



Escuela Técnica Superior de Ingenierías Informática y de Telecomunicaciones
Máster Oficial en Ingeniería Informática

Curso 2020/2021

PRÁCTICA 2 - QAP

Inteligencia Computacional

Breve descripción

Explicación

Autor

Álvaro de la Flor Bonilla

Propiedad Intelectual

Universidad de Granada



RESUMEN

El objetivo de esta práctica es resolver un problema de optimización típico utilizando técnicas de computación evolutiva. Debería implementar varias variantes de algoritmos evolutivos para resolver el problema de la asignación cuadrática, incluyendo como mínimo las variantes que se describen en este guion de prácticas.



1 ÍNDICE

Resumen	1
1 Algoritmo Genético Simple.....	4
1.1 Valores iniciales.....	4
1.2 Población inicial	4
1.2.1 Coste	4
1.3 Mecanismo de selección, cruce y mutación	4
1.3.1 Cruce.....	5
1.3.2 Mutación.....	6
1.4 Reemplazo	8
2 Variante Lamarckiana.....	9
3 Variante Baldwiniana.....	11
4 Resultados	13
4.1 Algoritmo genético simple.....	13
4.2 Algoritmo genético, variante lamarckiana	14
4.3 Algoritmo genético, variante baldwiniana.....	15

ÍNDICE DE ILUSTRACIONES

Ilustración 1 - Operador de cruce.....	5
Ilustración 2 - Debug código	6
Ilustración 3 - Operaciones de mutación.....	6
Ilustración 4 - Código de mutación.....	7
Ilustración 5 - Código de coste de mutación	7
Ilustración 6 - Coste de mutación modificado.....	7
Ilustración 7 - Algoritmo de Greedy.....	9
Ilustración 8 - Greedy Lamarck	10
Ilustración 9 - Aplicación de Greedy.....	10
Ilustración 10 - Notas de especificación.....	11
Ilustración 11 - Greedy para la variante baldwiniana.....	11
Ilustración 12 - Ejecución inicial AGS.....	13
Ilustración 13 - Ejecución final AGS	13
Ilustración 14 - Ejecución de variante lamarckiana	14
Ilustración 15 - Ejecución de variante baldwiniana.....	15

1 ALGORITMO GÉNÉTICO SIMPLE

En este apartado, explicaremos poco a poco el desarrollo de nuestro algoritmo genético para la resolución del problema. Todo el desarrollo se ha realizado utilizando Python como lenguaje principal.

1.1 Valores iniciales

El primer paso que tendremos que hacer es establecer los parámetros de número de cruces, número de mutaciones y la dimensión de nuestro problema.

La dimensión del problema queda establecida por el usuario al iniciar el script.

Los valores para el número de cruces y el número de mutaciones quedan establecidos como derivadas de la probabilidad de cruce y mutación respectivamente, los cuales son elegidos por el usuario al iniciar el script. En concreto, se ha utilizado una fórmula procedente de la documentación adicional que se ha añadido como “*cruce_mutación.pdf*” para el cálculo de estas variables.

```
numero_cruces = ceil(tamano_poblacion / 2.0 * probabilidad_cruce)
numero_mutaciones = ceil(tamano_poblacion * dimension *
probabilidad_mutacion)
```

1.2 Población inicial

El siguiente paso que hemos hecho es construir la población inicial, esta población inicial ha sido construida a partir de un valor aleatorio y calificada posteriormente a partir del coste de cada uno de los cromosomas. El tamaño de este cromosoma es igual al asignado por la dimensión de nuestro problema. Debemos tener en cuenta que además de la generación del cromosoma aleatorio tenemos que calcular el coste asociado para este y añadirlo a la información de nuestra población para agilizar el proceso de calificación posteriormente.

1.2.1 Coste

El cálculo del coste sigue una operación muy sencilla, simplemente multiplicamos la distancia por el coste asociado al flujo de los materiales.

```
def coste(self, cromosoma):
    return sum(np.sum(self.mP * self.mD[cromosoma[:, None],
cromosoma], 1))
```

Esta fórmula se ha utilizado por ejemplo para ordenar la población inicial anterior.

