Metaheurísticas (2021-2022)
Grado en Ingeniería Informática
Universidad de Granada

Memoria Proyecto Final: Bird Mating Optimizing

Fernando Luque de la Torre
20953877A
fl1562001@correo.ugr.es

24 de junio de 2022

Índice

1. Optimización basada en el apareamiento de aves

1.1. Apareamiento de aves

Esta, como tantas otras metaheurísticas, se trata de una metaheurística basada en el comportamiento de una clase de animales, en este caso las aves. Las aves son animales vertebrados con reproducción sexual. Además, estos animales, realizan varios tipos de aparejamiento:

- La mayoría de las aves son **monógamas sociales**. Es decir, las parejas se mantienen durante la temporada de cría e incluso varios años después de esta hasta llegar incluso a la muerte de uno de los dos individuos (wikipedia.org/wiki/Aves).
- Otro pequeño porcentaje de las aves, realizan la **poliginia**, es decir, un macho se aparea con varias hembras generalmente de forma exclusiva, es decir, no permitiendo que otros se apareen con ellas (wikipedia.org/wiki/Poliginia)
- Además de la poliginia, en algunas especies también se da la **poliandria**, un modelo parecido a la poliginia pero que puede incluir varios individuos del mismo sexo
- Otra forma de reproducción que apenas se da pero que será también de nuestro interés es la **partenogénesis**, basada en el desarrollo de óvulos no fecundados que no requieren de los genes de la célula masculina y que por tanto no pueden dar lugar a individuos masculinos. (wikipedia.org/wiki/Partenogénesis)
- El último tipo de aparejamiento que es de nuestro interés es la **promiscuidad**, en la que no se forman parejas, sino que el apareamiento se realiza entre diferentes grupos sexuales.

Existen, además de estos, otros tipos de aparejamiento pero no son tanto de interés para nuestro estudio.

1.2. Aplicación de esta MH

En esta metaheurística basada en poblaciones, nuestra población se denominará **sociedad** y cada miembro de la sociedad (cada solución) se denominará un **pájaro**. Esta sociedad incluye pájaros macho y hembra. [1]
Las hembras serán aquellas que contengan los genes más prometedores y estarán divididas en dos grupos: partenogenéticas y poliándricas. Por otro lado, los machos estarán divididos en tres grupos: monógamos, polígínicos y promiscuos.
Por tanto, el algoritmo utilizará 5 especies de aves, con patrones de comportamiento diferentes.

1.2.1. Partenogénesis

En este grupo se encontrarán las hembras más prometedoras, estas llevarán a cabo un tipo de reproducción en la que el macho no interviene, ya que simplemente, la obtención

de una cría de este tipo se basará en una mutación de los genes de la hembra.

Esta mutación se basará en el intercambio de un elemento no seleccionado por uno sí seleccionado, por tanto, será un proceso de búsqueda similar al de la búsqueda local:

```
Solucion.mutacion():
    i = Random.int(0,n)
    j = Random.int(0,n)
    while selected[i]==selected[j]:
        i= Random.int(0,n)
        j= Random.int(0,n)
    swap(selected[i],selected[j])
    update_deltas() // No incluyo la factorización para
                    // hacerlo más corto
```

1.2.2. Poliandria

En este grupo se encontrarán las segundas hembras más prometedoras. El 30 % de estas llevarán a cabo un tipo de reproducción en la que se cruzarán con varios machos al azar, sin tener en cuenta si son mejores o peores, de la siguiente manera:

```
Solucion.cruce_multiple(padres):
    h = [False,...,False]
    seleccionados = Random.vector(0,padres.size()+1,m)
    for i in range(n):
        // SE coge de la madre
        if seleccionados[i] == 0:
            h[i] = selected[i]
        // Se coge de un padre
        else:
            h[i] = padres[seleccionados[i]-1].selected[i]
    return repare(h)
```

La función cruce_multiple no tiene por qué generar soluciones factibles, por lo que es necesario el operador de reparación (utilizo el mismo que para el cruce uniforme de la práctica 2):

```
repare(h):
    v = m - |S|
    avg = deltas.avg()
    if v=0:
        return Solution(h)
    else if v<0:
        while v<0:
            find j that maximize |delta(j)-avg|
```

```
            h[j] = 0
            v += 1
    else if v>0:
        while v>0:
            find j that minimize |delta(j)-avg|
            h[j] = 1
            v -= 1
    return Solution(h)
```

Este operador de reparación utiliza una heurística basada en extraer o insertar el elemento seleccionado o no seleccionado que más o menos se aleje de la media de los deltas.

