, *max_iter*, *max_vecinos*
Ensure: *mejor_individuo*, *mejor_coste*
Inicializar *mejor_individuo* ← *individuo*
mejor_coste ← CALCULAR_COSTE(*individuo*, *flujo_matrix*, *distancia_matrix*)
Inicializar máscaras (*mascara*, *mascara_permutacion*)
for _ **in** 1 **to** *max_iter* **do**
    nuevo_individuo, *nuevo_coste*, *mascara*, *mascara_permutacion* ←
        MEJOR_VECINO_CON_MASCARA(*mejor_individuo*, *flujo_matrix*, *distancia_matrix*, *mascara*, *mascara_permutacion*, *mejor_coste*, *max_vecinos*)
    if *nuevo_coste* < *mejor_coste* **then**
        mejor_individuo ← *nuevo_individuo*
        mejor_coste ← *nuevo_coste*
    else
        break
    end if
end for
return *mejor_individuo*, *mejor_coste*

4.2.3. Ventajas del Enfoque Propuesto

El uso de máscaras en el enfoque de búsqueda local aporta varias ventajas clave para la optimización combinatoria, entre las que destacan:

- **Reducción eficiente del espacio de búsqueda:** Al evitar combinaciones ya evaluadas mediante el uso de máscaras booleanas, se elimina la redundancia en los cálculos. Esto reduce significativamente el espacio de búsqueda, permitiendo al algoritmo centrarse únicamente en soluciones no exploradas.
- **Aceleración del cálculo:** La implementación de funciones optimizadas con `Numba` permite realizar operaciones intensivas, como el cálculo incremental de costes (Δ), de manera más eficiente. Esto disminuye el tiempo de ejecución y hace viable el análisis de problemas de mayor escala.
- **Mejora del balance entre exploración y explotación:** Limitar el número de vecinos evaluados en cada iteración asegura un equilibrio entre la exploración de nuevas áreas del espacio de soluciones y la explotación de las regiones prometedoras. Esto optimiza la calidad de las soluciones alcanzadas en un tiempo razonable.
- **Adaptabilidad dinámica:** Las máscaras permiten ajustar dinámicamente qué combinaciones e índices pueden ser evaluados, proporcionando un control granular sobre la búsqueda. Esto evita que el algoritmo se estanque en óptimos locales y mejora su capacidad de adaptación durante el proceso de optimización.
- **Eficiencia en problemas de gran escala:** Este enfoque es especialmente adecuado para problemas combinatorios complejos, como el problema de asignación cuadrática (QAP), donde la cantidad de combinaciones posibles crece exponencialmente con el tamaño del problema.

4.3. Algoritmo Genético Genérico

En esta sección se describe la implementación de un algoritmo genético genérico diseñado para resolver el Problema de Asignación Cuadrática (*Quadratic Assignment Problem*, QAP). El algoritmo combina estrategias de inicialización, selección, cruce, mutación y elitismo para buscar soluciones óptimas en un espacio de búsqueda combinatoria.

4.3.1. Estructura General del Algoritmo

El algoritmo genético sigue los pasos tradicionales:

1. **Inicialización:** Se genera una población inicial compuesta por individuos generados de forma aleatoria y mediante una heurística greedy.
2. **Evaluación:** Se calcula el fitness de cada individuo utilizando las matrices de flujos y distancias.
3. **Selección:** Se seleccionan pares de padres mediante el método de torneo para producir descendencia.
4. **Cruce:** Se aplica el operador de cruce PMX (*Partially Mapped Crossover*) con una cierta probabilidad.
5. **Mutación:** Se realiza una mutación de intercambio (*swap mutation*) en los descendientes con una probabilidad definida.
6. **Elitismo:** El mejor individuo de cada generación se mantiene para garantizar la preservación de soluciones prometedoras.
7. **Iteración:** Se repiten los pasos anteriores durante un número fijo de generaciones o hasta alcanzar un criterio de parada.

4.3.2. Componentes del Algoritmo

4.3.2.1 Inicialización de la Población.

La población inicial se genera utilizando dos enfoques:

- **Individuos aleatorios:** Se generan permutaciones aleatorias de las asignaciones.
- **Individuos greedy:** Se asignan instalaciones con mayor flujo total a ubicaciones con menor distancia total, utilizando una heurística greedy.

4.3.2.2 Selección.

Se utiliza el método de selección por torneo, en el cual se seleccionan aleatoriamente dos subgrupos de individuos y el mejor de cada subgrupo es elegido como padre. Este método favorece individuos con mejor fitness sin excluir completamente a otros.

4.3.2.3 Cruce.

El cruce se realiza mediante el operador PMX (*Partially Mapped Crossover*), que garantiza que las permutaciones resultantes sean válidas, preservando la unicidad de las asignaciones.

4.3.2.4 Mutación.

Se utiliza un operador de mutación por intercambio (*swap mutation*) que intercambia dos posiciones de la permutación con una probabilidad determinada. Esto introduce diversidad en la población y permite escapar de óptimos locales.

4.3.2.5 Elitismo.

El mejor individuo de cada generación se copia directamente a la siguiente generación, garantizando la preservación de las mejores soluciones encontradas.

4.3.3. Implementación y Parámetros

El algoritmo utiliza los siguientes parámetros configurables:

- **Tamaño de la población:** Número de individuos en cada generación.
- **Número de generaciones:** Cantidad de iteraciones del algoritmo.
- **Tasa de cruce:** Probabilidad de aplicar el operador PMX.
- **Tasa de mutación:** Probabilidad de realizar una mutación en un individuo.
- **Elitismo:** Indicador de si se preserva el mejor individuo de cada generación.