1.3 Mecanismo de selección, cruce y mutación

En nuestro caso, vamos a utilizar un mecanismo de selección basada en torneo y posteriormente realizaremos cruce y mutación. Los pasos serán los siguientes:

1. En primer lugar, tenemos que seleccionar aleatoriamente los participantes elegidos.
2. Creamos los cromosomas “*padre*” y “*madre*” a partir de los pares seleccionados.

3. Crear un conjunto de puntos de cruces aleatoriamente que serán utilizados para elegir qué puntos del cromosoma serán cruzados respecto a los cromosomas “*padre*” y “*madre*” creados anteriormente.
4. Comenzamos el proceso de cruce.
5. Comenzamos el proceso de mutación.

1.3.1 Cruce

En primer lugar, para realizar el proceso de cruce, como siempre tendremos que crear un array vacío donde almacenaremos los nuevos resultados.

Nuestro mecanismo de cruce necesitará de 3 variables:

1. Cromosoma padre
2. Cromosoma madre
3. Puntos de cruce, que realmente lo dividiremos en dos variables para afrontar los dos cromosomas.

El mecanismo de cruce que hemos diseñado consiste en mantener estables (sin modificar) el intervalo que va entre los dos puntos de cruce seleccionados.

Para el resto del cromosoma intercambiaremos los puntos seleccionados utilizando su cromosoma par (si estamos realizando el cromosoma “*padre*” utilizaríamos en ese caso el cromosoma “*madre*” para obtener estos nuevos puntos).

Para obtener esta distribución de cambio utilizaremos una formula muy sencilla que consiste en construir un intervalo de posiciones que irá desde el último punto de corte hasta la dimensión del cromosoma anidándolo desde cero al primer punto de corte. Utilizaremos este intervalo para extraer de forma ordenada cada uno de los elementos del cromosoma par e insertándolo en el cromosoma a cruzar manteniendo el orden establecido por el intervalo expresado anteriormente, siempre y cuando este elemento no se encuentre ya presente en el intervalo invariable. Es decir, se ha intentado seguir el esquema visto en clase:

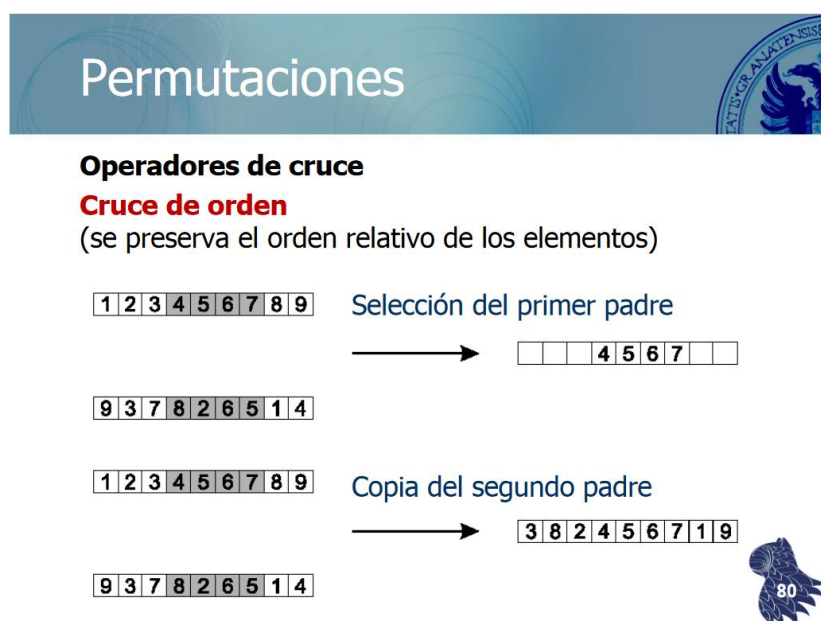


Ilustración 1 - Operador de cruce

Para que se entienda un poco mejor, podemos utilizar el siguiente extracto de nuestro código:

```
# En este apartado lo que vamos a hacer es cruzar a los dos padres
# En concreto vamos a cruzar desde la posición todos los elementos exteriores al intervalo i,j
# Tenemos que tener en cuenta que el nuevo elemento que insertemos no se encuentre ya en el intervalo i,j
child1["cromosoma"][idx] = [elem for elem in parent2["cromosoma"] if elem not in parent1["cromosoma"]][i:j+1]
child2["cromosoma"][idx] = [elem for elem in parent1["cromosoma"] if elem not in parent2["cromosoma"]][i:j+1]

a = child1["cromosoma"][j+1]    a: 161
a_1 = list(parent2["cromosoma"]).index(a)    a_1: 1
b = child1["cromosoma"][j+2]    b: 255
b_1 = list(parent2["cromosoma"]).index(b)    b_1: 2
c = a_1 == b_1 - 1    c: True
```

Ilustración 2 - Debug código

Como vemos, para el primer cromosoma que se ha cruzado, el primer valor inmediatamente a continuación del intervalo invariable corresponde al primer valor del cromosoma par (“parent2”). Consecutivamente se repitió el proceso para el segundo valor posterior al intervalo invariable y efectivamente corresponde al segundo valor del cromosoma par.

1.3.2 Mutación

El proceso de mutación es muy sencillo y una vez más hemos usado un criterio visto en clase:

Permutaciones

Operadores de mutación (1/2)

- **Inserción:** Elegir dos alelos aleatoriamente y colocar el segundo justo después del primero.

1

2

3

4

5

6

7

8

9

→

1

2

5

3

4

6

7

8

9

- **Intercambio:** Seleccionar dos alelos aleatoriamente e intercambiarlos.

1

2

3

4

5

6

7

8

9

→

1

5

3

4

2

6

7

8

9

NOTA: Observe como cada operador de mutación afecta de forma diferente al orden relativo y a las relaciones de adyacencia.




Ilustración 3 - Operaciones de mutación

Se ha realizado, en este caso, una mutación simple basada en el intercambio de valores a partir de dos puntos seleccionados aleatoriamente. Para generar estos índices aleatorios simplemente un array como en las ocasiones anteriores con tamaño igual al doble del número de las mutaciones que se van a realizar ya que se utilizará un mismo array para obtener los índices de intercambio para los dos cromosomas (“padre” y “madre”).

```
def mutar(self, individual, i, j):
    if i > j:
        i, j = j, i

    individual["coste"] = self.coste_mutacion(individual["cromosoma"], individual["coste"], i, j)
    individual["cromosoma"][i], individual["cromosoma"][j] = individual["cromosoma"][j], individual["cromosoma"][i]
```

Ilustración 4 - Código de mutación

Cabe destacar que además de la mutación en sí (intercambio de valores), es necesario realizar un reajuste del coste tras realizar la mutación de los genes, tal y como puede verse en la imagen superior.

Para calcular el coste de una mutación, directamente hemos utilizado lo visto en clase de prácticas y disponible en el siguiente enlace:

https://jamboard.google.com/d/1zhmP863Mku_DDmZx3KEYvmQAo5_yVpxC1vcTxvhZ_zl/viewer

```
def score(self, chromosome):
    return sum(np.sum(self.wmtx * self.dmtx[chromosome[:, None], chromosome], 1))
```

```
def mutationScoreOpt(self, currentChromosome, currentScore, i, j):
    idx = list(range(self.dim)); del(idx[i]); del(idx[j-1])

    mutationScore = currentScore - sum(np.sum(self.wmtx[[i,j], :], self.dmtx[currentChromosome[[i,j], None], currentChromosome], 1))
    mutationScore -= sum(sum(self.wmtx[idx[:, [i,j]] * self.dmtx[currentChromosome[idx, None], currentChromosome[[i,j]]]))

    currentChromosome[i], currentChromosome[j] = currentChromosome[j], currentChromosome[i]

    mutationScore += sum(np.sum(self.wmtx[[i,j], :] * self.dmtx[currentChromosome[[i,j], None], currentChromosome], 1))
    mutationScore += sum(sum(self.wmtx[idx[:, [i,j]] * self.dmtx[currentChromosome[idx, None], currentChromosome[[i,j]]]))

    #revert changes
    currentChromosome[i], currentChromosome[j] = currentChromosome[j], currentChromosome[i]

    return mutationScore
```

