UNIVERSIDAD DE GRANADA

Práctica 2. Algoritmos evolutivos para problemas de optimización combinatoria (QAP).

Inteligencia Computacional

Máster Profesional en Ingeniería Informática

Curso académico 2022/2023

Autor

José Alberto Gómez García

Índice de contenidos

1. Introducción.

En esta práctica trataremos de resolver un problema de optimización combinatoria, concretamente, el problema de la asignación cuadrática (QAP). Estos problemas son NP-completos y hallar una solución óptima mediante el uso de técnicas como la programación dinámica resulta totalmente inviable, como ya pudimos comprobar aquellos que estudiamos el grado en Ingeniería Informática en esta misma facultad.

En estas situaciones, los algoritmos evolutivos suelen resultar de gran utilidad, pues nos permiten obtener resultados bastante buenos en un tiempo que sí es asumible.

Así pues, en esta práctica se desarrollarán tres versiones de algoritmos evolutivos para resolver el problema del QAP. En primer lugar, desarrollaremos una versión "simple", y a partir de esta desarrollaremos otras dos versiones, una que incorpora las ideas de Lamarck y otra que incorpora las ideas de Baldwin.

2. Herramientas utilizadas.

2.1. Hardware.

Para la ejecución de los algoritmos se ha hecho uso de un equipo de sobremesa con un procesador Intel® Core™ i7-8700K y 32 GB de memoria RAM DDR4. A diferencia de la práctica anterior, no se aprovecha el uso de tarjeta gráfica.

2.2. Software.

Para desarrollar el software necesario para la realización de esta práctica se han utilizado Python 3.10.6, ejecutado sobre un sistema operativo Windows 11 Pro.

En esta ocasión, sólo hemos necesitado hacer uso de la biblioteca de *Numpy* (v.1.24), que agrega soporte para arrays y matrices multidimensionales, así como colecciones de funciones matemáticas. Su uso resulta fundamental para optimizar los cálculos a realizar. El resto de las bibliotecas, como *sys* y *Tkinter* (para un pequeño menú de selección de archivos), *math* y *random* se encuentran instaladas por defecto.

3. Lectura de datos.

Los ficheros de datos que utilizaremos para poner a prueba nuestros algoritmos se han obtenido de una biblioteca QAPLIB, a la cuál se puede acceder desde este enlace.

Estos ficheros, de extensión *.dat,* contienen un valor entero que indica el tamaño del problema y dos matrices A y B, a las que llamaremos matrices de flujos y distancias. Cada línea de la matriz se encuentra en una línea del fichero, y lo valores están separados por espacios.

Lo primero que deberemos hacer es desarrollar una función, cuyo nombre será "read_data_from_file" que nos permita cargar la información de estos ficheros en nuestro programa. Esta se muestra a continuación:

```python
def read_data_from_file(filename):

    with open(filename, 'r' ) as f:
        lines = f.readlines()
        lines = [line.split() for line in lines]
        both_matrix = [list(map(int, line)) for line in lines if line != []]    # Quitar líneas en blanco del fichero

        problem_size = int(lines[0][0])     # Primer elemento de la primera línea es el tamaño del problema
        # Separar ambas matrices se puede hacer con slicing fácilmente.
        flow_matrix = both_matrix[1:problem_size+1]
        distance_matrix = both_matrix[problem_size+1:]

        return problem_size, np.asfarray(flow_matrix), np.asfarray(distance_matrix)
```

Imagen 1. Función para leer datos.

La función recibe como parámetro el nombre del fichero del que leer los datos. Con este abierto, cargamos todo su contenido, de manera que el tamaño del problema será el primer valor que se encuentre en la primera línea. Posteriormente, eliminamos las líneas en blanco de todo el fichero, de manera que podamos cargar ambas matrices haciendo uso de *"list comprehension"* y separarlas mediante *"array slicing"* conocido el tamaño del problema. Finalmente, devolvemos el tamaño del problema como un entero y las matrices como arrays de *Numpy*.

Para evitar problemas con las rutas en función del sistema operativo, el programa principal incorpora un selector de archivos mediante GUI. Este programa principal también es el encargado de preguntar por el algoritmo a ejecutar. Los parámetros para cada algoritmo deben cambiarse en el código de este programa.

4. Métodos comunes de los algoritmos evolutivos.

Dado que las tres variantes de algoritmos evolutivos que vamos a desarrollar comparten los mismos métodos básicos, los agruparemos en una clase llamada "*AlgoritmoGenetico*", y posteriormente crearemos tres clases por herencia en las que se reflejen las diferencias de comportamiento entre estos algoritmos.

En el constructor de esta clase base, y por tanto en el de las hijas, deberemos aportar el tamaño del problema, las matrices obtenidas anteriormente, y parámetros relativos a la ejecución del algoritmo como el número de individuos de la población inicial, la probabilidad de cruce, la probabilidad de mutación o el porcentaje de la población que participará en la selección por torneo.

