UNIVERSIDAD
DE GRANADA

Práctica 2

Algoritmos evolutivos

Javier Martín Alcalde
20995539 N
jmartinalcalde@correo.ugr.es

Enero 2025

1. Problema de la asignación cuadrática

El problema de la asignación cuadrática (QAP) es un problema de optimización combinatoria que involucra la asignación de n instalaciones a n ubicaciones de tal manera que se minimice la suma de los productos de las distancias euclidianas entre las ubicaciones asignadas y los flujos entre las instalaciones asignadas. En otras palabras, el objetivo del problema es minimizar la distancia total recorrida por los flujos entre instalaciones asignadas, teniendo en cuenta las distancias entre las ubicaciones asignadas.

El problema se formula mediante dos matrices: la matriz de flujo, que representa el flujo que se traslada entre las distintas instalaciones, y la matriz de distancia, que representa las distancias entre las ubicaciones. La asignación de las instalaciones a las ubicaciones se representa mediante una matriz de permutaciones, donde cada fila y cada columna contiene un único valor igual a 1 y el resto son ceros. Cada valor distinto de cero en la matriz P indica la ubicación asignada a la correspondiente instalación.

El QAP es NP-completo, está considerado como uno de los problemas de optimización combinatoria más complejos. Debido a esto, es común utilizar algoritmos metaheurísticos para encontrar soluciones aproximadas de buena calidad.

1.1. Consideraciones comunes

Representación de soluciones: La solución a este problema se representa mediante una matriz de permutación de tamaño nxn, donde n es el número de instalaciones y ubicaciones a considerar. Esta matriz contiene un 1 en cada fila y columna, y el resto de valores son 0. Cada valor distinto de cero indica la ubicación asignada a la correspondiente instalación. Por tanto, hay n! posibles soluciones.

Función objetivo: La función objetivo a minimizar en este problema se puede expresar como la suma de los productos de los flujos entre instalaciones y las distancias entre las ubicaciones asignadas. Es decir, se busca minimizar la siguiente expresión:

```
1: suma ← 0
2: for i ← 1 hasta n do
3:     for j ← 1 hasta n do
4:         suma ← suma + f[i][j] * d[P[i]][P[j]]
5:     end for
6: end for
7: return suma
```

2. Algoritmo genético generacional

Los algoritmos genéticos son una técnica de optimización inspirada en la evolución biológica. Se basan en la generación de poblaciones de soluciones y su evolución mediante mecanismos de selección, cruce y mutación. En este apartado se va a describir el primer algoritmo que se ha realizado en esta práctica, el algoritmo genético generacional. El pseudocódigo del algoritmo es el siguiente:

```
1: Inicializar la población de soluciones aleatorias P
2: Evaluar cada individuo en la población
3: evaluaciones ← tam_poblacion
4: while evaluaciones < criterio_parada do
5:     Seleccionar padres mediante torneo
6:     Aplicar cruce con probabilidad p_c
7:     Aplicar mutación con probabilidad p_m
8:     Evaluar nuevos individuos
9:     Reemplazar la población con los mejores individuos
10:    Actualizar el número de evaluaciones
11: end while
12: Devolver la mejor solución encontrada
```

Se comienza generando la población inicial de forma completamente aleatoria, siendo cada individuo una permutación válida de las asignaciones. A continuación se evalúa la población generada, es decir, se calcula el coste de las permutaciones para cada individuo.

El siguiente paso es elegir los 'padres' que generarán la siguiente generación. Para ello se utiliza la selección por torneo. Se seleccionan dos individuos de la población de forma aleatoria y nos quedamos con el que tenga menor coste. Este tipo de selección introduce un equilibrio entre explotación y exploración del espacio de búsqueda. La explotación se debe a que favorece a las soluciones de mejor calidad, mientras que la exploración se mantiene al permitir que individuos con peor desempeño aún tengan una oportunidad de ser seleccionados.

Una vez seleccionados los padres se crean los hijos. Para ello, se ha implementado dos operadores de cruce:

- Cruce basado en posición: Este operador mantiene las posiciones de los elementos comunes en ambos padres y asigna los valores restantes de manera aleatoria. Su objetivo es preservar parte de la estructura de las soluciones originales, reduciendo la posibilidad de generar individuos con permutaciones poco prometedoras.
- Cruce PMX (Partially Mapped Crossover): Este método selecciona una porción de genes de un padre y la intercambia con la correspondiente en el otro padre, asegurando que la permutación resultante sea válida. Además, mantiene una correspondencia entre los valores intercambiados, lo que permite conservar relaciones estructurales en la asignación.