1.2.3. Monogamia

Este es el primer grupo de los machos, los cuales realizarán un cruce simple con una hembra. En concreto, se cruzarán el 50 % de los machos al azar con diferentes hembras, teniendo las mejores más probabilidades de cruzarse. El cruce monógamo se basará en un cruce por posición, también utilizado en la práctica 2:

```
cruce_posicion(p1,p2):
    h1 = new bool [n]
    h2 = new bool [n]
    posiciones_nocompletas = []
    restos_p1 = []
    // Posiciones comunes
    for i in range(n):
        if p1[i]==p2[i]:
            h1[i] = h2[i] = p1[i]
        else:
            posiciones_nocompletas.add(i)
            restos_p1.add(p1[i])
    // Completamos no comunes con los restos de p1 barajados
    restos_p1 = shuffle(restos_p1)
    i=0
    for pos in posiciones_nocompletas:
        h1[pos] = restos_p1[i]
        i+=1
    i=0
    restos_p1 = shuffle(restos_p1)
    for pos in posiciones_nocompletas:
        h2[pos] = restos_p1[i]
        i+=1
    return h1,h2
```

Además, el esquema de cruces será el siguiente:

```
machos_cruzados = range(0,monogamy.size())
shuffle(machos_cruzados)
cruces_monogamy = 0.5*monogamy.size()
for i in range(cruces_monogamy):
    macho = monogamy[machos_cruzados[i]]
    hembras_posibles = range(0,hembras.size())
    shuffle(hembras_posibles)
    cruzado = false
    j=0
    while !cruzado and j<hembras.size():
        if Random.float(0,1) < probabilidades[hembras_posibles[j]]:
            cria = cruce_posicion(macho,hembras[hembras_posibles[j]]
            if cria.get_diff() < macho.get_diff():
                monogamy[machos_cruzados[i]] = cria
            cruzado = true
        ++j
```

Se analizará posteriormente las probabilidades de cruce de cada individuo y el cálculo de esta cantidad

1.2.4. Poliginia

Se trata de un proceso similar a la poliandria pero para los machos. De nuevo el 30 % de estos machos se cruzarán con varias hembras al azar:

```
Solucion.cruce_multiple(madres):
    h = [False,...,False]
    seleccionados = Random.vector(0,madres.size()+1,m)
    for i in range(n):
        // SE coge del padre
        if seleccionados[i] == 0:
            h[i] = selected[i]
        // Se coge de una madre
        else:
            h[i] = madres[seleccionados[i]-1].selected[i]
    return repare(h)
```

De igual modo se repara la solución con el mismo operador.

1.2.5. Promiscuidad

Se trata de las soluciones con peores resultados de todas. Estas soluciones realmente no llevarán a cabo ningún cruce con otros pájaros, sino que simplemente serán sustituidas enteramente por una nueva población del mismo tamaño:

```
promiscuous = GeneraPoblacion(polygyny.size(), n, m, d)
```

1.2.6. Esquema general

En cada generación, cada solución tiene una cría con el método que le corresponda y en caso de que su cría sea mejor que ella es sustituida por su cria, salvo en los promiscuos que ni se comparan y directamente se sustituyen.

Por último, al final de cada generación, los nuevos individuos se reordenan haciendo que incluso algunas crías de machos (si llegan a ser superiores que otras hembras) puedan convertirse en hembras y viceversa.