En las siguientes secciones se detalla la implementación de los métodos comunes.

4.1. Inicialización de la población.

La población inicial se genera de manera aleatoria, de manera que creamos tantos individuos como hayamos especificado en el constructor.

Los individuos se representan como un vector con tantas posiciones como tamaño tenga el problema. Cada elemento del vector (cromosoma) indica el índice de la localización a la que se va a viajar, por lo que deberemos tener una única aparición de cada valor numérico en el rango [0, tamaño del problema – 1]. Esto lo conseguimos haciendo uso de la función *permutation* de la librería *Numpy*.

El código correspondiente a la función se muestra a continuación:

```
def inicializarPoblacion(self, sizepop=None):
    if sizepop is None:
        sizepop = self.num_individuos_inicial
    taman = self.tamanio_problema  # Longitud del vector igual al número de localizaciones
    poblacion = [np.random.permutation(taman) for _ in range(sizepop)]
    return poblacion
```

Imagen 2. Inicializar población.

4.2. Función de coste (fitness).

Una vez hemos generado la población, tenemos que calcular el coste de cada uno de los individuos. Este coste viene representado por la expresión matemática:

$$\sum_{i,j} w(i,j)d(p(i),p(j))$$

donde *d(i,j)* hace referencia a la distancia entre una localización *i* y una localización *j*; *w(i,j)* hace referencia al peso asociado al desplazamiento entre la posición *i* y la posición *j*, y p() define una permutación sobre el conjunto de localizaciones.

Implementado en Python, el cálculo de este coste corresponde a la expresión dentro de la función *"fitnessIndividuo",* como se puede observar en la imagen 3. Se añade otra función para calcular el coste de todos los individuos, dado que esta se tendrá en cuenta en la fase de selección de individuos.

```
def fitnessIndividuo(self, indv):
    return sum(np.sum(self.matriz_pesos * self.matriz_distancias[indv[:, None], indv], 1))

def fitnessTodaPoblacion(self, poblacion):
    return [(indv, self.fitnessIndividuo(indv)) for indv in poblacion]
```

Imagen 3. Cálculo de costes

4.3. Función de mutación.

La mutación permite introducir diversidad en nuestra población, al alterar los genomas de algunos individuos concretos, aunque se produzca en una baja probabilidad.

Debemos tener en cuenta que la mutación puede no ser siempre ventajosa; habrá situaciones en la que el coste asociado al individuo no varíe y ocasiones en las que aumente.

En mi caso, he decidido implementar un proceso de mutación en el que se intercambian los valores de dos elementos del vector que representa a un individuo elegidos aleatoriamente. Puede ser interesante, en especial en problemas de gran tamaño como el que debemos afrontar, que cuando se produzcan mutaciones se intercambie más de una pareja de elementos, por lo que se ha parametrizado este valor de manera que el usuario pueda elegir según le convenga.

En imagen 4 se muestra el código de la función responsable del proceso de mutación.

```
def mutacion(self, indv):
    if self.prob_mut >= random.random():    # Rango [0,1]. Tener en cuenta al definir probabilidad
        for _ in range(self.num_mutaciones):
            # Número de mutaciones parametrizables, para problemas grandes puede interesar sea mayor.
            cromo1 = int(random.choice(indv[0]))
            cromo2 = int(random.choice(indv[0]))
            indv[0][cromo1], indv[0][cromo2] = indv[0][cromo2], indv[0][cromo1]
    return indv
```

Imagen 4. Función de mutación

4.4. Operador de cruce.

El operador de cruce es el mecanismo del que disponemos para generar nuevos individuos a partir de una pareja de padres. Existen multitud de maneras de implementar este operador, pero en mi caso he decidido utilizar una de las más simples, la variante de cruce en un punto.

Esta variante elige un punto, de manera que desde el inicio del vector hasta ese punto se toma la información genética del padre A, mientras que desde ese punto al final se toma la información genética del padre B. Dado que cada pareja de padres genera una pareja de hijos, también sucederá el caso contrario.

El punto del vector en el que se realiza esa división podría delimitarse de manera estática, por ejemplo, en la mitad del vector. Sin embargo, he decidido que ese punto de corte se determine también de manera aleatoria para cada cruce, en un intento de imitar más fielmente al funcionamiento del cruce biológico real.

Cabe destacar que no todos los individuos se reproducen, como sucede en la vida real, por lo que aquellos que no lo hagan pasarán a la siguiente generación en lugar de los supuestos hijos que deberían haber tenido.