Esta operación depende del parámetro del algoritmo 'probabilidad de cruce', lo que significa que no todos los padres se cruzan en cada generación. Esto se hace para mantener el equilibrio entre la conservación de buenas soluciones y la introducción de nuevas combinaciones en la población.

Para evitar la convergencia a soluciones que no son óptimas, se aplica una mutación por intercambio con una probabilidad de mutación. Este operador selecciona dos posiciones aleatorias dentro de un individuo y las intercambia, generando una nueva permutación. El objetivo de la mutación es permitir la exploración de áreas del espacio de búsqueda que de otro modo no se alcanzarían. Sin embargo, la tasa de mutación debe ser controlada cuidadosamente puesto que una tasa demasiado alta introduce demasiada aleatoriedad y puede deshacer mejoras obtenidas a lo largo de las generaciones.

2.1. Variante baldwiniana

La variante baldwiniana del algoritmo genético introduce un mecanismo de aprendizaje local que permite mejorar la calidad de los individuos antes de ser evaluados, sin modificar directamente su material genético. En esta variante, cada individuo se somete a un proceso de búsqueda local antes de ser evaluado, lo que permite que su rendimiento mejore temporalmente y afecte su probabilidad de ser seleccionado para la reproducción. Sin embargo, la estructura genética original del individuo no se altera, por lo que las mejoras adquiridas no se heredan directamente a la descendencia.

En esta variante, antes de calcular el fitness de un individuo, se aplica un proceso de búsqueda local que explora soluciones cercanas con la intención de encontrar una permutación de menor coste. En este caso, la búsqueda local empleada sigue un esquema basado en intercambio de posiciones, donde se intentan intercambiar pares de elementos dentro de la permutación para minimizar la función objetivo.

2.2. Variante lamarckiana

La variante lamarckiana del algoritmo genético introduce un enfoque de aprendizaje heredable, en el que los individuos mejoran su desempeño mediante un proceso de búsqueda local, y las mejoras adquiridas se transmiten directamente a la siguiente generación. En esta variante, los individuos no solo utilizan la búsqueda local para mejorar su evaluación en el proceso de selección, sino que sus modificaciones se incorporan permanentemente a su material genético, asegurando que sus descendientes partan de soluciones ya optimizadas.