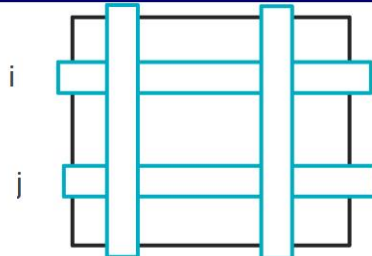


Ilustración 5 - Código de coste de mutación

```
def coste_mutacion(self, cromosoma_actual, coste_actual, i, j):
    newChromosome = np.copy(cromosoma_actual)
    newChromosome[i], newChromosome[j] = newChromosome[j], newChromosome[i]

    coste_mutacion = coste_actual - sum(np.sum(self.mP[[i, j], :] * self.mD[cromosoma_actual[[i, j], None], cromosoma_actual], 1))
    coste_mutacion += sum(np.sum(self.mP[[i, j], :] * self.mD[newChromosome[[i, j], None], newChromosome], 1))

    idx = list(range(self.dim))
    del(idx[i])
    del(idx[j-1])

    coste_mutacion -= sum(sum(self.mP[idx[:, [i,j]] * self.mD[cromosoma_actual[idx, None], cromosoma_actual[[i,j]]]))
    coste_mutacion += sum(sum(self.mP[idx[:, [i,j]] * self.mD[newChromosome[idx, None], newChromosome[[i,j]]]))

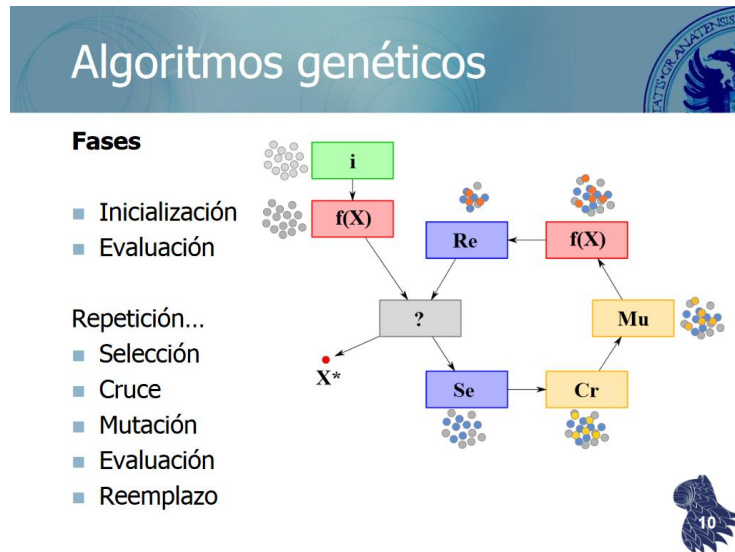
    return coste_mutacion
```

Ilustración 6 - Coste de mutación modificado

Simplemente se ha obviado el proceso de revertir los cambios ya que para nuestro caso en particular por el esquema de nuestro diseño únicamente nos interesa saber el coste de la mutación en específico.

1.4 Reemplazo

Continuando con el esquema de diseño de algoritmos genéticos visto en clase, el siguiente paso consiste en el reemplazo.



Una vez obtenidos los resultados productos del cruce y mutación el siguiente paso consiste en evaluar si efectivamente se han obtenido mejores resultados que los iniciales.

Este paso es muy sencillo y lo único que tendremos que realizar es sustituir el individuo hijo por el individuo padre en caso de que el coste sea menor, o bien no realizar ningún cambio en el caso de que los resultados hayan empeorado.