En la imagen 5 se muestra el código de la función que implementa lo explicado.

```python
def cruce(self, indv1, indv2):
    if self.prob_cruce >= random.random():
        cut = random.randrange(1, len(indv1[0]))  # Punto de corte aleatorio, puede aportar más diversidad y es más "real"

        hijoA = np.concatenate((indv1[0][:cut], indv2[0][cut:]), axis=None)
        hijoB = np.concatenate((indv2[0][:cut], indv1[0][cut:]), axis=None)
        # Resolver problemas de cromosomas repetidos, es probable que le vengan de los padres algunos repetidos.
        hijoA = (self.gestionarCromosomasRepes(hijoA), self.fitnessIndividuo(hijoA))
        hijoB = (self.gestionarCromosomasRepes(hijoB), self.fitnessIndividuo(hijoB))
    # No se cruzan, se devuelven los padres
    else:
        hijoA = indv1
        hijoB = indv2
    # Despues de cruzar, vemos si mutan
    return self.mutacion(hijoA), self.mutacion(hijoB)
```

Imagen 5. Función de cruce.

Por como hemos definido el operador de cruce es probable que los hijos generados tengan cromosomas repetidos y le falten algunos. Recordemos que en cada individuo debe aparecer una única vez cada índice de localización.

Supongamos un problema de tamaño 6, si un hijo recibe del padre A los valores [1,4,2] y del padre B los valores [2,5,6], tendrá dos veces el cromosoma 2 y le faltará el cromosoma 3.

Para resolver esta problemática se hace uso de la función “gestionarCromosomasRepes". Este método lo que hace es buscar los valores de los que hay más de una ocurrencia e irlos sustituyendo por valores de los que no hay ninguna ocurrencia en el individuo. El valor que se asignará en lugar del repetido es escogido aleatoriamente del conjunto de valores posibles.

La implementación de esta función se puede ver en la imagen 6.

```python
def gestionarCromosomasRepes(self, indv):
    for n in range(len(indv)):
        ocurrences = np.count_nonzero(indv == indv[n])
        if ocurrences > 1:
            replaces = [r for r in range(len(indv)) if r not in list(indv)]
            indv[n] = random.choice(replaces)
    return indv
```

Imagen 6. Gestión de cromosomas repetidos.

Finalmente, tras haber generado hijos cuya combinación genética es válida, probamos a ver si se produce una mutación en alguno de ellos. Esto se puede apreciar en la última línea de la imagen 5.

4.5. Operador de selección.

A la hora de realizar la selección de que padres se van a reproducir se ha decidido utilizar un mecanismo de selección por torneo.

El mecanismo de selección implementado, el cual se muestra en la imagen 7, no es completamente elitista. Es decir, no siempre se van a escoger a los mejores padres (aquellos con menor costo asociado) para que se reproduzcan.

```python
def seleccionPorTorneo(self, scores):
    min_score = min(scores, key=lambda x: x[1])[1]
    num_participants = math.ceil(self.porcentaje_torneo * len(scores))  # Escogemos participantes de acuerdo a parámetro
    random.shuffle(scores)
    participants = []
    p = 0
    while len(participants) != num_participants:
        prob_selec = min_score / scores[p][1]
        if prob_selec > random.random():
            participants.append(scores[p])
        p += 1

    return list(min(participants, key=lambda x: x[1]))
```

Imagen 7. Selección por torneo.

En su lugar, cada padre tendrá una probabilidad asociada a su coste; de manera que su probabilidad de ser seleccionado para reproducirse será mayor cuanto menor sea su coste asociado. Esto permite que algún padre "de mitad de la tabla" pueda ser seleccionado también, aportando posible diversidad. La función matemática empleada es la siguiente:

$$Probabilidad\ de\ selección = \frac{Menor\ coste\ en\ la\ población}{Coste\ del\ individuo}$$

El número de participantes en la selección es un valor parametrizable que se deja a elección del usuario, pero que obviamente no puede ser mayor al número total de individuos de la población.

Cabe destacar que padres e hijos pasan a la siguiente generación, o iteración del algoritmo. Los individuos que no se reproducen son reinicializados, haciendo uso de la función mostrada en la imagen 2.

5. Algoritmo evolutivo simple.

5.1. Implementación.

Para la implementación de este algoritmo evolutivo se ha creado una nueva clase llamada "*AlgoritmoGeneticoSimple",* hija de la clase base que contiene los métodos anteriormente descritos.