4.3.4. Resultados y Métricas de Evaluación

Durante la ejecución del algoritmo, se registra:

- El mejor fitness de cada generación, que permite evaluar la mejora progresiva del algoritmo.
- El individuo correspondiente al mejor fitness encontrado, representando la solución óptima propuesta.
- Un historial de fitness que ilustra la evolución de la calidad de las soluciones a lo largo de las generaciones.

4.3.5. Ventajas del Algoritmo

- Combina exploración global (individuos aleatorios) con explotación local (individuos greedy) en la inicialización.
- Utiliza operadores de cruce y mutación que garantizan la validez de las soluciones.
- Implementa elitismo para preservar soluciones prometedoras entre generaciones.
- Permite la personalización mediante parámetros ajustables que se adaptan a las características del problema.

Algorithm 7 Algoritmo Genético Genérico

Require: n: Número de instalaciones/localizaciones, *flujo_matrix*, *distancia_matrix*, *parámetros*.
Ensure: Mejor solución encontrada (*individuo*, *coste*) e historial de fitness.

Inicializar parámetros:

- *poblacion*: Tamaño de la población.
- *generaciones*: Número de generaciones.
- *tasa_cruce*: Probabilidad de aplicar cruce.
- *tasa_mutacion*: Probabilidad de aplicar mutación.
- *elitismo*: Mantener el mejor individuo entre generaciones.

Paso 1: Inicialización de la población
Generar *mitad_poblacion* individuos aleatorios.
Generar el resto de los individuos utilizando la heurística greedy.
Calcular *fitness* para cada individuo en la población.
Paso 2: Evaluación inicial
Encontrar el mejor individuo (*mejor_individuo*, *mejor_fitness*).
Inicializar el historial de fitness con *mejor_fitness*.
for $gen \leftarrow 1$ **to** *generaciones* **do**
    Paso 3: Generación de nueva población
    Inicializar una nueva población vacía.
    if *elitismo* **then**
        Incluir *mejor_individuo* en la nueva población.
    end if
    while *nueva población* no esté completa **do**
        Seleccionar dos padres usando *selección por torneo*.
        Cruce:
        if *random()* $<$ *tasa_cruce* **then**
            Generar dos hijos con el operador *cruce_pmx*.
        else
            Copiar los padres directamente como hijos.
        end if
        Mutación:
        Aplicar *mutacion_swap* a cada hijo con probabilidad *tasa_mutacion*.
        Añadir los hijos a la nueva población.
    end while
    Paso 4: Evaluación de la población
    Calcular *fitness* para cada individuo.
    Encontrar el mejor individuo de la generación actual.
    Paso 5: Actualización del mejor individuo
    if *mejor_fitness actual* $<$ *mejor_fitness global* **then**
        Actualizar *mejor_individuo* y *mejor_fitness*.
    end if
    Añadir *mejor_fitness actual* al historial.
    if $(gen \mod 100 = 0) \vee (gen = 1)$ **then**
        Imprimir el progreso de la generación actual.
    end if
end for
return *mejor_individuo*, *mejor_fitness*, historial de fitness.

4.4. Variante Baldwiniana del Algoritmo Genético

La variante baldwiniana del algoritmo genético introduce una búsqueda local en el proceso de evaluación de fitness para mejorar la calidad de las soluciones en cada generación. Este enfoque combina la exploración global del espacio de búsqueda, característica de los algoritmos genéticos, con la explotación local proporcionada por métodos de optimización individual.

4.4.1. Estructura del Algoritmo

El algoritmo sigue estos pasos principales:

1. **Inicialización:** Se genera una población inicial compuesta por permutaciones aleatorias. Una parte de esta población se optimiza utilizando búsqueda local basada en máscaras booleanas.
2. **Evaluación:** El fitness de cada individuo se calcula a partir de las matrices de flujos y distancias.
3. **Selección:** Se seleccionan padres mediante el método de torneo, favoreciendo los individuos con mejor fitness.
4. **Cruce:** Se utiliza el operador de cruce PMX (*Partially Mapped Crossover*) con una probabilidad determinada para generar descendencia.
5. **Mutación:** Se aplica una mutación por intercambio (*swap mutation*) con una probabilidad definida, asegurando la diversidad en la población.
6. **Optimización local:** Una parte de la población se optimiza mediante búsqueda local al final de cada generación.
7. **Elitismo:** El mejor individuo de cada generación se conserva en la siguiente para mantener las soluciones prometedoras.
8. **Iteración:** El proceso se repite durante un número fijo de generaciones o hasta que se alcance un criterio de parada.

4.4.2. Componentes del Algoritmo

4.4.2.1 Inicialización de la Población.

La población inicial se genera de forma completamente aleatoria mediante permutaciones válidas. Se asegura que una parte de los individuos sea optimizada localmente desde el inicio para proporcionar una base sólida al algoritmo.

4.4.2.2 Optimización Local.

La optimización local se aplica a una proporción fija de la población en cada generación, utilizando un enfoque de búsqueda local con máscaras booleanas. Este paso mejora la calidad de las soluciones sin alterar el comportamiento exploratorio del algoritmo genético.

4.4.2.3 Selección.

Se emplea el método de selección por torneo para elegir a los padres que generarán la descendencia. Este método favorece individuos con mejor fitness, pero también da oportunidad a otros para mantener la diversidad.

4.4.2.4 Cruce y Mutación.

- **Cruce:** El operador PMX asegura que las soluciones descendientes sean permutaciones válidas y combinan las características de ambos padres.
- **Mutación:** Se aplica el operador de intercambio a los descendientes con una probabilidad fija, introduciendo nuevas configuraciones al espacio de búsqueda.

4.4.2.5 Elitismo.

El mejor individuo de cada generación se preserva en la siguiente, asegurando la continuidad de las mejores soluciones encontradas.

