Metaheurísticas - Grupo A1 (2021-2022)
Grado en Ingeniería Informática
Universidad de Granada

Memoria Práctica 2: Técnicas de Búsquedas Basadas en Poblaciones

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

10 de junio de 2022

Índice

1. 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 $diff(S) = \max_{u \in S}(\Delta u) - \min_{v \in S}(\Delta v)$
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 $n \geq m$)

1.1. 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 $\forall s_i \in \{0, 1, ..., n-1\}$.

1.2. 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.

1.3. Generación de soluciones aleatorias

La generación de una solución aleatoria de m elementos escogidos entre n es muy simple:

```
SolucionAleatoria(m,n):
    escogidos = shuffle(range(0,n))
```

```
    s = escogidos.sub_vector(0,m)
    return s
```

2. Descripción de los algoritmos

2.1. ES

El algoritmo basado en enfriamiento simulado será un procedimiento similar a la búsqueda local, pero una variable temperatura nos permitirá aceptar algunas soluciones vecinas algo peores con el objetivo de escapar de mínimos locales.

2.1.1. Gestión de la temperatura

El aceptar o no más o menos soluciones peores dependerá del valor de la temperatura que se irá decrementando a medida que avance el algoritmo. Para ello, es necesario inicializarla y controlarla.

Temperatura inicial y final Tanto la temperatura inicial como la final se establecerán en unos valores predefinidos. En concreto, la temperatura inicial se calculará de la siguiente forma, en función del coste de la solución inicial aleatoria escogida:

```
CS0 = SolucionAleatoria(n,m).get_diff()
mu = phi = 0.3
Temperatura_inicial = (mu*CS0)/(-log(phi))
```

Por otro lado, la temperatura final se fijará a un valor de 10^{-3}, pero realizaremos la comprobación de que la final sea menor que la inicial de la siguiente manera:

```
    Temperatura_final = 0.001
    if Temperatura_inicial < Temperatura_final:
        Temperatura_final = Temperatura_inicial/100
```

Esto con el objetivo de permitir algunas iteraciones en caso de que la temperatura inicial sea muy baja.
Esquema de enfriamiento Para el esquema de enfriamiento emplearemos el esquema de Cauchy modificado, que se basa en un valor calculado de la siguiente manera. Para el cálculo de este valor necesitaremos conocer M que será el número de enfriamientos a realizar y que en nuestro caso será $100000/max\_vecinos$, donde max_vecinos será un valor que comentaremos más adelante:

```
    Beta = (Temperatura_inicial-Temperatura_final)/
           (M*Temperatura_inicial*Temperatura_final)
```

Este valor se utilizará cada vez que se alcance una de las condiciones de enfriamiento y la temperatura se reducirá de la siguiente forma:

```
T = T/(1+Beta*T)
```

Condiciones de enfriamiento Son dos las condiciones que establecen cuando enfriar la temperatura:

- La primera, el número de vecinos generados en esta temperatura. Cuando para una misma temperatura se hayan generado un número máximo de vecinos, se reduce esta temperatura. En concreto, el valor máximo de vecinos será $max\_vecinos = 10n$, siendo n el tamaño del problema (en mi caso el número de elementos a seleccionar)

- La segunda, el número de éxitos en esta temperatura. Aclarar que en este apartado se entiende por éxito el paso de una solución a una vecina, no tiene por qué ser porque sea una mejor solución, también puede darse el caso de que lo permita la temperatura aún siendo peor, eso también se considera un éxito. Cuando para una misma temperatura se hayan dado un número máximo de éxitos, se reduce la temperatura. En concreto, el número máximo de éxitos será el 10 % del número máximo de vecinos: $max\_exitos = 0,1 max\_vecinos$

Cabe aclarar que para el control de estos máximos se llevarán dos variables vecinos_generados y exitos que se incrementarán cuando corresponda y se resetearán a 0 cuando se enfríe la temperatura
Condiciones de parada Son dos las condiciones de parada:

- La primera es que se alcance el número máximo de evaluaciones, que en este caso se fijará en 100000. Por evaluación se entiende el cálculo de la función objetivo, que en mi caso se realiza cuando se genera un vecino.