1.2.7. Generación aleatoria de población

Tanto para la inicialización de la población en la primera generación como para sustituir la población promiscua en cada iteración, se utiliza la generación de soluciones aleatorias factibles de la siguiente forma:

```
GeneraPoblacion(M, n, m, d):
    vector<Solution> p(M) // Generamos M soluciones
                          // con todos los elementos sin seleccionar
    posiciones = range(0,n)
    for i in range(M):
        selecteds(n,false)
        shuffle(posiciones)
        for j in range(m):
            selecteds[posiciones[j]] = true
        p[i] = Solution(selecteds, n,m,d)
    return p
```

2. Descripción del problema

El problema a resolver en esta práctica será el problema de la mínima dispersión diferencial (en inglés, *minimum differential dispersion problem*, MDD). Se trata de un problema de optimización combinatoria consistente en seleccionar un subconjunto M de m elementos ($|M| = m$) de forma que se minimice la dispersión entre los elementos de M.
La expresión de esta dispersión se puede formular como:

$$diff(M) = \max_{i \in M} \sum_{j \in M} d_{ij} - \min_{i \in M} \sum_{j \in M} d_{ij},\ para\ M \subset N, |M| = m$$

El objetivo de los algoritmos que desarrollaremos más tarde será el de encontrar S para minimizar este valor de $diff(S)$

En esta práctica, el modo de cálculo de dispersión será el siguiente:

1. Para cada punto v elegido, calcularemos $\Delta(v) = \sum_{u \in S} d_{uv}$

2. Por tanto la dispersión de una solución $diff(S) = \text{máx}_{u \in S}(\Delta u) - \text{mín}_{v \in S}(\Delta v)$

3. El objetivo por tanto es minimizar la dispersión de la solución obtenida, siendo S, la solución, un subconjunto, de tamaño m del conjunto total de puntos V (el cual tiene tamaño n, siendo $n \geq m$)

2.1. Representación de soluciones

La representación de soluciones utilizada en esta práctica es la siguiente:

- Sea $V = \{0, 1, ..., n-1\}$ el conjunto de puntos a elegir, de forma que $|V| = n$
- Sea m la cantidad de puntos que tenemos que elegir
- Sea d la matriz de distancias simétrica de modo que d_{ij} representa la distancia entre el punto i y el j

Una solución $S = \{s_1, s_2, ..., s_m\}$ será un conjunto de puntos de V, de forma que $\forall s_i \in \{0, 1, ..., n-1\}$.

2.2. Función objetivo

Por tanto, si la función $\Delta(S, v)$, se define como:

$$\Delta(S, v) = \sum_{u \in S} d_{uv}$$

nuestra función objetivo a minimizar será:

$$diff(S) = \max_{u \in S}(\Delta(S, u)) - \min_{v \in S}(\Delta(S, v))$$

El pseudocódigo resultante del cálculo de esta función sería algo así:

```
delta(S,u):
    delta = 0
    for v in S:
        if (u!=v)
            delta+=d[u][v]
    return delta

diff(S):
    min_delta = delta(S, S[0])
    max_delta = delta(S, S[0])
    for u in S:
        delta = delta(S, u)
        if (delta < min_delta):
```

```
            min_delta = delta
        if (delta > max_delta):
            max_delta = delta
    return max_delta - min_delta
```

Para la codificación binaria bastaría con modificar la función delta de la siguiente manera:

```
delta(S,u):
    delta = 0
    if S[u]:
        for n in S:
            if (n!=u) and (S[n]):
                delta+=d[u][n]
    return delta
```

Sin embargo, en nuestro caso, los algoritmos no van a utilizar exactamente esta función objetivo, sino que cada uno, incluirá modificaciones para llevar a cabo su factorización, es decir, se calculará la función objetivo a partir de cada modificación que se realice en el conjunto y no desde 0.
Es un buen momento para aclarar que, el hecho de haber querido mantener cierta factorización (básicamente mantener los valores de delta actualizados y almacenados, así como el valor máximo y mínimo de estos) en la función objetivo para todos los algoritmos, hace que el criterio de parada en función del número de llamadas a la función objetivo sea algo impreciso y difícil de ajustar, algo que aclararé más adelante.

3. Descripción de los algoritmos

3.1. Búsqueda Local

Nuestro algoritmo de Búsqueda Local comenzará con una solución inicial que recibe como parámetro. Después, para generar un vecino, intercambiaremos algún punto i que ya pertenezca al conjunto solución, por otro punto j que no pertenezca a la solución, es decir, que pertenezca a N todavía. Para cada vecino generado, veremos si el valor de su función objetivo es mejor que el que actual tenemos y si es mejor, como escogemos el esquema de primer mejor, actualizamos el valor de la solución actual y empezamos a generar vecinos de esta.