4.4.3. Implementación y Parámetros

El algoritmo utiliza los siguientes parámetros configurables:

- **Tamaño de la población (*poblacion*):** Número total de individuos en cada generación.
- **Número de generaciones (*generaciones*):** Límite de iteraciones del algoritmo.
- **Tasa de cruce (*tasa_cruce*):** Probabilidad de aplicar el operador de cruce PMX.
- **Tasa de mutación (*tasa_mutacion*):** Probabilidad de realizar una mutación en cada individuo.
- **Tamaño de población optimizada (*tam_poblacion_opt*):** Proporción de individuos sometidos a optimización local.
- **Elitismo (*elitismo*):** Indica si el mejor individuo de cada generación se conserva.

4.4.4. Ventajas del Enfoque Baldwiniano

- Combina la exploración global del algoritmo genético con la explotación local de la búsqueda local.
- Mejora la calidad inicial de las soluciones mediante la optimización local de una parte de la población.
- Mantiene la diversidad de la población al limitar la aplicación de la búsqueda local y al utilizar operadores genéticos estándar.
- Preserva las mejores soluciones gracias al elitismo.
- Proporciona un equilibrio entre el tiempo computacional y la calidad de las soluciones mediante parámetros ajustables.

Algorithm 8 Variante Baldwiniana del Algoritmo Genético

Require: n: Número de instalaciones/localizaciones, *flujo_matrix*, *distancia_matrix*, *parámetros*.
Ensure: Mejor solución encontrada (*individuo*, *coste*) e historial de fitness.

Inicializar parámetros:

- *poblacion*: Tamaño de la población.
- *generaciones*: Número de generaciones.
- *tasa_cruce*: Probabilidad de aplicar el cruce.
- *tasa_mutacion*: Probabilidad de aplicar mutación.
- *elitismo*: Mantener el mejor individuo entre generaciones.
- *tam_poblacion_opt*: Tamaño de la población optimizada localmente.

Paso 1: Inicialización de la población
Generar *poblacion* aleatoria utilizando permutaciones válidas.
Seleccionar aleatoriamente *tam_poblacion_opt* individuos y aplicar búsqueda local.
Calcular *fitness* para cada individuo.
Encontrar el mejor individuo (*mejor_individuo*, *mejor_fitness*).
Inicializar *historial* con *mejor_fitness*.
for $gen \leftarrow 1$ **to** *generaciones* **do**
    Paso 2: Generación de nueva población
    Inicializar una nueva población vacía.
    if *elitismo* **then**
        Incluir *mejor_individuo* en la nueva población.
    end if
    while *nueva población* no esté completa **do**
        Seleccionar dos padres usando *selección por torneo*.
        Cruce:
        if *random()* $<$ *tasa_cruce* **then**
            Generar dos hijos con *cruce_pmx*.
        else
            Copiar directamente a los padres como hijos.
        end if
        Mutación:
        Aplicar *mutacion_swap* a cada hijo con probabilidad *tasa_mutacion*.
        Añadir los hijos a la nueva población.
    end while
    Paso 3: Optimización Local
    Seleccionar *tam_poblacion_opt* individuos aleatorios de la nueva población.
    Aplicar búsqueda local a los individuos seleccionados.
    Paso 4: Evaluación de la población
    Calcular *fitness* para cada individuo.
    Encontrar el mejor individuo de la generación actual.
    Paso 5: Actualización del mejor individuo
    if *mejor_fitness actual* $<$ *mejor_fitness global* **then**
        Actualizar *mejor_individuo* y *mejor_fitness*.
    end if
    Añadir *mejor_fitness actual* al historial.
    if $(gen \mod 100 = 0) \vee (gen = 1)$ **then**
        Imprimir el progreso de la generación actual.
    end if
end for
return *mejor_individuo*, *mejor_fitness*, historial de fitness.

4.5. Variante Lamarckiana del Algoritmo Genético

La variante Lamarckiana del algoritmo genético introduce un enfoque en el cual las mejoras realizadas durante la búsqueda local se incorporan directamente en los individuos de la población. Esto significa que la información aprendida a través de la búsqueda local se transmite explícitamente a las futuras generaciones, impactando la evolución genética del conjunto de soluciones.

4.5.1. Estructura del Algoritmo

El algoritmo sigue los pasos principales descritos a continuación:

1. **Inicialización:** Se genera una población inicial compuesta por permutaciones aleatorias. Una parte de esta población se optimiza mediante búsqueda local.
2. **Evaluación:** El fitness de cada individuo se calcula basándose en las matrices de flujos y distancias.
3. **Selección:** Se seleccionan padres mediante el método de torneo, favoreciendo los individuos con mejor fitness.
4. **Cruce:** Se aplica el operador PMX (*Partially Mapped Crossover*) con una probabilidad específica para generar descendientes.
5. **Mutación:** Se realiza una mutación por intercambio (*swap mutation*) en los descendientes con una probabilidad dada.
6. **Optimización local (Lamarckiana):** Cada descendiente generado se somete a optimización local, actualizando su configuración directamente en la población.
7. **Elitismo:** El mejor individuo de cada generación se mantiene para asegurar la continuidad de soluciones prometedoras.
8. **Iteración:** El proceso se repite durante un número predefinido de generaciones o hasta alcanzar un criterio de parada.

4.5.2. Componentes del Algoritmo

4.5.2.1 Optimización Local.

La optimización local se aplica tanto a la población inicial como a cada descendiente generado durante el proceso evolutivo. Utiliza un enfoque de búsqueda local con máscaras booleanas, lo que mejora la calidad de las soluciones y transmite el aprendizaje a las generaciones futuras.

4.5.2.2 Selección.

El método de selección por torneo se utiliza para elegir a los padres. Este método favorece los individuos con mejor fitness, pero también permite cierta diversidad al incluir individuos menos óptimos en la selección.