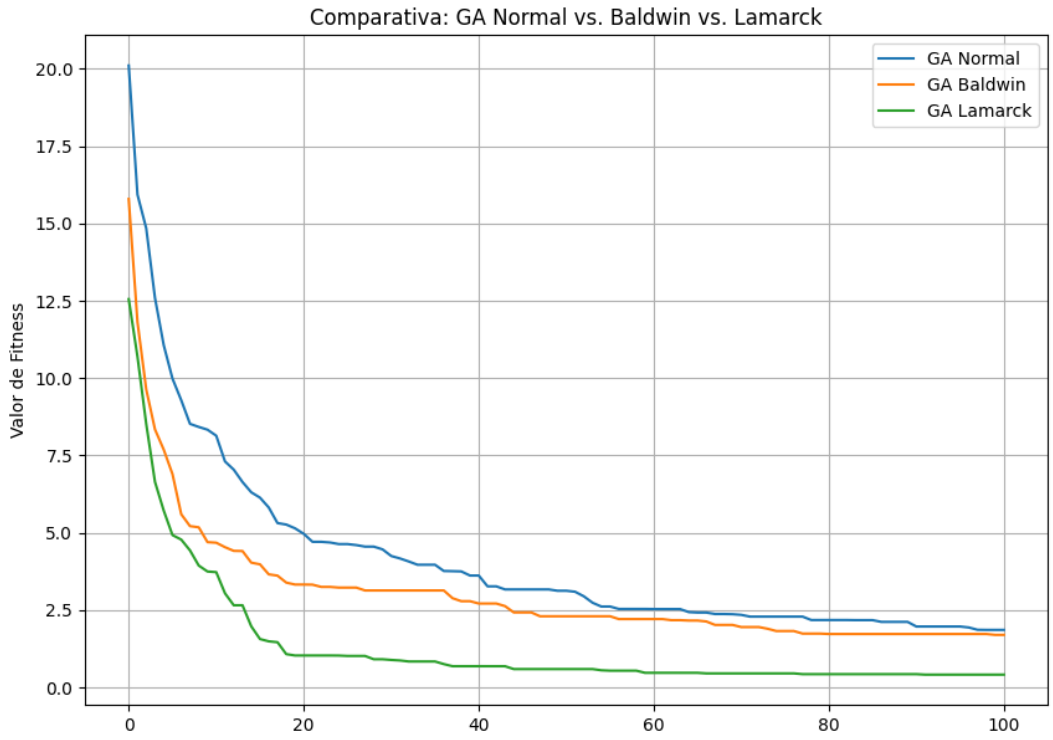
2.3. Comparación de variantes

Para comparar el comportamiento de las variantes se han ejecutado utilizando los mismos parámetros: Tamaño de población de 50, cruce de posición, 0.7 de probabilidad de cruce y 0.1 de probabilidad

de mutación. Los valores finales de fitness obtenidos tras la ejecución de cada algoritmo han sido los siguientes:

- Algoritmo genético normal: fitness = 1.85
- Variante baldwiniana: fitness = 1.69
- Variante Lamarckiana: fitness = 0.41

Se observa que la variante lamarckiana obtiene los mejores resultados finales, reduciendo significativamente el valor del fitness en comparación con el algoritmo estándar y la variante baldwiniana. El siguiente gráfico muestra la evolución del fitness a lo largo de las iteraciones para cada uno de los algoritmos:



Observando el gráfico podemos sacar varias conclusiones. La aplicación de la búsqueda local se nota sobre todo en las primeras iteraciones, convergiendo ambas variantes considerablemente más rápido que el algoritmo normal. Sin embargo, a medida que avanzan las iteraciones la variante baldwiniana tiende a igualarse con el algoritmo normal, mientras que la variante lamarckiana sigue marcando la diferencia.

En cuanto al tiempo de ejecución, cabe destacar que el algoritmo normal necesita entre 30 y 60 segundos para alcanzar el máximo de iteraciones establecido, mientras que ambas variantes necesitan entre 1 y 2 horas. Sabiendo esto podemos llegar a la conclusión de que la variante baldwiniana no merece la pena en la mayoría de los casos pues requiere demasiado tiempo de cómputo y obtiene resultados mejores pero similares al algoritmo normal.

Estas ejecuciones se han realizado con un máximo de iteraciones de 50000 puesto que el objetivo es encontrar una solución con el menor fitness (y por tanto, coste) posible. Viendo el gráfico podemos observar que a partir de la mitad de la ejecución (unas 20-30 mil iteraciones) la mejora del fitness es mínima.

3. Búsqueda de la mejor solución

Como bien indica el guión de esta práctica, uno de los objetivos es conseguir la mejor solución posible. Con el estudio anterior se ha llegado a la clara conclusión de que la variante lamarckiana es la que

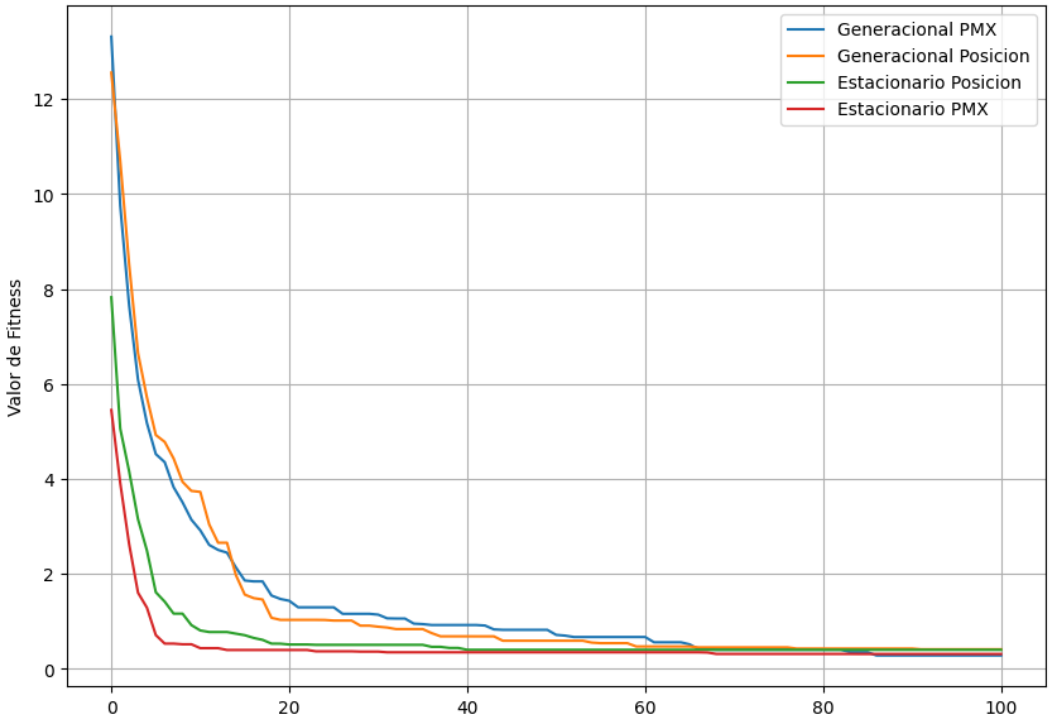
obtiene soluciones con menor coste. En este último apartado he implementado también un algoritmo genético estacionario con la intención de mejorar aún más los resultados obtenidos anteriormente.