3.1.1. Factorización de la función objetivo

La función objetivo debe estar factorizada, esto lo haremos mientras hacemos el propio intercambio de un elemento que pertenecía a la solución por uno que no.
Para ello, habrá que ir actualizando los valores de los deltas, restando el valor de distancia del elemento que sale (i) y sumando la del elemento que entra (j).
En esta función he simplificado el cálculo del nuevo máximo y el nuevo mínimo pero el proceso sería ir comprobando cada delta que actualizamos para hacerlo más eficiente.

```
Solucion.intercambia(i, j):
    // Borramos el delta en la posición en que se encontrara el elemento
    deltas.erase(find(selected, i))
    selected.erase(i) // Borramos el elemento de seleccionados

    for (int k=0; k<deltas.size(); k++):
        deltas[k] = deltas[0] - d[i][selected[k]] + d[j][selected[k]]
    max_delta = max(deltas)
    min_delta = min(deltas)
    diff = max_delta - min_delta
```

3.1.2. Generación de vecino

Para generar un vecino, basta con crear una copia de una solución y realizar el intercambio de un elemento que pertenecía por uno que sí

```
Solucion.generavecino(i, j):
    Vecino = Solucion
    vecino.intercambia(i,j)
    return vecino
```

3.1.3. Descripción del algoritmo

Una vez definidas las particularidades de este algoritmo, podemos definir el algoritmo completo que utiliza las funciones previamente definidas.
De esta forma, nuestro algoritmo de búsqueda local partirá de una solución, elegida aleatoriamente como un conjunto de puntos del conjunto inicial. A partir de este conjunto de puntos inicial, generaremos el conjunto de intercambios posibles, es decir, el conjunto de parejas (i, j) tal que $i \in S$ y $j \in N$, que nos darán el vecindario de la solución actual. Posteriormente, barajaremos este conjunto de parejas y los iremos explorando, generando la solución correspondiente a realizar el intercambio que nos indica cada pareja, y en caso de que esta mejore la función objetivo, actualizando la función actual.
Además, añadimos un número máximo de soluciones exploradas, como nos indica el guión de 10^5.

```
busqueda_local(P0):
    actual = P0
    explorados = 0
    while (explorados < 10000):
        // Se describe la creación de la lista de candidatos
        intercambios = []
        vecindario_explorado = true
        for saca in actual:
            for mete in N:
                intercambios.add((saca,mete))
```

```
        intercambios.shuffle()
        // intercambios es la lista de candidatos

        // Se recorren los candidatos hasta encontrar un vecino mejor o hasta que
        // se recorra el vecindario sin mejorar
        for inter in intercambios:
            ++explorados
            vecino = actual.generavecino(inter.saca, inter.mete)
            if vecino.get_diff()<actual.get_diff():
                actual = vecino
                N.add(saca)
                N.delete(mete)
                vecindario_explorado = false
                break
        // Cuando explore todo un vecindario sin encontrar mejora sale del while
        if vecindario_explorado:
            break
    return actual
```

3.2. Mejora al BMO

Como mejora al algoritmo de Bird Mating Optimizer, lo combinaremos con la búsqueda local con el objetivo de realizar un algoritmo memético. Además, aprovechando el hecho de que la población está constantemente siendo ordenada tras cada generación, aprovecharemos para, en lugar de aplicar la búsqueda local sobre toda la población, aplicar la búsqueda local sobre el 10 % de población que mejores resultados obtenga.

La búsqueda local se aplicará cada 10 generaciones, como ya hemos comentado, sobre el 10 % de los mejores individuos de la población. La búsqueda local tendrá un límite de 1000 evaluaciones y el algoritmo completo, un límite de 10000.