4.5.2.3 Cruce y Mutación.

- **Cruce:** El operador PMX garantiza que los descendientes sean permutaciones válidas, combinando características de ambos padres.
- **Mutación:** Se aplica el operador de intercambio para alterar ligeramente las permutaciones y mantener la diversidad de la población.

4.5.2.4 Elitismo.

El mejor individuo de cada generación se copia directamente a la siguiente, asegurando que las mejores soluciones se mantengan y puedan influir en las generaciones futuras.

4.5.3. Implementación y Parámetros

Los parámetros del algoritmo son configurables para adaptarse a diferentes instancias del problema:

- **Tamaño de la población (*poblacion*):** Número total de individuos en cada generación.
- **Número de generaciones (*generaciones*):** Número máximo de iteraciones.
- **Tasa de cruce (*tasa_cruce*):** Probabilidad de aplicar el operador de cruce PMX.
- **Tasa de mutación (*tasa_mutacion*):** Probabilidad de aplicar mutación en un individuo.
- **Tamaño de la población optimizada (*tam_poblacion_opt*):** Porción de la población sometida a búsqueda local.
- **Iteraciones de búsqueda local (*hill_climbing_max_iter*):** Límite de iteraciones para la optimización local.
- **Elitismo (*elitismo*):** Si se conserva o no el mejor individuo de cada generación.

4.5.4. Ventajas del Enfoque Lamarckiano

- La optimización local mejora significativamente la calidad de los individuos en cada generación.
- El aprendizaje individual se transfiere directamente a la población genética, acelerando la convergencia hacia soluciones óptimas.
- El enfoque Lamarckiano es especialmente efectivo en problemas de gran escala donde la búsqueda local puede encontrar configuraciones de alta calidad en regiones prometedoras.
- Preserva la diversidad de la población mediante mutación y selección.
- Mantiene un equilibrio entre exploración global y explotación local gracias a los parámetros ajustables.

Algorithm 9 Variante Lamarckiana del Algoritmo Genético

Require: n: Número de instalaciones/localizaciones, *flujo_matrix*, *distancia_matrix*, *parámetros*.
Ensure: Mejor solución encontrada (*individuo*, *coste*) e historial de fitness.

Inicializar parámetros:

- *poblacion*: Tamaño de la población.
- *generaciones*: Número de generaciones.
- *tasa_cruce*: Probabilidad de aplicar cruce.
- *tasa_mutacion*: Probabilidad de aplicar mutación.
- *elitismo*: Mantener el mejor individuo entre generaciones.
- *tam_poblacion_opt*: Tamaño de la población optimizada localmente.
- *hill_climbing_max_iter*: Número máximo de iteraciones para búsqueda local.

Paso 1: Inicialización de la población
Generar *poblacion* inicial aleatoria utilizando permutaciones válidas.
Seleccionar *tam_poblacion_opt* individuos aleatorios y aplicar búsqueda local para optimizarlos.
Calcular *fitness* para cada individuo.
Encontrar el mejor individuo (*mejor_individuo*, *mejor_fitness*).
Inicializar *historial* con *mejor_fitness*.
for *gen* $\leftarrow 1$ **to** *generaciones* **do**
    Paso 2: Generación de nueva población
    Inicializar una nueva población vacía.
    if *elitismo* **then**
        Incluir *mejor_individuo* en la nueva población.
    end if
    while *nueva población* no esté completa **do**
        Seleccionar dos padres usando *selección por torneo*.
        Cruce:
        if *random()* $<$ *tasa_cruce* **then**
            Generar dos hijos con *cruce_pmx*.
        else
            Copiar directamente a los padres como hijos.
        end if
        Mutación:
        Aplicar *mutacion_swap* a cada hijo con probabilidad *tasa_mutacion*.
        Optimización Local (Lamarckiana):
        Aplicar búsqueda local a ambos hijos, actualizando sus configuraciones en la población.
        Añadir los hijos optimizados a la nueva población.
    end while
    Paso 3: Evaluación de la población
    Calcular *fitness* para cada individuo.
    Encontrar el mejor individuo de la generación actual.
    Paso 4: Actualización del mejor individuo
    if *mejor_fitness actual* $<$ *mejor_fitness global* **then**
        Actualizar *mejor_individuo* y *mejor_fitness*.
    end if
    Añadir *mejor_fitness actual* al historial.
    if $(gen \mod 100 = 0) \vee (gen = 1)$ **then**
        Imprimir el progreso de la generación actual.
    end if
end for
return *mejor_individuo*, *mejor_fitness*, historial de fitness.

5. Estudio Comparativo y Resultados

En esta sección se presentan los resultados obtenidos al aplicar las variantes del algoritmo genético (Genérico, Baldwiniano y Lamarckiano) al Problema de Asignación Cuadrática (QAP). Para ello, se llevaron a cabo múltiples experimentos variando los tamaños de población y el número de generaciones, analizando el impacto de estos parámetros en la calidad de las soluciones y en el coste computacional. Para la reproducibilidad de los experimentos se ha usado la semilla 42.

5.1. Metodología Experimental

Se realizaron experimentos utilizando tres variantes del algoritmo genético:

- **Algoritmo Genético Genérico:** No incluye optimización local en ninguna etapa, lo que lo hace menos costoso computacionalmente.
- **Variante Baldwiniana:** Aplica optimización local únicamente para evaluar el fitness de los individuos, sin modificar directamente la población.
- **Variante Lamarckiana:** Incorpora el aprendizaje de la optimización local en los individuos, actualizando directamente sus configuraciones y transmitiendo estos cambios a la población.