El algoritmo genético estacionario realiza reemplazos parciales, es decir, en cada iteración solo se sustituyen unos pocos individuos de la población en lugar de renovarla por completo como hace el generacional. Para ello, se seleccionan dos padres, se generan dos descendientes y estos reemplazan a los peores individuos de la población si tienen mejor fitness. Este enfoque mantiene más diversidad en la población y permite que la evolución sea más gradual, lo que en algunos casos evita la convergencia prematura en óptimos locales. Sin embargo, al introducir menos cambios en cada iteración, puede requerir más generaciones para alcanzar soluciones de alta calidad.

Para realizar el experimento se han ejecutado tanto el algoritmo genético generacional como el estacionario, ambos en su variante lamarckiana, y cada uno con cruce de posición y cruce pmx, Los parámetros utilizados han sido los mismos que antes: Tamaño de población de 50, cruce de posición, 0.7 de probabilidad de cruce y 0.1 de probabilidad de mutación. Los valores finales de fitness obtenidos tras la ejecución de cada algoritmo han sido los siguientes:

- Generacional, cruce posición: fitness = 0.41
- Generacional, cruce PMX: fitness = 0.28
- Estacionario, cruce posición: fitness = 0.40
- Estacionario, cruce PMX: firness = 0.31

El que mejor resultado obtiene es el algoritmo generacional con cruce pmx pero vamos a ver cómo alcanza su solución final cada algoritmo. Al igual que antes, se ha generado un gráfico con la evolución del fitness a lo largo de la ejecución de los algoritmos:



Viendo el gráfico podemos comparar tanto el algoritmo generacional con el estacionario como los diferentes tipos de cruce. En cuanto a los algoritmos, se ve claramente como el estacionario converge mucho más rápido por lo que puede ser una mejor opción si necesitamos una ejecución rápida. Viendo la gráfica, el valor fitness prácticamente no mejora cuando se alcanza el 20 % de las iteraciones máximas.

Si comparamos las soluciones finales, para ambos algoritmos el cruce PMX obtiene un fitness menor que la ejecución con el cruce de posición. Viendo el gráfico podemos afirmar además que en ambos algoritmos (sobre todo en el estacionario) la ejecución con el cruce PMX converge antes.

En cuanto a los tiempos de cómputo, el algoritmo generacional ha necesitado entre 60 y 90 minutos para terminar su ejecución mientras que el algoritmo memético ha necesitado entre 90 y 120 minutos. Es una obviedad que el algoritmo generacional es más rápido pero como en este apartado nuestro objetivo es encontrar la mejor solución posible, no tiene mayor importancia.

4. Entregables

Además de esta memoria, se ha entregado un proyecto C++ para el que he utilizado CLion con la siguiente estructura:

- Ficheros datos.h y datos.cpp: estructuras de datos utilizadas y funciones para leer los datos de partida, la solución, etc.
- Ficheros algoritmo.h y algoritmo.cpp: implementación de los algoritmos.
- Carpeta resultados: ficheros csv con la evolución del fitness para cada ejecución del algoritmo.

También añado a la entrega un cuaderno de google colab .pynb en el que he implementado un script para generar los gráficos que muestran la evolución del fitness a lo largo de la ejecución del algoritmo.