4. Procedimiento para el desarrollo de la práctica

En mi caso, he desarrollado la práctica en el lenguaje C++, sin usar más recursos que algunas estructuras de datos de la STL como vector o pair.
Adicionalmente, he creado dos estructuras de datos (clases en C++) para representar el problema de una forma más clara
La primera de ellas es la clase Solution_enteros, en la que he incluído todos los métodos de gestión de los deltas, de máximos y mínimos, de modificación del conjunto (y en consecuencia actualización de los deltas y el valor de dispersión), etc. En esta clase he incluído también las dos factorizaciones de la función objetivo para los distitntos algoritmos.
Adicionalmente, también he implementado una clase muy simple, la clase Problem, la cual se encarga de gestionar la entrada del problema, es decir, los valores de m, n y

la matriz de distancias d, así como la generación del conjunto N de puntos libres y su posterior gestión (extracción e inserción de puntos).
Para la práctica 2 la clase Solution pasa a ser aquella que utiliza la codificación binaria. Esta también incluye la gestión de los deltas, máximos y mínimos necesarios para las factorizaciones.
Con el objetivo de automatizar la tarea de generación de los datos, he utilizado la librería `filesystem` de C++ que me permite iterar en los archivos de un directorio. De esta forma, para cada archivo del directorio datos he realizado 5 ejecuciones y almacenado los valores de coste (diff) obtenidos así como el tiempo empleado. Posteriormente, he extraído estos datos en un fichero csv con su correspondiente formato para posteriormente incluirlos en la tabla de plantilla proporcionada.
Para terminar con este apartado, también señalar que como generador de números aleatorios, he utilizado el fichero random.hpp proporcionado en la página de la asignatura. En cuanto a la semilla utilizada, la he inicializado una vez antes de las 5 ejecuciones de cada fichero al valor 10 tanto para el algoritmo greedy como para la búsqueda local, asegurándome así de que las diferentes ejecuciones de un mismo archivo tienen diferentes valores de semilla y que en todas las veces que repita el experimento tendré el mismo valor.

5. Experimentos y análisis de resultados

Se incluye la tabla de resultados para el Bird Mating Optimizer:

Caso	Coste medio obtenido	Desv	Tiempo
GKD-b_1_n25_m2	0.0000	0.00	4
GKD-b_2_n25_m2	0.0000	0.00	4
GKD-b_3_n25_m2	0.0000	0.00	3
GKD-b_4_n25_m2	0.0000	0.00	5
GKD-b_5_n25_m2	0.0000	0.00	5
GKD-b_6_n25_m7	28.1763	0.55	161
GKD-b_7_n25_m7	37.4335	0.62	69
GKD-b_8_n25_m7	37.7944	0.56	57
GKD-b_9_n25_m7	25.8258	0.34	134
GKD-b_10_n25_m7	26.2380	0.11	148
GKD-b_11_n50_m5	14.8870	0.87	119
GKD-b_12_n50_m5	15.8849	0.87	123
GKD-b_13_n50_m5	17.2919	0.86	185
GKD-b_14_n50_m5	17.1688	0.90	144
GKD-b_15_n50_m5	13.6104	0.79	65
GKD-b_16_n50_m15	107.2880	0.60	796
GKD-b_17_n50_m15	76.0718	0.37	919
GKD-b_18_n50_m15	92.0475	0.53	852
GKD-b_19_n50_m15	105.5870	0.56	656

GKD-b_20_n50_m15	105.2720	0.55	524
GKD-b_21_n100_m10	43.3783	0.68	1659
GKD-b_22_n100_m10	60.0420	0.77	839
GKD-b_23_n100_m10	47.6127	0.68	1462
GKD-b_24_n100_m10	75.8781	0.89	558
GKD-b_25_n100_m10	69.6649	0.75	952
GKD-b_26_n100_m30	234.6750	0.28	9754
GKD-b_27_n100_m30	226.0000	0.44	7233
GKD-b_28_n100_m30	272.4090	0.61	7597
GKD-b_29_n100_m30	340.8930	0.60	4933
GKD-b_30_n100_m30	263.0190	0.52	3746
GKD-b_31_n125_m12	54.3574	0.78	1801
GKD-b_32_n125_m12	83.4023	0.77	2773
GKD-b_33_n125_m12	57.6749	0.68	2244
GKD-b_34_n125_m12	95.1949	0.80	3052
GKD-b_35_n125_m12	57.1967	0.68	1682
GKD-b_36_n125_m37	297.9540	0.48	13293
GKD-b_37_n125_m37	443.1750	0.55	7267
GKD-b_38_n125_m37	323.4740	0.42	11829
GKD-b_39_n125_m37	306.1040	0.45	10033
GKD-b_40_n125_m37	366.8190	0.51	14730
GKD-b_41_n150_m15	117.8070	0.80	4760
GKD-b_42_n150_m15	110.0310	0.76	2861
GKD-b_43_n150_m15	111.2440	0.76	2715
GKD-b_44_n150_m15	138.5740	0.81	3604
GKD-b_45_n150_m15	106.4880	0.74	4286
GKD-b_46_n150_m45	348.6190	0.35	26104
GKD-b_47_n150_m45	481.2220	0.52	21156
GKD-b_48_n150_m45	427.3410	0.47	29553
GKD-b_49_n150_m45	449.5590	0.50	26626
GKD-b_50_n150_m45	362.5670	0.31	26912