- La segunda es que, cuando se vaya a enfriar la temperatura por el criterio de vecinos máximos, el número de éxitos sea igual a 0, es decir, que no se haya aceptado ninguna solución (ni mejor ni peor) para la última temperatura.

2.1.2. Algoritmo

Este sería el pseudocódigo del Enfriamiento Simulado:

```
ES(n,m,d,N):
    s = SolucionAleatoria(n,m)
    // Sacamos los seleccionados del conjunto de
    // seleccionables
    N.extract(s.selected)

    // Calculos previos
    mu = phi = 0.3;
    T0 = (mu*s.get_diff())/(-log(phi))
    Tf = 0.001
    if (T0<Tf)
```

```
    Tf = T0/100

// Condiciones de enfriamiento
max_vecinos = 10*n
vecinos_generados = 0
max_exitos = 0.1*max_vecinos
exitos = 0

// Condición de parada 1
max_evaluaciones = 100000
evaluaciones = 0

Beta = (T0-Tf)/(M*T0*Tf)

// Almacenamos la mejor solución
mejor = s
T = T0
para = false

while ((T>Tf) and (evaluaciones < max_evaluaciones) and (!para)):
    // Similar a la BL
    // 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 recorre la lista de candidatos
    for cambio in intercambios if !para:
        vecina = s.neighbor(cambio.saca,cambio.mete)
        evaluaciones++
        vecinos_generados++
        // Se puede aceptar si es mejor o si es peor y lo permite la temperatura
        dist_diff = vecina.get_diff() - s.get_diff()
        if ((dist_diff<0) or (random.float(0,1) <= exp(-dist_diff/T))):
            s = vecina
            N.insert(cambio.saca)
            N.extract(cambio.mete)

            if (s.get_diff() < mejor.get_diff()):
                mejor = s
            exitos++
```

```
            // Si acepta pasa de vecindario
            break
            // Condiciones de enfriamiento
            if (vecinos_generados>=max_vecinos):
                T = T/(1+beta*T)
                vecinos_generados = 0
                // Condición de parada 2
                if (exitos==0):
                    para = true
                else
                    exitos = 0
            if (exitos>=max_exitos):
                T = T/(1+beta*T)
                exitos = 0
                vecinos_generados = 0

    return mejor
```

2.2. BMB

La búsqueda multiarranque básica no es más que una aplicación iterativa de la búsqueda local sobre diferentes soluciones iniciales aleatorias. El pseudocódigo quedaría tal que así:

```
BMB(d,n,m):
    mejor = Solucion()
    for i in range(10):
        inicial = SolucionAleatoria(n,m)
        s = bl(s)
        if (i==0) or (mejor.get_diff()>s.get_diff()):
            mejor = s
    return mejor
```

De esta forma, generaríamos 10 soluciones aleatorias que serían optimizadas cada una de ellas por la búsqueda local (limitada a 10000 evaluaciones) y nos quedaríamos con la mejor de estas soluciones encontradas por las búsquedas locales.

2.3. ILS

La búsqueda local iterativa se basa en la aplicación de la idea de la BMB pero con la mejora de iniciar cada una de las búsquedas locales con el mejor resultado encontrado hasta el momento. Sin embargo, con el objetivo de introducir más variaciones la búsqueda local no se realiza simplemente sobre la mejor solución sino sobre una mutación bastante severa (el 30 % de los elementos) de la misma. Este sería el pseudocódigo:

```
Solucion.mutacion(d,N,m,proporcion=0.3):
```

```
    // Suponemos que el conjunto N no tiene a los elementos que están en selected
    num_cambios = prop*selected.size()
    Solucion mutada = this
    for i in range(num_cambios):
        saca = selected[random.get(0,selected.size())]
        mete = N[random.get(0,N.size())]
        mutada = mutada.neighbor(saca,mete)
        N.erase(mete)
        N.insert(saca)
    return mutada
```

De esta forma, generamos una solución que comparte suficiente información del individuo original pero también introduce cambios bastante relevantes. El pseudocódigo del algoritmo completo sería el siguiente:

```
ILS(d,n,m):
    mejor = SolucionAleatoria(n,m)
    mejor = bl(mejor)
    for i in range(9):
        s = bl(mejor.mutacion(d,N,m)
        if mejor.get_diff()>s.get_diff():
            mejor = s
    return mejor
```

2.4. 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.