En esta nueva clase sólo añadimos un método para ejecutar el algoritmo en sí mismo, cuyo único parámetro es el número de iteraciones a realizar, dado que el resto de los parámetros fueron especificados en el constructor. El código se muestra a continuación.

```python
class AlgoritmoGeneticoSimple(AlgoritmoGenetico):

    def executeAlgorithm(self, num_iter_max=350):
        # Se podría mandar al constructor y tenerlos como atributos, pero así permitimos diferentes poblaciones con un mismo objeto del algoritmo.
        population = self.inicializarPoblacion()
        scores = self.fitnessTodaPoblacion(population)

        num_iter = 0
        mejor_global = (np.zeros(self.tamanio_problema), math.inf)    # La mejor solución global es nula de inicio
        while num_iter < num_iter_max:
            print("Generación: ", num_iter+1)
            parent = [self.seleccionPorTorneo(scores) for _ in range(self.num_padres_torneo)]
            for i in range(len(parent)-1):
                s1, s2 = self.cruce(parent[i], parent[i+1])
                parent.append(s1)   # Se añaden los hijos al final de la lista de los padres.
                parent.append(s2)   # Por reusar un poco el nombre de la variable, pero hace más referencia a la población que se mantiene.
            mejor_hijo = list(min(parent, key=lambda x: x[1]))
            if mejor_hijo[1] < mejor_global[1]:
                mejor_global = mejor_hijo

            new_population = self.inicializarPoblacion(self.num_individuos_inicial - len(parent))
            scores = self.fitnessTodaPoblacion(new_population) + parent
            random.shuffle(scores)

            num_iter += 1
            print("Mejor fitness por ahora:", mejor_global[1], "\n")
            # print("Mejor solución por ahora: ", mejor_global[0])


        print("Mejor combinación al final:", mejor_global)
```

Imagen 8. Algoritmo evolutivo simple.

Lo primero que hacemos es inicializar la población con el número de habitantes deseados, este valor se especificó en el constructor y es un atributo de clase, por lo que no hay que aportarlo; posteriormente calculamos el coste de cada uno de los elementos en esta primera generación.

Dentro de un bucle cuya variable de control es el número de generaciones máximas a generar se realiza todo el proceso de cálculo de la solución. Se seleccionan los padres ganadores del torneo, se reproducen (o no), añaden sus hijos a la población (mutados o no) y determinamos cual es la mejor solución (individuo) por el momento. Después, aquellos padres que no se reprodujeron ni fueron ganadores del torneo mueren y esa parte de la población es reinicializada aleatoriamente, tras lo que se calcula su costo.

5.2. Pruebas y resultados obtenidos.

El tiempo de ejecución de los algoritmos genéticos dependen en gran medida del tamaño de la población que especifiquemos y el número de generaciones que tengan lugar. Sin aplicar aún ninguna técnica de optimización, este algoritmo se ejecuta en un tiempo bastante razonable, por lo que hemos podido realizar varias pruebas.

Las ejecuciones se han realizado sobre el fichero de prueba especificado en el guion de la práctica, *tai256c.* La mejor solución encontrada por el momento para este escenario tiene un coste de **44759294**.

Los resultados obtenidos han sido los siguientes:

N.º individuos por generación	**% cruce**	**% mutación**	**N.º padres por generación**	**N.º iteraciones**	**Resultado**
100	70	10	10	100	48731646
100	70	10	10	500	48544510

300	70	5	20	500	47177428
500	70	5	30	300	46993082
500	80	10	30	300	**46275636**
750	70	5	45	300	46696640
750	80	10	45	300	46466488
1000	70	5	70	300	48035878

Tabla 1. Ejecución del algoritmo evolutivo simple.

Uno de los primeros parámetros sobre el que debemos decidir es el número de individuos que habrá por generación y el número de generaciones.

Para comenzar a obtener resultados rápidamente definimos 100 individuos y 100 generaciones. El coste de la mejor solución descendía rápidamente cada pocas generaciones, por lo que realicé una segunda ejecución con más iteraciones. Sobre la iteración 200 el coste dejó de bajar y se mantuvo hasta el final, con un valor final de **48544510**.

Posteriormente, decidí aumentar el tamaño de la población significativamente hasta los 300 individuos, así como el número de padres que se reproducían. En este caso, reduje el porcentaje de mutación. El número de iteraciones se definió bastante alto (500) para comprobar si se repetía el comportamiento anteriormente mencionado, y así fue. El costo descendía rápidamente cada pocas generaciones, pero a partir de la iteración 150 apenas disminuía, manteniéndose constante a partir de la iteración 250 aproximadamente. Se produjo una mejora cercana al 3% respecto de las ejecuciones con 100 individuos, reduciéndose el coste hasta los **47177428.**

Dado que el coste no parecía reducirse más allá de las 250 iteraciones, se fijó un valor algo superior (300) para el resto de los experimentos. Así pues, continuamos incrementando el tamaño de la población inicial y el número de padres por generación.

Se obtuvieron resultados algo mejores con poblaciones de tamaño 500 y 750, probabilidad de cruce 70% y probabilidad de mutación 5%. Por tanto, se decidió replicar estos experimentos con un mayor porcentaje de cruce y mutación.