Adicionalmente, se incluye también la tabla de resultados del correspondiente algoritmo memético con la mejora de la búsqueda local:

Caso	Coste medio obtenido	Desv	Tiempo
GKD-b_1_n25_m2	0.0000	0.00	249
GKD-b_2_n25_m2	0.0000	0.00	247
GKD-b_3_n25_m2	0.0000	0.00	250
GKD-b_4_n25_m2	0.0000	0.00	248

Caso	Coste medio obtenido	Desv	Tiempo
GKD-b_5_n25_m2	0.0000	0.00	248
GKD-b_6_n25_m7	20.3955	0.38	375
GKD-b_7_n25_m7	29.7190	0.53	362
GKD-b_8_n25_m7	20.9588	0.20	365
GKD-b_9_n25_m7	26.7920	0.36	364
GKD-b_10_n25_m7	26.8843	0.13	404
GKD-b_11_n50_m5	10.9009	0.82	356
GKD-b_12_n50_m5	11.6767	0.82	339
GKD-b_13_n50_m5	7.3062	0.68	355
GKD-b_14_n50_m5	6.1524	0.73	347
GKD-b_15_n50_m5	8.4780	0.66	333
GKD-b_16_n50_m15	82.6981	0.48	847
GKD-b_17_n50_m15	72.3527	0.34	761
GKD-b_18_n50_m15	76.8140	0.44	860
GKD-b_19_n50_m15	79.2386	0.41	849
GKD-b_20_n50_m15	70.2728	0.32	742
GKD-b_21_n100_m10	17.4818	0.21	942
GKD-b_22_n100_m10	30.8622	0.56	973
GKD-b_23_n100_m10	28.0668	0.45	906
GKD-b_24_n100_m10	35.9077	0.76	934
GKD-b_25_n100_m10	30.3663	0.43	940
GKD-b_26_n100_m30	203.4620	0.17	2261
GKD-b_27_n100_m30	178.4450	0.29	2496
GKD-b_28_n100_m30	164.8970	0.35	2969
GKD-b_29_n100_m30	200.6270	0.31	3057
GKD-b_30_n100_m30	154.0700	0.17	2931
GKD-b_31_n125_m12	29.2286	0.60	1434
GKD-b_32_n125_m12	38.1335	0.51	1479
GKD-b_33_n125_m12	38.3681	0.52	1437
GKD-b_34_n125_m12	37.8781	0.49	1544
GKD-b_35_n125_m12	32.2569	0.44	1168
GKD-b_36_n125_m37	191.1150	0.19	4399
GKD-b_37_n125_m37	239.8790	0.17	4791
GKD-b_38_n125_m37	284.0230	0.34	4785
GKD-b_39_n125_m37	271.3580	0.38	3597
GKD-b_40_n125_m37	296.2920	0.40	3834
GKD-b_41_n150_m15	47.3979	0.51	1981
GKD-b_42_n150_m15	42.6155	0.37	1779
GKD-b_43_n150_m15	57.1531	0.53	1823
GKD-b_44_n150_m15	46.4446	0.44	1997
GKD-b_45_n150_m15	55.5571	0.50	2253
GKD-b_46_n150_m45	304.1780	0.25	7232
GKD-b_47_n150_m45	346.4890	0.34	7234
GKD-b_48_n150_m45	353.3330	0.36	5536
GKD-b_49_n150_m45	350.1690	0.35	6419
GKD-b_50_n150_m45	381.4670	0.35	5747

Se incluye también la tabla comparativa con los resultados de otras prácticas:

Algoritmo	Desv	Tiempo
Greedy	0.67	24.496
BL	0.52	593.236
AGG-uniforme	0.46	2458.62
AGG-posicion	0.51	1459.12
AGE-uniforme	0.53	554.5
AGE-posicion	0.63	248.52
AM-10-1	0.44	2982.7
AM-10-0.1	0.49	2211.1
AM-10-0.1mej	0.51	2178.42
ES	0.52	303.94
BMB	0.37	3044.68
ILS	0.34	2903.18
ILS-ES	0.39	1828.36
BMO	0.55	5219.74
BMO-memetico	0.38	1955.58

Como podemos observar, no se trata del mejor algoritmo analizado durante la asignatura, sin embargo, tampoco obtiene tan malos resultados, si lo comparamos principalmente con otros algoritmos genéticos como el modelo estacionario con cruce por posición. Bien es verdad que sí se podría optimizar aún más este algoritmo modificando por ejemplo, las proporciones de pájaros que cruzan de cada tipo y darle más importancia incluso a los mejores pájaros (las hembras).

En cuanto al modelo memético correspondiente, podemos observar que la mejora tanto en tiempo como en resultados es bastante significativa respecto al algoritmo sin búsqueda local. Esto se debe a que, tal y como hemos visto a lo largo del desarrollo de todas las prácticas, la búsqueda local es de las mejores metaheurísticas posibles para este problema entonces, casi cualquier algoritmo que incluya algo similar a esta va a tener bastantes buenos resultados.

6. Tutorial para la ejecución

En este apartado se aportan las directrices para que el profesor pueda replicar fácilmente el experimento.
Antes de nada, cabe destacar que como IDE para el desarrollo he utilizado CLion, que incluye compatibilidad con CMake para la compilación y ejecución de los proyectos. La estructura de mi proyecto es la siguiente:

- En el directorio `comun` se pueden encontrar los ficheros cabecera donde están implementadas las estructuras utilizadas para la resolución del problema, así como el fichero `random.hpp`.

- En el directorio `src` se incluyen los dos ficheros `.cpp` que nos crean los dos ejecutables para los dos algoritmos. Estos ficheros en su función `main` ya se encargan de recorrer el directorio `datos` y ejecutar el algoritmo con cada uno de los problemas, así como de dar el coste y el tiempo de ejecución en el fichero `resultados/[nombre_algoritmo].csv`.
- El directorio `datos` contiene los diferentes problemas.
- El directorio `resultados` contiene los ficheros de salida de las ejecuciones de los algoritmos que ya he mencionado anteriormente y el fichero `Tabla Completa.xlsx` donde se agrupan estos datos obtenidos y se calculan los valores de desviación para cada problema, así como la media en coste y tiempo de ejecución para cada algoritmo. En la página 1 se mantienen los datos de la práctica 1 y en la 2 se incluyen los de esta práctica. Adicionalmente, en la 3 se mantiene un acumulado de estos resultados finales.
- El directorio `bin` que incluye ya las versiones compiladas de los dos experimentos.

Para la ejecución de los archivos, he añadido ejecutables al proyecto en el archivo `CMakeLists.txt`, uno para cada algoritmo. Por tanto, para replicar el experimento, únicamente habría que llamar al objetivo correspondiente con CMake y ejecutarlo. De igual modo, se incluye una versión del programa ya compilada en el directorio `bin` tal y como he dicho antes.
Destacar que es posible que los ficheros `resultados/[nombre_algoritmo].csv` contengan datos de tiempo diferentes a los del fichero `resultados/Tabla Completa.xlsx` ya que pertenecen a ejecuciones posteriores. Sin embargo, podemos ver cómo los valores de coste obtenidos no varían ya que siempre se inicializa con la misma semilla. En cambio los valores de tiempo pueden variar ligeramente, porque, aunque he intentado dejar el equipo ejecutando los experimentos, los diferentes componentes del sistema operativo pueden afectar a la ejecución de estos.

7. Bibliografía

Referencias

[1] Yannis Marinakis, Magdalene Marinaki y Nikolaos Matsatsinis. "A hybrid discrete Artificial Bee Colony - GRASP algorithm for clustering". En: *2009 International Conference on Computers Industrial Engineering*. 2009, págs. 548-553. DOI: 10.1109/ICCIE.2009.5223810.