2.4.1. 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
```

2.4.2. 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
```

2.4.3. 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. 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.

4. Experimentos y análisis de resultados

Para todas las ejecuciones de todos los problemas con todos los algoritmos se ha utilizado la semilla 0 al inicio de las ejecuciones. Los parámetros de vecinos máximos, temperatura inicial, éxitos máximos, etc. son los indicados en el enunciado.

4.1. ES

Caso	Coste medio obtenido	Desv	Tiempo
GKD-b_1_n25_m2	0.0000	0.00	0
GKD-b_2_n25_m2	0.0000	0.00	0
GKD-b_3_n25_m2	0.0000	0.00	0
GKD-b_4_n25_m2	0.0000	0.00	0
GKD-b_5_n25_m2	0.0000	0.00	0
GKD-b_6_n25_m7	30.0073	0.58	11
GKD-b_7_n25_m7	20.1299	0.30	16
GKD-b_8_n25_m7	32.0345	0.48	10
GKD-b_9_n25_m7	44.2962	0.61	10
GKD-b_10_n25_m7	41.5800	0.44	18
GKD-b_11_n50_m5	8.4164	0.77	56
GKD-b_12_n50_m5	13.0283	0.84	41
GKD-b_13_n50_m5	10.7556	0.78	40
GKD-b_14_n50_m5	15.8076	0.89	41
GKD-b_15_n50_m5	7.7217	0.63	37
GKD-b_16_n50_m15	69.8455	0.39	59
GKD-b_17_n50_m15	75.6458	0.36	58
GKD-b_18_n50_m15	99.0735	0.56	72
GKD-b_19_n50_m15	106.2340	0.56	55
GKD-b_20_n50_m15	109.3800	0.56	38
GKD-b_21_n100_m10	28.6834	0.52	255
GKD-b_22_n100_m10	19.2227	0.29	186
GKD-b_23_n100_m10	34.0771	0.55	169
GKD-b_24_n100_m10	24.6465	0.65	126
GKD-b_25_n100_m10	48.9650	0.65	120

Caso	Coste medio obtenido	Desv	Tiempo
GKD-b_26_n100_m30	316.5830	0.47	455
GKD-b_27_n100_m30	243.6950	0.48	444
GKD-b_28_n100_m30	528.1430	0.80	457
GKD-b_29_n100_m30	430.4800	0.68	381
GKD-b_30_n100_m30	290.6660	0.56	327
GKD-b_31_n125_m12	47.1648	0.75	359
GKD-b_32_n125_m12	32.5372	0.42	188
GKD-b_33_n125_m12	54.9799	0.66	189
GKD-b_34_n125_m12	41.1028	0.53	194
GKD-b_35_n125_m12	38.8301	0.53	269
GKD-b_36_n125_m37	554.1590	0.72	586
GKD-b_37_n125_m37	444.9110	0.55	622
GKD-b_38_n125_m37	515.4590	0.64	916
GKD-b_39_n125_m37	428.2460	0.61	495
GKD-b_40_n125_m37	298.1240	0.40	812
GKD-b_41_n150_m15	48.8088	0.52	534
GKD-b_42_n150_m15	47.0662	0.43	397
GKD-b_43_n150_m15	51.7457	0.48	934
GKD-b_44_n150_m15	101.2530	0.74	274
GKD-b_45_n150_m15	47.5284	0.42	518
GKD-b_46_n150_m45	625.9860	0.64	917
GKD-b_47_n150_m45	453.1830	0.50	1180
GKD-b_48_n150_m45	787.1460	0.71	422
GKD-b_49_n150_m45	664.5500	0.66	1007
GKD-b_50_n150_m45	724.9640	0.66	902