Los mejores resultados de la tanda de experimentos se obtuvieron para una población de 500 habitantes, probabilidad de cruce 80%, probabilidad de mutación 10%, 30 padres por generación y 300 iteraciones. El costo obtenido fue **46275636,** el cual es un 3.3% peor que el mejor resultado conocido hasta el momento.

La permutación que da lugar a este costo se puede ver en la imagen 9.

```
Mejor combinación al final: [array([ 55, 220,  43,  96,  22, 134, 230, 235, 216,  85,  10, 237, 123,
   209,  82, 105,  58, 188, 167,  18, 143,  24, 186, 165, 129, 173,
   119,  32, 146, 121,  90, 176,  20, 232,  77, 114, 116,  87, 175,
   196,  29,   0, 178,  70,  73, 182,  46, 136, 223,   6,  35, 252,
   127, 192, 144, 211, 241,  41, 115,   4,  53, 243, 157, 229,  91,
   108,  81,  68, 218, 224, 109, 201,  67,  49, 249, 154,  95, 171,
    64, 184, 199, 139,  60,  13, 205, 148, 226, 180, 102, 141,  31,
   254,   7, 238, 214,  34, 177, 110,   8,  84, 225,  79,  19, 118,
   206,  42, 137, 213, 130, 128,  99, 120, 169, 228,  92,  61,  63,
   126,  59,  33, 158, 152, 193,  98, 156,  52,  86,  54, 221,  37,
   231,   1, 215, 240,  72, 147, 246, 185, 138, 117,  80,  16, 113,
     9,  47,   3, 112, 207, 149, 131, 101, 208,  76, 255,  50, 239,
   187,  39,  65, 166, 227, 183, 142,  88, 163,  30, 234,  26, 236,
    89,  28, 181, 164, 122, 210, 159, 111,   5, 133,  83,  57, 104,
    48,  38, 194, 100, 222, 198,  40, 103, 145, 151, 107,  21,  44,
   217, 106,  23,  27,  36, 162, 140, 247, 172,  14, 204, 253, 161,
   153,  11,  15, 251, 125,  69, 242,  71,  75, 135, 244, 191, 233,
     2,  51, 248, 160, 124,  97,  78,  12,  93, 132, 189, 150, 155,
   203, 200,  94,  62, 245, 170, 174,  74,  45, 179,  25, 212, 219,
    66, 195, 190, 250, 197, 202,  56, 168,  17]), coste = 46275636.0]
```

Imagen 9. Mejor permutación obtenida haciendo uso del algoritmo evolutivo simple.

Cabe destacar que ninguna de las ejecuciones que han tenido lugar en este apartado han necesitado de más de 2 minutos de tiempo de procesamiento, aproximadamente.

6. Algoritmo evolutivo baldwiniano.

6.1. Implementación.

Para la implementación de este algoritmo evolutivo se ha creado una nueva clase llamada "*AlgoritmoGeneticoBaldwiniano*". Esta clase hereda de la clase base y añade el método que implementa la técnica de optimización haciendo uso de la heurística "*greedy*" con estrategia de transposición.

El código que se muestra a continuación corresponde con la implementación del pseudocódigo para un algoritmo "*greedy*" basado en 2-opt que se muestra en el guion de la práctica.

```python
def opt_greedy_baldwiniana(self, scores):
    print("Ejecución greedy. This may take a while...")
    for x in range(len(scores)):
        # print("I am doing calculation ", x+1, " of ", len(scores))
        ind = scores[x][0]
        best = ind.copy()
        for i in range(len(ind)):
            for j in range(i + 1, len(ind)):
                if i != j:
                    T = best.copy()
                    T[i], T[j] = T[j], T[i]
                    if self.fitnessIndividuo(T) < self.fitnessIndividuo(best):
                        best = T
        scores[x] = (ind, self.fitnessIndividuo(best))
    # print("Finalizado Greedy Baldwin")
    return scores
```

Imagen 10. Implementación optimización greedy I.

Si nos fijamos, la actualización de costes que tiene en la penúltima línea de código no actualiza al individuo en sí, porque lo que este no incluye las mejoras aprendidas durante este proceso.