2 VARIANTE LAMARCKIANA

Para llevar a cabo esta variante hemos utilizado la información de esta web <https://planetachatbot.com/entendiendo-los-algoritmos-gen%C3%A9ticos-un-caso-de-uso-en-el-entorno-organizacional-a745c157fa8c> en la que se indica lo siguiente:

“Metodología anterior añadiendo las teorías de Lamarck:

1. *Población inicial.*
2. *Adaptación Lamarckiana.*
3. *Evaluación de la adaptabilidad.*
4. *Selección y cruce.*
5. *Mutación.*

Reorganización de la población con las nuevas generaciones.

La adaptación cromosómica lamarckiana puede ser llevada a cabo con algoritmos de optimización de búsqueda local, por ejemplo el algoritmo Hill climbing. Esta adaptación afecta a nivel genético y, por lo tanto, es transmitida a siguientes generaciones.”

En nuestro caso hemos decidido replicar el algoritmo genético simple anterior al completo, he introducir como modificación el uso de técnicas de optimización local, en este caso la heurística de “Greedy”.

En concreto, en la especificación del ejercicio se nos aconseja utilizar el siguiente pseudocódigo:

```

1 S = candidato inicial con coste c(S)
2
3 do {
4
5     mejor = S
6
7     for i=1..n
8         for j=i+1..n
9             T = S tras intercambiar i con j
10            if c(T) < c(S)
11                S = T
12
13 } while (S != mejor)
```

Ilustración 7 - Algoritmo de Greedy

Como puede comprobar, se trata de realizar continuos intercambios en bucle hasta encontrar un candidato que represente un coste inferior al actual. Nuestra versión, básicamente se basa en utilizar esta estrategia.

```
def twoOpt(self, initialIndividual):
    S = initialIndividual.copy()
    T = S.copy()
    T["coste"] = -1

    while (S["coste"] != T["coste"]):
        s_coste = S["coste"]
        t_coste = T["coste"]
        for i in range(self.dim):
            for j in range(i+1, self.dim):
                T = S.copy()
                T["cromosoma"][i], T["cromosoma"][j] = S["cromosoma"][j], S["cromosoma"][i]
                T["coste"] = self.coste(T["cromosoma"])

                if T["coste"] > S["coste"]:
                    S = T
        s_coste = S["coste"]
        t_coste = T["coste"]
    return S
```

Ilustración 8 - Greedy Lamarck

¿Qué debemos tener en cuenta en el algoritmo anterior? Pues que tal y como hemos indicado antes, esta adaptación afecta a nivel genético y por lo tanto se transmite al resto de generaciones.

```
pool = multiprocessing.Pool(processes=8)
parent[:n0pts] = pool.map(opt.twoOpt, parent[:n0pts])

pool.close()

parent.sort(order_="__coste", kind_='mergesort')

for i in range(10):
    # Mecanismo de selección basada en torneo
```

Ilustración 9 - Aplicación de Greedy

En concreto, esta adaptación es aplicada justo antes del proceso de selección, cruce y mutación (véase la imagen superior) por lo que nos aseguramos de que todos estos cambios se transmiten al resto de generaciones. Utilizamos multiprocessing para agilizar la ejecución.

Además, volvemos a utilizar Greedy una vez finalizado el proceso de cruce y mutación.

3 VARIANTE BALDWINIANA

A diferencia del caso anterior, tal y como se expresa en el esquema informativo de la especificación del problema, la variante baldwiniana utiliza para calcular el nuevo fitness o coste del individuo que alcanza el óptimo local, sin embargo, cuando se comienzan a crear los descendientes únicamente se utiliza el material genético del individuo original.

- Implemente una variante baldwiniana de su algoritmo genético estándar, que incorpore técnicas de optimización local (p.ej. heurísticas greedy) para dotar a los individuos de su población de capacidad de “aprendizaje”. Compare los resultados obtenidos con respecto a los que obtuvo el algoritmo genético estándar.