Los parámetros iniciales considerados en los experimentos fueron:

- Tamaño de la población (*poblacion*): Se probaron valores de 50, 150 y 500 individuos.
- Número de generaciones (*generaciones*): Se evaluaron 500 generaciones y, para el Algoritmo Genético Genérico, se utilizó una configuración de 300 poblaciones y 3000 generaciones. Esta última configuración se estableció para equiparar el tiempo de ejecución aproximado de 1 hora y 45 minutos, similar al de las otras dos variantes.
- Tasa de cruce (*tasa_cruce*): Fijada en 0.8.
- Tasa de mutación (*tasa_mutacion*): Fijada en 0.08 para el Genérico y 0.12 para las variantes Baldwiniana y Lamarckiana.
- Tamaño de la población optimizada (*tam_poblacion_opt*): Búsqueda local aplicada a solo 50 individuos de la población en las variantes Baldwiniana y Lamarckiana.

Cada experimento se repitió 10 veces para reducir la variabilidad inherente a los algoritmos genéticos, y se reportó el mejor, peor y promedio de los valores de fitness obtenidos.

5.2. Resultados Comparativos

A continuación, se presenta un análisis del coste computacional y la calidad de las soluciones obtenidas para cada variante del algoritmo:

5.2.1. Coste Computacional

El coste computacional de las variantes se analizó en términos del tiempo total de ejecución. Como era de esperarse, el coste aumenta con la inclusión de optimización local, siguiendo la tendencia:

$$\text{Genérico} < \text{Baldwiniano} < \text{Lamarckiano}$$

- **Algoritmo Genético Genérico:** Presenta el menor coste computacional debido a la ausencia de optimización local. Es adecuado para problemas con restricciones de tiempo o escalas muy grandes.

- **Variante Baldwiniana:** Incrementa el coste debido a la aplicación de búsqueda local durante la evaluación del fitness. Sin embargo, este enfoque equilibra bien el tiempo de ejecución y la calidad de las soluciones.
- **Variante Lamarckiana:** Tiene el mayor coste computacional, ya que aplica búsqueda local tanto en la evaluación como en la actualización directa de los individuos en la población. No obstante, esta variante converge más rápidamente hacia soluciones de alta calidad.

5.2.2. Calidad de las Soluciones

En términos de calidad, las soluciones obtenidas siguieron la misma tendencia:

$$\text{Genérico} < \text{Baldwiniano} < \text{Lamarckiano}$$

De forma general, donde un fitness más bajo indica una mejor calidad de la solución.

- **Algoritmo Genético Genérico:** Aunque menos costoso, las soluciones obtenidas tienden a ser menos óptimas, especialmente en problemas de mayor complejidad.
- **Variante Baldwiniana:** Mejora significativamente la calidad de las soluciones gracias a la evaluación basada en búsqueda local, sin alterar la población directamente.
- **Variante Lamarckiana:** Produce las mejores soluciones debido a la integración directa de los resultados de la búsqueda local en los individuos, acelerando la convergencia hacia configuraciones óptimas.

Sin embargo, el mejor coste total se obtuvo con el Algoritmo Genético Genérico utilizando una configuración de 300 poblaciones y 3000 generaciones (Genérico4), que equilibró eficazmente el tiempo de ejecución con la calidad de las soluciones. Le siguieron como mejores resultados generales la variante Baldwiniana3 y Lamarckiano3, ambas con 500 poblaciones y 500 generaciones, mostrando un rendimiento superior en términos de calidad con un coste computacional manejable.

5.3. Impacto del Tamaño de Población y Generaciones

Se experimentó con diferentes configuraciones de tamaño de población y número de generaciones para analizar su efecto en las variantes del algoritmo. Los resultados mostraron que:

- **Tamaño de población:** Un mayor tamaño de población mejora la exploración del espacio de búsqueda y la calidad de las soluciones, pero incrementa linealmente el coste computacional. Por ejemplo, en el Algoritmo Genético Genérico, aumentar la población de 50 a 500 individuos redujo el fitness promedio de 46 785 484 a 45 606 166. Los tamaños de 100 y 200 individuos lograron un buen balance entre calidad y tiempo de ejecución.
- **Número de generaciones:** Incrementar el número de generaciones mejora la convergencia hacia soluciones óptimas, aunque con retornos decrecientes. Se observó que entre 500 y 1000 generaciones fueron suficientes para la mayoría de los experimentos, permitiendo una adecuada exploración y explotación del espacio de soluciones.

5.4. Resumen de Resultados

La Tabla 1 muestra un resumen de los resultados obtenidos para cada variante del algoritmo genético, considerando diferentes tamaños de población y generaciones.

Cuadro 1: Comparativa de variantes del algoritmo genético

Variante	Calidad Promedio (Fitness)	Población	Generaciones
Genérico1	46 785 484	50	500
Genérico2	45 971 404	150	500
Genérico3	45 606 166	500	500
Genérico4	44 905 542	300	3000
Baldwiniano1	44 892 868	50	500
Baldwiniano2	46 126 746	150	500
Baldwiniano3	44 907 876	500	500
Lamarckiano1	44 951 010	50	500
Lamarckiano2	44980736.0	150	500
Lamarckiano3	44 926 294	500	500
Baldwiniano Mejorado	**4.49e7**	**50**	**500**

5.5. Análisis de Gráficas de Evolución del Fitness

En esta sección se presentan las gráficas más interesantes que ilustran la evolución del fitness a lo largo de las generaciones para las distintas variantes del algoritmo genético. Estas gráficas permiten analizar la velocidad de convergencia y la calidad de las soluciones alcanzadas por cada variante.