En la imagen 11 se muestra el código para la ejecución de esta variante del algoritmo. La única diferencia que presenta respecto del código de la imagen 8, correspondiente a la versión simple del algoritmo, es la inclusión del cálculo de la optimización.

```python
def executeAlgorithm(self, num_iter_max=10):
    population = self.inicializarPoblacion()
    scores = self.fitnessTodaPoblacion(population)
    scores_baldwin = self.opt_greedy_baldwiniana(scores.copy())

    num_iter = 0
    mejor_global = (np.zeros(self.tamanio_problema), math.inf)
    while num_iter < num_iter_max:
        print("Generación: ", num_iter+1)
        next_gen = [self.seleccionPorTorneo(scores_baldwin) for _ in range(self.num_padres_torneo)]
        print("Tamaño de la población: ", len(next_gen))
        for i in range(len(next_gen) - 1):
            s1, s2 = self.cruce(next_gen[i], next_gen[i + 1])
            next_gen.append(s1)
            next_gen.append(s2)
        mejor_hijo = list(min(next_gen, key=lambda x: x[1]))
        if mejor_hijo[1] < mejor_global[1]:
            mejor_global = mejor_hijo
            print("Mejor fitness por ahora: ", mejor_global[1])
            print("Mejor individuo por ahora: ", mejor_global[0])

        population = self.inicializarPoblacion(self.num_individuos_inicial - len(next_gen))
        scores = self.fitnessTodaPoblacion(population) + next_gen
        scores_baldwin = self.opt_greedy_baldwiniana(scores)
        random.shuffle(scores_baldwin)

        num_iter += 1

    print("Mejor combinación al final:", mejor_global)
```

Imagen 11. Algoritmo evolutivo baldwiniano.

6.2. Pruebas y resultados obtenidos.

El proceso de optimización *"greedy"* es un proceso de eficiencia cuadrática. Su inclusión en el algoritmo tiene un alto impacto computacional, por lo que se han tenido que utilizar valores de los parámetros mucho más bajos y se han podido realizar menos ejecuciones.

Los resultados obtenidos a lo largo de las ejecuciones se resumen en la siguiente tabla:

N.º individuos por generación	**% cruce**	**% mutación**	**N.º padres por generación**	**N.º iteraciones**	**Resultado**
12	70	5	2	5	45121612
15	75	10 (*)	2	3	45070488
15	80	5 (*)	2	3	45175234
20	80	5	2	5	45047070
30	75	10	2	5	**44981380**

Tabla 2. Ejecución del algoritmo evolutivo baldwiniano.

NOTA: en todas las ejecuciones de los algoritmos se establece que, en caso de haber mutación, se intercambien dos parejas de valores. En el caso de las ejecuciones marcadas con (*) se intercambiaron 4 y 3 parejas respectivamente.

Comenzamos haciendo uso de 12 individuos por generación y 5 iteraciones. Este proceso tardó varios minutos y nos arrojó un resultado de 45121612. Este supone una mejora del 2.56% respecto del mejor resultado obtenido con el algoritmo evolutivo simple.

Durante la ejecución del algoritmo observamos que sólo se producían mejoras en las tres primeras generaciones, por lo que en la dos siguientes ejecuciones reducimos el número de iteraciones.

Aumentando ligeramente el tamaño de la población y variando el porcentaje de cruce y mutación conseguimos resultados similares, aunque para el caso con 75% de probabilidad de cruce y 10% de mutación se consiguió reducir el coste ligeramente.

Los dos últimos experimentos se realizaron con poblaciones de mayor tamaño, siendo la ejecución considerablemente más larga (en torno a la hora). El mejor resultado se obtuvo con la población de mayor tamaño, cuyo coste no disminuyó ni en la cuarta ni quinta ejecución. El coste final obtenido fue de **44981380**, el cual es solamente un 0.5% peor que la mejor solución conocida.

Aunque las diferencias parezcan pequeñas, en aplicaciones reales como la distribución de plantas de producción, o diseño de circuitos integrados, esto puede resultar en grandes ahorros en costes.

La permutación que da lugar a este costo se puede ver en la imagen 12. Este coste será el menor conseguido durante toda la práctica.

```
Mejor combinación al final: [array([253, 157, 6, 250, 80, 28, 152, 134, 221, 38, 135, 237, 199, 68, 108,
    64, 217, 215, 60, 207, 177, 92, 100, 122, 106, 70, 236, 154, 113, 10, 126,
    9, 33, 85, 66, 206, 129, 200, 203, 171, 116, 201, 20, 175, 88, 211, 86, 2,
    54, 5, 55, 90, 252, 186, 79, 73, 97, 190, 179, 184, 234, 230, 183, 127, 12,
    107, 11, 49, 19, 228, 51, 181, 248, 169, 130, 222, 195, 99, 44, 142, 50, 178,
    102, 41, 213, 110, 145, 180, 62, 164, 87, 146, 56, 65, 158, 156, 57, 160, 198,
    36, 168, 255, 163, 182, 4, 202, 120, 244, 159, 172, 170, 69, 96, 115, 223, 23,
    52, 229, 153, 91, 77, 214, 8, 242, 83, 139, 140, 196, 246, 232, 192, 34, 245,
    25, 138, 204, 93, 176, 101, 103, 241, 14, 187, 174, 191, 216, 132, 1, 165, 42,
    254, 89, 81, 193, 212, 46, 231, 209, 188, 32, 114, 71, 247, 205, 141, 7, 197,
    189, 24, 82, 208, 155, 133, 185, 111, 0, 94, 17, 233, 3, 47, 251, 143, 220, 13,
    151, 37, 72, 78, 76, 243, 31, 27, 39, 162, 148, 173, 210, 21, 227, 123, 35, 40,
    219, 61, 238, 125, 119, 118, 218, 235, 239, 18, 194, 136, 112, 95, 105, 26, 240,
    84, 149, 224, 22, 45, 167, 67, 161, 147, 75, 29, 16, 104, 225, 249, 137, 150, 58,
    109, 30, 74, 131, 117, 124, 144, 48, 63, 53, 43, 121, 59, 15, 128, 226, 98, 166]), coste = 44981380.0]
```