NOTA: En la variante baldwiniana del algoritmo genético, para evaluar el fitness de cada individuo, se utiliza dicho individuo como punto inicial de una búsqueda local (p.ej. ascensión de colinas por la máxima pendiente) hasta que se alcanza un óptimo local. El valor de ese óptimo local determina el fitness del individuo. Sin embargo, a la hora de formar descendientes, se utiliza el material genético del individuo original (sin incluir las mejoras “aprendidas” al aplicar la técnica de búsqueda local).

Ilustración 10 - Notas de especificación

¿Cómo vemos reflejado este proceso? Pues en nuestro caso se ha tenido en cuenta en el algoritmo de optimización “Greedy”.

```
def twoOptBalwin(self, initialIndividual):
    bestIndividual = initialIndividual.copy()
    bestIndividual["coste"] = -1
    S = initialIndividual.copy()

    a_cost = 0
    b_cost = 1

    while (a_cost != b_cost):
        bestIndividual = S.copy()

        for i in range(self.dim):
            for j in range(i+1, self.dim):
                cromosoma = S["cromosoma"][0]
                cromosoma[i], cromosoma[j] = cromosoma[j], cromosoma[i]
                newScore = self.coste(cromosoma)

                if S["coste"] > newScore:
                    S["coste"] = newScore
                else:
                    cromosoma[i], cromosoma[j] = cromosoma[j], cromosoma[i]
        a_cost = S["coste"]
        b_cost = bestIndividual["coste"]

    #print("Devuelvo:" _bestIndividual["coste"])
    return bestIndividual["coste"]
```

Ilustración 11 - Greedy para la variante baldwiniana



Como puede verse, en este caso únicamente modificamos el valor óptimo que alcanza el individuo, en las posteriores etapas como mutación y cruce que le siguen a continuación se utilizará el individuo original, a diferencia de la variante lamarckiana que utiliza este nuevo individuo creado para proceder a los cruces y mutaciones.

4 RESULTADOS

Para todos los algoritmos que se muestran a continuación se utilizará la siguiente configuración:

1. Tamaño de la población: 100.
2. Probabilidad de cruce: 0,5.
3. Probabilidad de mutación: 0,02.

4.1 Algoritmo genético simple

```
Se está ejecutando el algoritmo genético simple en un problema de
Coste del mejor padre en la generación (AGS) 0 51295600
Coste del mejor padre en la generación (AGS) 1 51124546
Coste del mejor padre en la generación (AGS) 2 50870582
Coste del mejor padre en la generación (AGS) 3 50526580
Coste del mejor padre en la generación (AGS) 4 50337860
Coste del mejor padre en la generación (AGS) 5 50337860
Coste del mejor padre en la generación (AGS) 6 50188894
Coste del mejor padre en la generación (AGS) 7 49774130
Coste del mejor padre en la generación (AGS) 8 49774130
Coste del mejor padre en la generación (AGS) 9 49681894
Coste del mejor padre en la generación (AGS) 10 49567204
Coste del mejor padre en la generación (AGS) 11 49485852
Coste del mejor padre en la generación (AGS) 12 49483990
```