5.5.1. Variante Genérica Standard (300 Población, 3000 Generaciones)

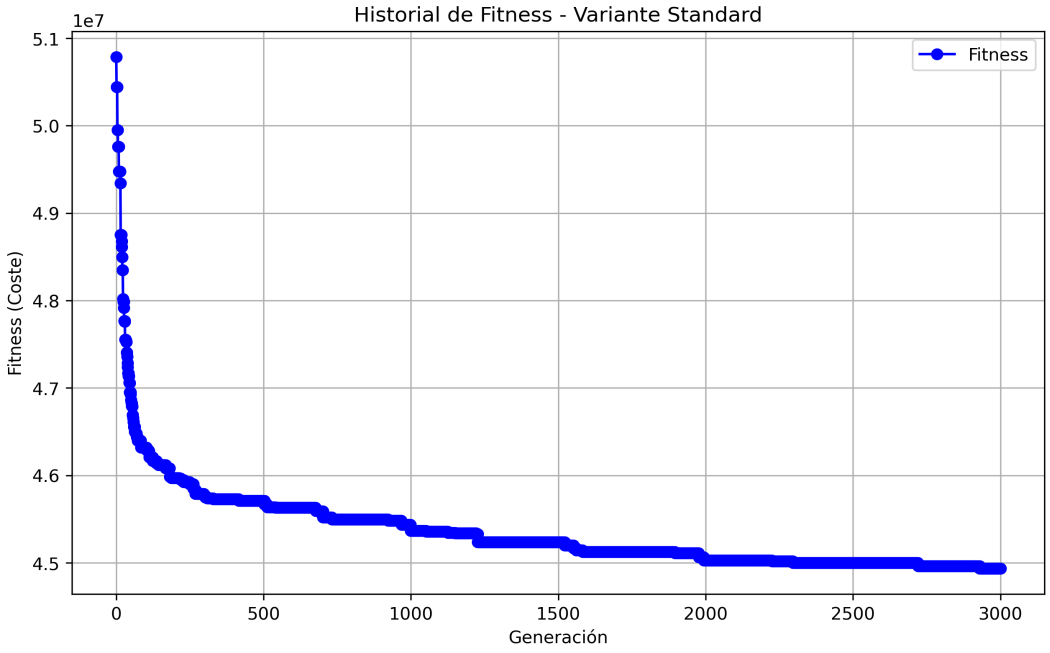

Figura 1: Evolución del Fitness - Variante Genérico Standard.

La variante Genérica Standard muestra una mejora gradual en el fitness durante las 3000 generaciones, con una convergencia lenta pero consistente. Esto permite obtener soluciones cada vez mejores a costa de un mayor tiempo de ejecución.

5.5.2. Variante Genérica Standard (500 Población, 500 Generaciones)

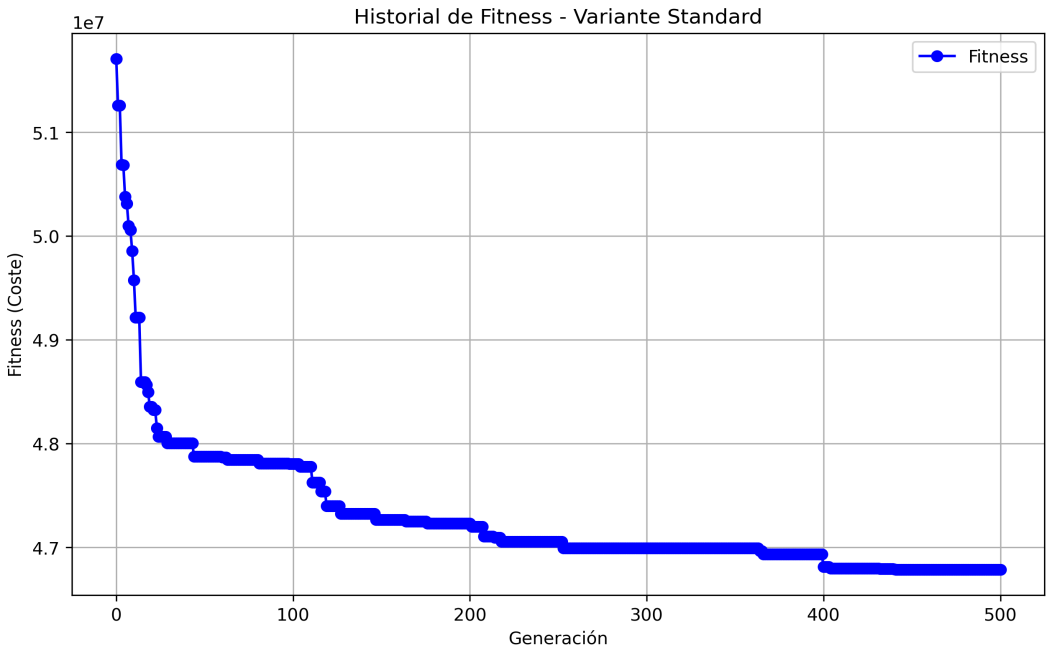

Figura 2: Evolución del Fitness - Variante Genérico Standard.

La variante Genérica Standard muestra una mejora gradual en el fitness durante las 500 generaciones, con una convergencia más rápida que la configuración anterior, permitiendo obtener soluciones aceptables en menos tiempo de ejecución.