Imagen 12. Mejor permutación obtenida haciendo uso del algoritmo evolutivo baldwiniano.

Las ejecuciones de este apartado han requerido de un tiempo de ejecución bastante significativo, el cual ha oscilado entre los 5-6 minutos para las poblaciones más pequeñas y los 45 minutos requeridos por el experimento con mayor tamaño de población. De hecho, este último experimento estaba planeado que tuviera 10 iteraciones, pero dado que ni en la cuarta ni quinta se produjeron mejoras, y el resultado era muy cercano al óptimo conocido, se decidió abortar la ejecución.

7. Algoritmo evolutivo lamarckiano.

7.1. Implementación.

Para la implementación de este algoritmo evolutivo se ha creado una nueva clase llamada "*AlgoritmoGeneticoLamarckiano*". Esta clase hereda de la clase base y añade el método que implementa la técnica de optimización haciendo uso de la heurística "*greedy*" con estrategia de transposición.

El código del algoritmo "*greedy*" utilizado es esencialmente el mismo que el mostrado en la figura 10, con la salvedad de que en esta ocasión sí que se salva el individuo con la mejor solución encontrada. Será a partir de estos individuos que se generará la siguiente generación.

```python
def opt_greedy_lamarckiana(self, scores):
    print("Ejecución greedy. This may take a while...")
    for x in range(len(scores)):
        # print("I am doing calculation ", x+1, " of ", len(scores))
        ind = scores[x][0]
        best = ind.copy()
        for i in range(len(ind)):
            for j in range(i + 1, len(ind)):
                if i != j:
                    T = best.copy()
                    T[i], T[j] = T[j], T[i]
                    if self.fitnessIndividuo(T) < self.fitnessIndividuo(best):
                        best = T
        scores[x] = (best, self.fitnessIndividuo(best))
    # print("Finalizado Greedy Lamarck")
    return scores
```

Imagen 13. Implementación optimización greedy II.

El código utilizado para la ejecución del algoritmo es esencialmente el mismo que el mostrado en la imagen 11, con la salvedad de que se utiliza la función de optimización lamarckiana en lugar de la baldwiniana.

```python
def executeAlgorithm(self, num_iter_max=10):
    population = self.inicializarPoblacion()
    scores = self.fitnessTodaPoblacion(population)
    scores_lamarck = self.opt_greedy_lamarckiana(scores.copy())

    num_iter = 0
    mejor_global = (np.zeros(self.tamanio_problema), math.inf)
    while num_iter < num_iter_max:
        print("Generación: ", num_iter+1)
        next_gen = [self.seleccionPorTorneo(scores_lamarck) for _ in range(self.num_padres_torneo)]
        for i in range(len(next_gen) - 1):
            s1, s2 = self.cruce(next_gen[i], next_gen[i + 1])
            next_gen.append(s1)
            next_gen.append(s2)
        mejor_hijo = list(min(next_gen, key=lambda x: x[1]))
        if mejor_hijo[1] < mejor_global[1]:
            mejor_global = mejor_hijo
            print("Mejor fitness por ahora: ", mejor_global[1])
            print("Mejor individuo por ahora: ", mejor_global[0])

        population = self.inicializarPoblacion(self.num_individuos_inicial - len(next_gen))
        scores = self.fitnessTodaPoblacion(population) + next_gen
        scores_lamarck = self.opt_greedy_lamarckiana(scores)
        random.shuffle(scores_lamarck)

        num_iter += 1

    print("Mejor combinación al final:", mejor_global)
```

Imagen 14. Algoritmo evolutivo lamarckiano.

7.2. Pruebas y resultados obtenidos.

Al igual que en el caso del algoritmo baldwiniano, la inclusión de la función de optimización incrementa considerablemente el tiempo de ejecución, por lo que debemos recurrir a tamaños de población pequeños y pocas iteraciones.