Ilustración 12 - Ejecución inicial AGS

```
Coste del mejor padre en la generación (AGS) 97 47409672
Coste del mejor padre en la generación (AGS) 98 47409672
Coste del mejor padre en la generación (AGS) 99 47409672
(array([203, 62, 126, 167, 134, 31, 8, 165, 123, 103, 178, 192, 12,
247, 229, 50, 26, 218, 183, 209, 235, 33, 158, 154, 214, 138,
163, 169, 111, 230, 36, 249, 112, 172, 119, 98, 4, 40, 74,
45, 250, 89, 101, 220, 115, 80, 97, 147, 140, 200, 180, 69,
25, 254, 243, 239, 121, 226, 222, 116, 216, 68, 13, 194, 188,
78, 109, 100, 240, 212, 252, 63, 65, 43, 145, 2, 37, 6,
152, 19, 129, 55, 189, 56, 64, 191, 136, 143, 1, 87, 60,
91, 207, 82, 184, 221, 106, 5, 72, 244, 79, 234, 107, 57,
29, 48, 213, 76, 142, 77, 10, 118, 225, 58, 255, 14, 193,
173, 61, 28, 105, 30, 157, 198, 32, 155, 110, 24, 156, 196,
236, 130, 47, 66, 201, 20, 128, 22, 7, 94, 245, 54, 114,
241, 242, 182, 75, 15, 186, 153, 90, 162, 166, 179, 161, 251,
44, 51, 135, 71, 253, 104, 96, 16, 39, 132, 102, 144, 42,
171, 117, 120, 38, 95, 52, 148, 232, 67, 34, 150, 219, 81,
149, 231, 205, 210, 211, 159, 21, 59, 122, 248, 35, 187, 0,
199, 93, 185, 108, 41, 181, 160, 9, 168, 151, 227, 23, 92,
204, 164, 85, 217, 124, 237, 70, 83, 238, 202, 131, 190, 141,
174, 177, 195, 206, 125, 18, 73, 88, 246, 27, 137, 99, 175,
3, 113, 84, 197, 86, 176, 170, 208, 53, 127, 228, 139, 49,
46, 224, 17, 11, 146, 223, 233, 133, 215]), 47409672)
```

Ilustración 13 - Ejecución final AGS

Como puede observarse, en las etapas iniciales se comienza por un coste de 51295600 y finalmente, tras 100 iteraciones se reduce hasta 47409672. Lo cual representa una mejora del 8,20% aproximadamente, es decir, una media de un 0.082% de mejora en cada iteración.

4.2 Algoritmo genético, variante lamarckiana

```
Se está ejecutando el algoritmo genético Lamarck en un problema de dimensión: 256
Coste mejor padre en la generación (AGL) 0 44985284
Coste mejor padre en la generación (AGL) 1 44962086
Coste mejor padre en la generación (AGL) 2 44959918
Coste mejor padre en la generación (AGL) 3 44958146
Coste mejor padre en la generación (AGL) 4 44958146
Coste mejor padre en la generación (AGL) 5 44958146
Coste mejor padre en la generación (AGL) 6 44958146
Coste mejor padre en la generación (AGL) 7 44958146
Coste mejor padre en la generación (AGL) 8 44958146
Coste mejor padre en la generación (AGL) 9 44958146
(array([187, 83, 96, 155, 33, 163, 216, 224, 165, 0, 110, 95, 172,
138, 170, 40, 46, 133, 233, 248, 128, 231, 74, 195, 31, 148,
220, 189, 102, 54, 245, 10, 150, 88, 174, 35, 209, 91, 130,
44, 50, 197, 121, 223, 127, 124, 57, 48, 206, 93, 37, 22,
25, 145, 180, 71, 183, 254, 29, 243, 136, 98, 106, 78, 226,
52, 237, 214, 159, 115, 241, 100, 235, 192, 76, 4, 65, 59,
168, 18, 7, 178, 160, 252, 85, 27, 153, 228, 141, 119, 201,
218, 56, 242, 84, 152, 99, 175, 1, 51, 13, 164, 154, 212,
196, 230, 176, 142, 184, 49, 77, 140, 244, 5, 208, 94, 213,
171, 151, 24, 69, 66, 215, 68, 221, 113, 101, 203, 53, 249,
253, 234, 193, 81, 39, 229, 3, 135, 200, 38, 156, 92, 64,
188, 205, 20, 9, 11, 47, 179, 166, 80, 169, 41, 87, 34,
82, 199, 114, 255, 75, 58, 232, 204, 157, 32, 147, 190, 104,
202, 186, 162, 219, 28, 173, 63, 181, 251, 97, 8, 61, 250,
227, 129, 15, 42, 89, 126, 118, 207, 167, 139, 225, 217, 12,
21, 60, 86, 107, 109, 144, 247, 116, 19, 73, 120, 158, 2,
111, 72, 117, 123, 134, 108, 70, 137, 236, 149, 23, 55, 112,
125, 210, 6, 238, 79, 161, 14, 132, 143, 45, 182, 240, 146,
43, 26, 194, 177, 211, 16, 185, 17, 122, 30, 67, 90, 191,
131, 103, 246, 222, 198, 105, 239, 36, 62]), 44958146)
```

Ilustración 14 - Ejecución de variante lamarckiana

Como puede verse en esta variante se produce una gran mejora respecto al algoritmo anterior. Ya en el primer resultado (44985284) se ha producido una mejora del 14.02% frente a la variante simple.