4.2. BMB

Caso	Coste medio obtenido	Desv	Tiempo
GKD-b_1_n25_m2	0.0000	0.00	7
GKD-b_2_n25_m2	0.0000	0.00	11
GKD-b_3_n25_m2	0.0000	0.00	9
GKD-b_4_n25_m2	0.0000	0.00	10
GKD-b_5_n25_m2	0.0000	0.00	7
GKD-b_6_n25_m7	20.3956	0.38	46
GKD-b_7_n25_m7	24.0961	0.41	52
GKD-b_8_n25_m7	16.7612	0.00	43
GKD-b_9_n25_m7	17.0693	0.00	53
GKD-b_10_n25_m7	30.8806	0.25	67

Caso	Coste medio obtenido	Desv	Tiempo
GKD-b_11_n50_m5	3.7086	0.48	110
GKD-b_12_n50_m5	6.0228	0.65	151
GKD-b_13_n50_m5	11.2391	0.79	124
GKD-b_14_n50_m5	7.6390	0.78	129
GKD-b_15_n50_m5	6.7300	0.58	107
GKD-b_16_n50_m15	62.6460	0.32	645
GKD-b_17_n50_m15	74.7455	0.36	517
GKD-b_18_n50_m15	68.6451	0.37	417
GKD-b_19_n50_m15	78.8237	0.41	494
GKD-b_20_n50_m15	60.9226	0.22	465
GKD-b_21_n100_m10	20.5240	0.33	826
GKD-b_22_n100_m10	28.6768	0.52	932
GKD-b_23_n100_m10	32.4675	0.53	908
GKD-b_24_n100_m10	24.5836	0.65	885
GKD-b_25_n100_m10	28.6836	0.40	1178
GKD-b_26_n100_m30	264.9280	0.36	4835
GKD-b_27_n100_m30	231.1070	0.45	4332
GKD-b_28_n100_m30	200.8510	0.47	4803
GKD-b_29_n100_m30	199.9580	0.31	5089
GKD-b_30_n100_m30	164.8400	0.23	4528
GKD-b_31_n125_m12	21.7667	0.46	3021
GKD-b_32_n125_m12	36.9075	0.49	1794
GKD-b_33_n125_m12	33.4733	0.45	1867
GKD-b_34_n125_m12	29.9044	0.35	1720
GKD-b_35_n125_m12	26.4912	0.32	2006
GKD-b_36_n125_m37	291.4340	0.47	8295
GKD-b_37_n125_m37	276.8100	0.28	7763
GKD-b_38_n125_m37	329.9630	0.43	6222
GKD-b_39_n125_m37	267.9700	0.37	8503
GKD-b_40_n125_m37	295.0180	0.40	8535
GKD-b_41_n150_m15	44.9507	0.48	4046
GKD-b_42_n150_m15	46.2430	0.42	3001
GKD-b_43_n150_m15	47.4301	0.44	4784
GKD-b_44_n150_m15	44.7902	0.42	3096
GKD-b_45_n150_m15	48.8884	0.43	3351
GKD-b_46_n150_m45	412.3070	0.45	10938
GKD-b_47_n150_m45	363.4110	0.37	10682
GKD-b_48_n150_m45	393.1790	0.42	10912
GKD-b_49_n150_m45	423.5530	0.47	10239
GKD-b_50_n150_m45	519.1680	0.52	9679