Los resultados obtenidos a lo largo de las ejecuciones se resumen en la siguiente tabla:

N.º individuos por generación	**% cruce**	**% mutación**	**N.º padres por generación**	**N.º iteraciones**	**Resultado**
12	70	5	2	5	**45065100**
12	80	10	2	3	45100930
15	75	10 (*)	2	3	45157462
20	80	5	2	5	45146262
24	80	10	2	4	45110452

Tabla 3. Ejecución del algoritmo evolutivo lamarckiano.

NOTA: en todas las ejecuciones de los algoritmos se establece que, en caso de haber mutación, se intercambien dos parejas de valores. En la ejecución marcada con (*) se intercambian 4 parejas de valores.

En esta ocasión los mejores resultados se obtienen precisamente con las ejecuciones que menos números de individuos por generación empleaban. Al igual que en otras ejecuciones, el coste dejaba de decrecer a partir de la tercera iteración. En un intento de mejorar los resultados, se intentó hacer uso de poblaciones más grandes. Estas ejecuciones no consiguieron proporcionar mejores resultados.

Resulta curioso que el algoritmo lamarckiano permite obtener resultados algo mejores para poblaciones pequeñas, mientras que el algoritmo baldwiniano consiguió mejores resultados en poblaciones más grandes. Téngase en cuenta que las poblaciones generadas son aleatorias y las ejecuciones no son totalmente comparables.

El mejor coste obtenido fue de **45065100**, el cual es un 0.68% peor que el mejor resultado conocido para el problema que abordamos. La permutación que nos ha permitido obtener este coste se muestra en la figura 15.

```
Mejor combinación al final:
[array([ 19, 187,  83, 220,  17, 192, 185, 111,  52, 224, 235, 209, 156,
   214, 142,  34, 243,  86, 159, 181, 241, 231,   7, 202, 205, 115,
   237, 196,  79, 170, 211, 255, 248, 233,  94,  66,   4,  14, 125,
   191, 103,   0,  88,  77, 118,  45,  98, 173, 226,  69, 133, 100,
   139, 113, 160, 164, 200, 105, 250,  73,  40,  28, 228, 108,  26,
   178, 252, 150,  55,  49,  58,  22,  60,  80, 147,  37, 128, 130,
   183, 245,  91, 222,   9, 145, 168, 122,  43, 166, 153, 136,  47,
    62, 152,  27,  59, 137,  92, 134,  10,  42,  76,  65, 124, 251,
   207,  56, 184,   5,  99,   8,  90, 121,  84, 109,  36, 215, 206,
   129,  15, 138,  12, 131,  35, 234,  63,  61, 217,  68,  50,  48,
   132,  71,  21, 102, 161, 232,  30, 223, 218, 135, 143,  81,  82,
    75,  16, 112, 242, 110, 195, 175, 227, 188, 239,   3, 238, 182,
    51, 229, 123, 210, 171,  64, 247,  39, 208,  78,  31, 216, 176,
   246, 180,  13,  95, 172,  85, 193, 148,   6, 253, 254,  53,  57,
   204,  72, 236, 225, 127, 163,  70, 157, 146, 116, 186,  67, 240,
   120, 169, 117, 126,  87,  33, 114,  20, 154,  54, 203,  89, 106,
   197, 212, 174,  29,  11, 158,   2, 155, 107, 119,  18, 141, 144,
   189, 149,  25,  23, 213,  38,  32,  46, 140, 104, 221, 162, 244,
    24, 151,  44,   1, 179, 177, 199, 165, 249,  96, 101, 201,  41,
    93, 167,  97, 230, 190, 198, 194,  74, 219]), coste = 45065100.0]
```

Imagen 15. Mejor permutación obtenida haciendo uso del algoritmo evolutivo lamarckiano.

Los tiempos de ejecución de esta versión del algoritmo han sido bastante similares a los que necesitados por la versión baldwiniana, comentados anteriormente. En particular, la ejecución para 24 individuos duró cerca de 32 minutos.

8. Conclusiones.

- En esta práctica se ha abordado el problema de optimización combinatoria QAP, y la utilidad de técnicas como algoritmos evolutivos para obtener soluciones suficientemente buenas en problemas NP-completos.
- Las variantes baldwiniana y lamarckiana, en especial la primera, permiten mejorar los resultados obtenidos por un algoritmo genético simple, mediante la incorporación de heurísticas "*greedy*". Como punto negativo, cabe mencionar que estas provocan un tiempo de ejecución considerablemente mayor. Este tiempo de computación extra puede estar justificado por los ahorros en costes derivados de una mejor solución, por pequeño que parezca el margen de mejora.
- Se ha alcanzado una puntuación solamente un 0.5% peor que la mejor conocida, lo cual consideramos un gran éxito. Para ello, hicimos uso del algoritmo baldwiniano.

Detalles adicionales sobre permutaciones obtenidas, no expuestas en esta memoria, pueden encontrase en el archivo comprimido de la entrega.