Como contra, esta variante consume mucho más tiempo, de hecho, hemos tenido que reducir las iteraciones totales a 10. De media, cada iteración consume un total de 13,1 segundos, en total el proceso completo ha tardado 131 segundos.

Como dato final se ha obtenido muy buen resultado, un fitness de 44958146, siendo una mejora del 5,45% frente a la variante simple.

4.3 Algoritmo genético, variante baldwiniana

```

Escriba el tipo de Algoritmo genético que desea utilizar
1 - Algoritmo Genético Simple
2 - Algoritmo Genético Lamarck
3 - Algoritmo Genético Balwin
Elección: 3
Se está ejecutando el algoritmo genético Balwin en un problema de dimensión: 256
Coste mejor padre en la generación (AGB) 0 44956682
Coste mejor padre en la generación (AGB) 1 44900312
(array([ 36, 96, 19, 172, 177, 56, 196, 230, 253, 45, 208, 237, 157,
        138, 184, 24, 46, 101, 249, 248, 76, 109, 241, 126, 31, 148,
         5, 72, 84, 54, 229, 10, 150, 88, 48, 35, 209, 91, 106,
        29, 81, 41, 224, 222, 202, 124, 171, 175, 191, 93, 20, 22,
        25, 145, 180, 213, 183, 254, 243, 79, 99, 98, 67, 78, 226,
        68, 221, 214, 144, 116, 212, 100, 235, 176, 60, 4, 65, 43,
       168, 2, 7, 59, 153, 252, 86, 27, 189, 228, 141, 119, 201,
       219, 103, 112, 1, 123, 234, 158, 51, 77, 164, 114, 154, 32,
       215, 12, 160, 75, 33, 49, 165, 140, 244, 199, 142, 94, 69,
        74, 151, 242, 62, 66, 231, 52, 17, 128, 233, 203, 53, 113,
        63, 40, 193, 82, 133, 147, 3, 166, 200, 245, 156, 92, 64,
       188, 205, 155, 9, 11, 47, 179, 121, 80, 169, 118, 87, 34,
       173, 181, 104, 30, 13, 57, 232, 220, 174, 110, 130, 190, 37,
       159, 186, 162, 218, 28, 195, 50, 163, 251, 97, 8, 139, 250,
       204, 129, 15, 207, 89, 95, 39, 18, 167, 117, 225, 217, 0,
        21, 108, 85, 107, 127, 58, 247, 115, 14, 73, 120, 143, 206,
       111, 227, 61, 152, 134, 198, 70, 137, 236, 192, 23, 55, 187,
       125, 210, 6, 238, 135, 161, 71, 132, 38, 44, 182, 240, 146,
        42, 26, 194, 178, 211, 16, 185, 136, 122, 255, 83, 90, 149,
       131, 102, 246, 223, 197, 105, 239, 170, 216]), 44900312)

```

Ilustración 15 - Ejecución de variante baldwiniana

Definitivamente este es el método más costoso de todos (ha tardado 185 segundos). Debido a las capacidades de nuestro equipo solo se han realizado 2 iteraciones, ya que cada iteración de media consumía 92,5 segundos (unos 1,5 minutos). Sin embargo, los resultados finalmente han sido el mejor de todas las variantes utilizadas, con un fitness de 44900312 en la última iteración.

Este resultado representa un 0,12% de mejoría frente al mejor resultado obtenido por la segunda variante, la lamarckiana. Sin embargo, tenemos que destacar que, para la otra variante, se obtuvo su resultado en tan solo 39 segundos, diferencia bastante considerable ya que en este caso se han utilizado 185 segundos.