4.3. ILS

Caso	Coste medio obtenido	Desv	Tiempo
GKD-b_1_n25_m2	0.0000	0.00	7
GKD-b_2_n25_m2	0.0000	0.00	9
GKD-b_3_n25_m2	0.0000	0.00	8
GKD-b_4_n25_m2	0.0000	0.00	9
GKD-b_5_n25_m2	0.0000	0.00	8
GKD-b_6_n25_m7	12.7180	0.00	46
GKD-b_7_n25_m7	23.6648	0.40	39
GKD-b_8_n25_m7	21.2855	0.21	53
GKD-b_9_n25_m7	25.5400	0.33	48
GKD-b_10_n25_m7	23.2651	0.00	49
GKD-b_11_n50_m5	8.8289	0.78	86
GKD-b_12_n50_m5	3.2941	0.36	89
GKD-b_13_n50_m5	12.9767	0.82	112
GKD-b_14_n50_m5	8.6732	0.81	112
GKD-b_15_n50_m5	2.9436	0.03	99
GKD-b_16_n50_m15	85.9270	0.50	553
GKD-b_17_n50_m15	48.1074	0.00	593
GKD-b_18_n50_m15	65.8839	0.34	437
GKD-b_19_n50_m15	63.9427	0.27	443
GKD-b_20_n50_m15	53.5642	0.11	450
GKD-b_21_n100_m10	23.7280	0.42	962
GKD-b_22_n100_m10	21.7585	0.37	822
GKD-b_23_n100_m10	35.2775	0.57	927
GKD-b_24_n100_m10	26.5731	0.67	1053
GKD-b_25_n100_m10	26.0747	0.34	838
GKD-b_26_n100_m30	278.6860	0.39	4116
GKD-b_27_n100_m30	234.2660	0.46	3977
GKD-b_28_n100_m30	214.9430	0.51	4768
GKD-b_29_n100_m30	167.7950	0.18	4174
GKD-b_30_n100_m30	184.3310	0.31	3240
GKD-b_31_n125_m12	15.9229	0.26	3127
GKD-b_32_n125_m12	37.3917	0.50	2101
GKD-b_33_n125_m12	30.4420	0.39	2282
GKD-b_34_n125_m12	36.4823	0.47	1866
GKD-b_35_n125_m12	24.9415	0.27	2072
GKD-b_36_n125_m37	261.6650	0.41	6497

Caso	Coste medio obtenido	Desv	Tiempo
GKD-b_37_n125_m37	344.0800	0.42	7175
GKD-b_38_n125_m37	292.5730	0.36	7641
GKD-b_39_n125_m37	313.1930	0.46	7620
GKD-b_40_n125_m37	294.0910	0.39	6938
GKD-b_41_n150_m15	38.5493	0.39	3584
GKD-b_42_n150_m15	49.9539	0.46	3888
GKD-b_43_n150_m15	43.4363	0.38	3691
GKD-b_44_n150_m15	37.5311	0.31	3109
GKD-b_45_n150_m15	44.5115	0.38	3238
GKD-b_46_n150_m45	436.9640	0.48	9651
GKD-b_47_n150_m45	360.2760	0.37	11162
GKD-b_48_n150_m45	430.6950	0.47	9814
GKD-b_49_n150_m45	374.9860	0.40	10870
GKD-b_50_n150_m45	377.2700	0.34	10706

4.4. ILS-ES

Caso	Coste medio obtenido	Desv	Tiempo
GKD-b_1_n25_m2	0.0000	0.00	1
GKD-b_2_n25_m2	0.0000	0.00	2
GKD-b_3_n25_m2	0.0000	0.00	1
GKD-b_4_n25_m2	0.0000	0.00	1
GKD-b_5_n25_m2	0.0000	0.00	1
GKD-b_6_n25_m7	13.4795	0.06	105
GKD-b_7_n25_m7	22.2649	0.37	94
GKD-b_8_n25_m7	16.7612	0.00	107
GKD-b_9_n25_m7	25.0145	0.32	114
GKD-b_10_n25_m7	26.8842	0.13	100
GKD-b_11_n50_m5	7.9082	0.76	287
GKD-b_12_n50_m5	5.6085	0.62	297
GKD-b_13_n50_m5	6.8864	0.66	313
GKD-b_14_n50_m5	1.6632	0.00	309
GKD-b_15_n50_m5	9.8429	0.71	305
GKD-b_16_n50_m15	91.1630	0.53	684
GKD-b_17_n50_m15	57.8190	0.17	754
GKD-b_18_n50_m15	69.1221	0.38	581
GKD-b_19_n50_m15	67.7699	0.32	698
GKD-b_20_n50_m15	63.5687	0.25	559
GKD-b_21_n100_m10	33.4336	0.59	1461
GKD-b_22_n100_m10	30.7441	0.56	1570