5.5.3. Variante Lamarckiana (500 Población, 500 Generaciones)

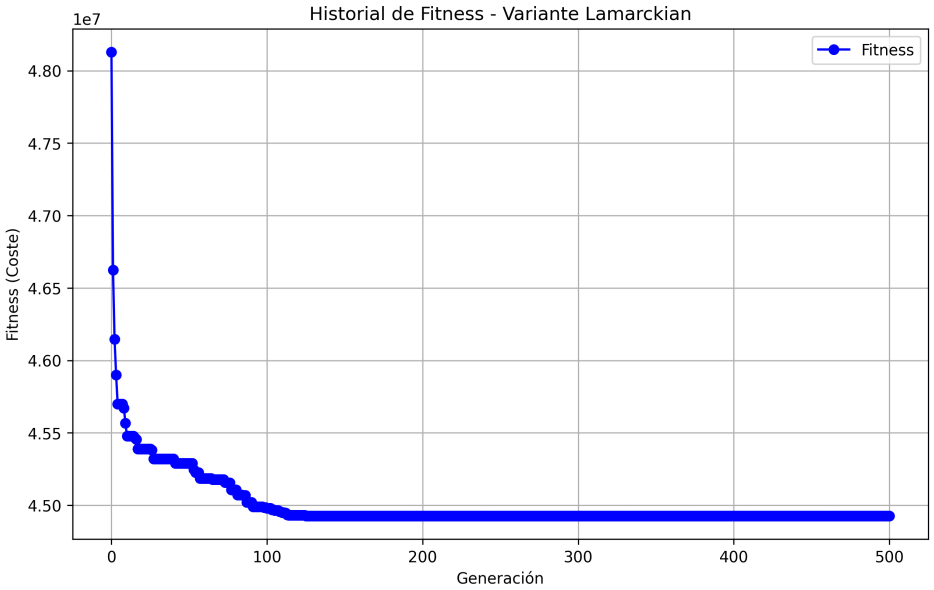

Figura 3: Evolución del Fitness - Variante Lamarckiana.

En la variante Lamarckiana, se observa una rápida convergencia en menos de 100 generaciones, estabilizándose cerca de un fitness de $4{,}5 \times 10^7$. Este comportamiento se debe a la incorporación de optimización local en cada generación, que acelera la búsqueda de soluciones cercanas al óptimo.

5.5.4. Variante Baldwiniana (500 Población, 500 Generaciones)

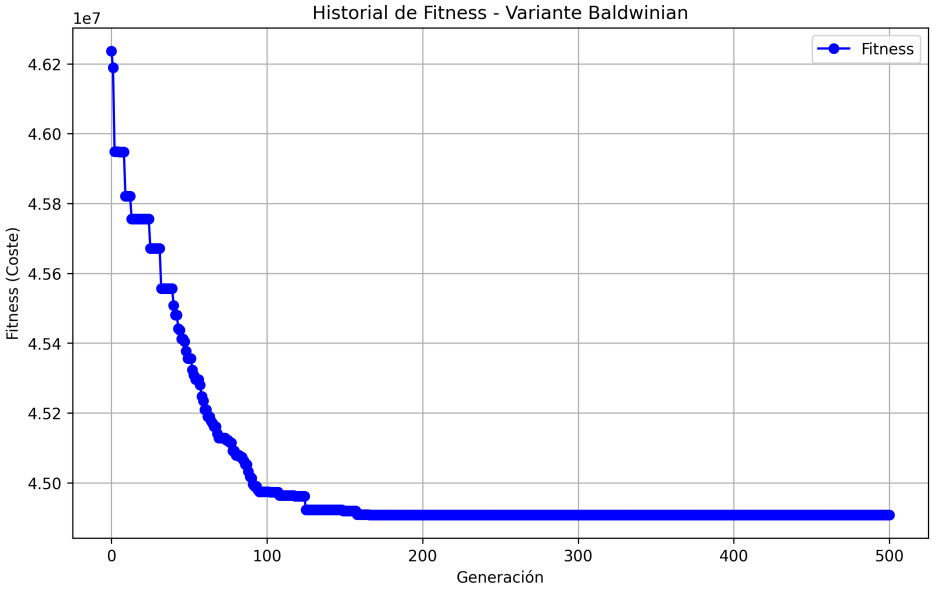

Figura 4: Evolución del Fitness - Variante Baldwiniana.

La variante Baldwiniana sigue un comportamiento similar al de la Lamarckiana, con una rápida mejora inicial y estabilización alrededor de $4{,}5 \times 10^7$. Sin embargo, debido a que la optimización local se aplica solo durante la evaluación del fitness, la convergencia es ligeramente más lenta.

5.5.5. Variante Baldwiniana Mejorada (50 Población, 500 Generaciones)

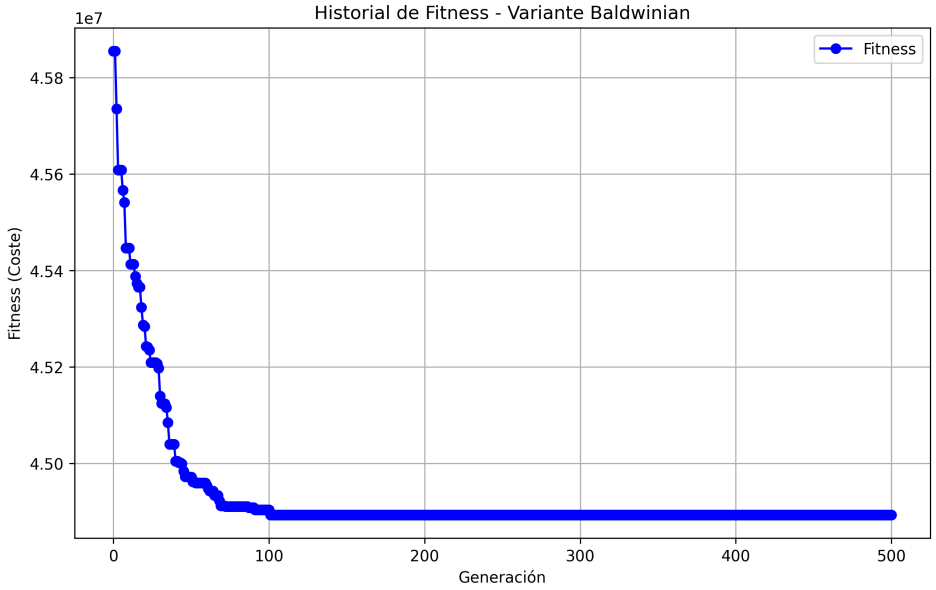

Figura 5: Evolución del Fitness - Variante Baldwiniana Mejorada.