Caso	Coste medio obtenido	Desv	Tiempo
GKD-b_23_n100_m10	26.9775	0.43	1489
GKD-b_24_n100_m10	28.6639	0.70	1277
GKD-b_25_n100_m10	25.8499	0.33	1296
GKD-b_26_n100_m30	275.8190	0.39	2467
GKD-b_27_n100_m30	188.4860	0.33	2597
GKD-b_28_n100_m30	247.1200	0.57	2673
GKD-b_29_n100_m30	234.8450	0.41	2251
GKD-b_30_n100_m30	335.6750	0.62	2210
GKD-b_31_n125_m12	22.9978	0.49	2185
GKD-b_32_n125_m12	38.7311	0.51	2063
GKD-b_33_n125_m12	31.4794	0.41	1960
GKD-b_34_n125_m12	32.1151	0.39	1929
GKD-b_35_n125_m12	30.5819	0.41	2004
GKD-b_36_n125_m37	369.5080	0.58	3347
GKD-b_37_n125_m37	383.9600	0.48	3611
GKD-b_38_n125_m37	404.7450	0.54	3237
GKD-b_39_n125_m37	312.9790	0.46	3853
GKD-b_40_n125_m37	368.2200	0.52	3827
GKD-b_41_n150_m15	46.5720	0.50	2745
GKD-b_42_n150_m15	45.2834	0.41	2730
GKD-b_43_n150_m15	58.8889	0.55	2698
GKD-b_44_n150_m15	52.1240	0.50	2938
GKD-b_45_n150_m15	47.9926	0.42	2979
GKD-b_46_n150_m45	426.6280	0.47	5297
GKD-b_47_n150_m45	413.6820	0.45	5009
GKD-b_48_n150_m45	381.7540	0.41	4504
GKD-b_49_n150_m45	463.3590	0.51	4944
GKD-b_50_n150_m45	440.7940	0.44	4954

4.5. Comparación y valoración de resultados

Se incluye la tabla comparativa de todas las 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

Como podemos observar, en estos cuatro algoritmos desarrollados en esta última práctica son en los que mejores resultados hemos obtenido en general. Como conclusiones más concretas podemos extraer que:

- El algoritmo de Enfriamiento Simulado no es demasiado efectivo en este problema. Si analizamos un poco las ejecuciones, podemos comprobar fácilmente el comportamiento de la temperatura y vemos lo siguiente: la temperatura inicial sí que toma un valor considerable que nos permite aceptar soluciones relativamente peores con el objetivo de escapar de esos mínimos locales. Sin embargo, al menos con este esquema, la primera vez que realizamos un enfriamiento, la temperatura se reduce drásticamente y esto, unido al hecho de que, al no ser un dominio totalmente continuo el de la función objetivo, que hace que soluciones vecinas tengan valores de esta función muy diferentes, provoca que tras el primer enfriamiento, la probabilidad de que se escoja alguna solución peor va a ser muy baja ya que: la temperatura es muy baja y la diferencia de dispersión entre soluciones vecinas es muy alta. Es por esto, que los resultados de este algoritmo son prácticamente idénticos a los de la búsqueda local "simple"que desarrollamos en la práctica 2.

- En general, los algoritmos que utilizan búsqueda local son muy buenos en este problema, se comprobó en la P1, también en la P2 (donde los meméticos superaron a los genéticos) y por último, se está comprobando en esta P3, en la que la mayoría de los algoritmos son mejores que todos los anteriores. Esto, unido al punto anterior que nos dice que el enfriamiento simulado no mejora la búsqueda local apenas, se ve reflejado en el hecho de que los algoritmos de BMB e ILS sean mejores que el ILS hibridado con ES.

- Por último, parece lógico que el algoritmo ILS obtenga mejores resultados que el BMB ya que el punto de inicio del ILS ya es un resultado mejor que el del BMB y normalmente se trata de conservar estas características en la mutación. Sin embargo, el BMB simplemente comienza cada iteración con soluciones aleatorias que no tienen por qué ser buenas.

5. 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.