La variante Baldwiniana Mejorada con solo 50 individuos en la población y 500 generaciones consigue la mejor solución global, alcanzando un fitness de $4{,}49 \times 10^7$**.** Esto demuestra que un tamaño reducido de población, combinado con una estrategia eficiente de optimización local, puede superar a configuraciones con poblaciones mayores.

5.5.6. Comparación General

Las gráficas permiten observar las siguientes diferencias:

- La variante Genérica Standard mejora gradualmente durante 3000 generaciones, pero su velocidad de convergencia es significativamente menor que la de las variantes Baldwiniana y Lamarckiana.
- Las variantes Lamarckiana y Baldwiniana (500 población) convergen rápidamente en menos de 100 generaciones, con un fitness final cercano a $4{,}5 \times 10^7$.
- **La variante Baldwiniana Mejorada (50 población) destaca al combinar rapidez de convergencia y el mejor fitness global, logrando la solución óptima** ($4{,}49 \times 10^7$)**.**

Estas diferencias resaltan la importancia de ajustar el tamaño de la población y las estrategias de optimización local para obtener un equilibrio entre coste computacional y calidad de las soluciones.

5.6. Conclusiones

En este estudio comparativo se evaluaron tres variantes del algoritmo genético (Genérico, Baldwiniano y Lamarckiano) aplicadas al Problema de Asignación Cuadrática (QAP). Se analizó el impacto

de diferentes tamaños de población y números de generaciones en la calidad de las soluciones y en el coste computacional.

Los resultados, resumidos en la Tabla 1, indican que:

- **Algoritmo Genético Genérico:** La configuración Genérico4 (300 poblaciones y 3000 generaciones) obtuvo un fitness promedio de 44 905 542, seguida de Baldwiniana3 (44 907 876) y Lamarckiano3 (44 926 294). Sin embargo, **la mejor solución global fue obtenida por la variante Baldwiniana Mejorada con 50 individuos en la población y 500 generaciones, alcanzando un fitness de** $4{,}49 \times 10^7$.
- **Impacto del Tamaño de Población:** Aumentar el tamaño de la población en el Algoritmo Genético Genérico mejoró significativamente la calidad de las soluciones, pasando de un fitness promedio de 46 785 484 con 50 individuos a 45 606 166 con 500 individuos. Esto sugiere que una mayor población facilita una mejor exploración del espacio de búsqueda y reduce la probabilidad de converger en óptimos locales.
- **Configuración Estándar:** La configuración estándar del Algoritmo Genético Genérico (300 poblaciones y 3000 generaciones) logró un equilibrio óptimo entre coste computacional y calidad de las soluciones, con un tiempo de ejecución aproximado de 1 hora y 45 minutos. Esta configuración mostró un rendimiento comparable o incluso superior al de las variantes Baldwiniana y Lamarckiana, que operaron con configuraciones de población y generaciones más pequeñas.

Además, las tendencias observadas hasta ahora permiten concluir que:

- **Calidad de las Soluciones:** Las variantes que incorporan optimización local (Baldwiniana y Lamarckiana) tienden a generar soluciones de mayor calidad en configuraciones equivalentes de población y generaciones. Sin embargo, el Algoritmo Genético Genérico con una población mayor y más generaciones (Genérico4) puede superar a estas variantes en términos de fitness promedio. **No obstante, la variante Baldwiniana Mejorada (50 población, 500 generaciones) proporciona la mejor calidad de solución global.**
- **Coste Computacional:** La inclusión de optimizaciones locales incrementa el coste computacional, siendo la variante Lamarckiana la más costosa debido a la actualización directa de los individuos. No obstante, este coste adicional se compensa con una mayor calidad de las soluciones y una convergencia más rápida.
- **Balance entre Población y Generaciones:** Es crucial encontrar un equilibrio adecuado entre el tamaño de la población y el número de generaciones para optimizar tanto la calidad de las soluciones como el coste computacional. Un tamaño de población mayor favorece una mejor exploración, mientras que un mayor número de generaciones permite una convergencia más precisa.
- **Configuración Estándar Efectiva:** La configuración estándar del Algoritmo Genético Genérico demostró ser efectiva para comparar de manera justa el rendimiento con las variantes Baldwiniana y Lamarckiana, proporcionando soluciones de alta calidad sin incurrir en costes computacionales prohibitivos.

Para futuras investigaciones, se recomienda:

- **Completar Datos Pendientes:** Finalizar los valores faltantes en la Tabla 1 para obtener una visión completa y precisa de todas las variantes evaluadas.
- **Explorar Configuraciones Adicionales:** Investigar configuraciones adicionales de tamaño de población y número de generaciones que podrían optimizar aún más el balance entre calidad de las soluciones y coste computacional.

- **Evaluación en Otros Problemas:** Probar estas variantes en diferentes tipos de problemas de optimización para validar y generalizar los hallazgos obtenidos en este estudio.
- **Optimización de Estrategias Locales:** Profundizar en las estrategias de optimización local utilizadas en las variantes Baldwiniana y Lamarckiana para identificar mejoras que puedan reducir el coste computacional sin comprometer la calidad de las soluciones.

En conclusión, la elección de la variante del algoritmo genético y sus parámetros de configuración debe considerar tanto la calidad de las soluciones requeridas como las restricciones de tiempo y recursos computacionales disponibles. **La variante Baldwiniana Mejorada con 50 individuos en la población y 500 generaciones ofrece la mejor calidad de solución global, siendo adecuada para aplicaciones donde la precisión es crítica y el coste computacional es manejable.** Por otro lado, las variantes **Baldwiniana** y **Lamarckiana** proporcionan un buen equilibrio entre calidad y coste computacional, siendo ideales para escenarios donde se busca optimizar rápidamente sin incurrir en altos costes.