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

16 de mayo 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

1.1.1. Representación con enteros

Una de las dos representaciones utilizadas 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 $\forall s_i \in \{0, 1, ..., n-1\}$.

1.1.2. Representación binaria

La otra representación se basa en lo 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 $S = \{s_1, s_2, ..., s_n\}$ será un vector de tamaño n binario en el que el elemento s_i, con $i \in \{1, ..., n\}$ nos indica si el punto i pertenece o no a la solución.

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.

2. Operadores comunes

2.1. Operador de mutación

El operador de mutación se basa en un intercambio. Para la codificación binaria, basta con elegir un elemento que valga 0 (o 1) e intercambiar su valor con el de otro que valga 1 (o 0). La idea para la codificación de enteros es similar, intercambiando un elemento que no esté seleccionado por uno que sí (una idea similar a la aplicada en la búsqueda local para la generación de vecinos):

```
mutacion_binaria():
    i = random.int(0,n)
    j = random.int(0,n)
    while (s[i]==s[j]):
        i = random.int(0,n)
        j = random.int(0,n)
    swap(s[i],s[j])
    // Mantenimiento de los deltas:
    // Suponemos que delta es el valor del delta y delta.index el punto
    // al que hace referencia. d es la matriz de distancias
    // Caso 1: s[i] pasa de false a true, s[j] de true a false
    if s[i]:
        for delta in deltas:
            delta = delta - d[delta.index][j] + d[delta.index][i]
    // Caso 2: s[i] pasa de true a false, s[i] de false a true
    else:
        for delta in deltas:
            delta = delta - d[delta.index][i] + d[delta.index][j]
    // No incluyo el checkeo de máximos y mínimos para no ensuciar el
    // pseudocódigo, ya que es muy sencillo.

mutacion_enteros():
    saca = s[random.int(0,m)]
```

```
    mete = random.int(0,n)
    while (s.contains(mete)):
        mete = random.int(0,n)
    s.erase(saca)
    s.add(mete)
    for delta in deltas:
        delta = delta - d[delta.index][saca] + d[delta.index][mete]
    // Igual el checkeo de max_min
```

2.2. Generación de soluciones aleatorias

La generación de una solución aleatoria de m elementos escogidos entre n es muy simple: En el caso binario, generamos un vector de 0s y cambiamos por 1s m posiciones aleatorias (sin repetición) del mismo:

```
genera_binario():
    s = [false_1,...,false_n]
    cambiar = shuffle(range(0,n))
    for i in {1,...,m}:
        s[cambiar[i]] = true
    return s
```

En el caso de enteros, es quizás más sencillo, generamos un vector aleatorio sin repetición de números de 0 a n-1 de tamaño m:

```
genera_enteros():
    escogidos = shuffle(range(0,n))
    s = []
    for i in {1,...,m}:
        s.add(escogidos[i])
    return s
```

2.3. Operador de selección

El operador de selección utilizado será el torneo binario, es decir, se escogen dos elementos aleatorios de la población y se comparan entre ellos. El que mejor valor de en la función objetivo es con el que nos quedamos.
En el caso del modelo generacional, se realizarán 50 torneos aleatorios entre los 50 padres que nos darán lugar a 50 de los mejores individuos de la población. Puede que en esta población seleccionada encontremos elementos repetidos porque puede que un mismo padre haya ganado dos torneos, sin embargo, lo que está claro es que los peores elementos no ganarán sus respectivos torneos y por tanto no se reproducirán.
En el modelo estacionario, el número de torneos se reduce a dos, se escogen dos parejas que compiten entre sí y los dos ganadores serán los dos padres.
La descripción en pseudocódigo del torneo binario es muy simple y sería algo así:

```
torneo(ind1, ind2):
    if ind1.diff() < ind2.diff():
        return ind1
    else:
        return ind2
```

Bastaría con ejecutar este torneo con números aleatorios las veces que hiciera falta para cada uno de los modelos.

2.4. Operadores de cruce

2.4.1. Cruce uniforme

El operador de cruce uniforme genera un hijo (aplicándolo dos veces genera 2, que es lo que se hace en este algoritmo), a partir de dos padres. Para ello, el algoritmo coge las posiciones comunes a las dos soluciones (ya sean 0s o 1s) y el resto, las escoge aleatoriamente de un padre u otro. El pseudocódigo sería algo así:

```
cruce_uniforme(p1,p2):
    h1 = new bool [n]
    h2 = new bool [n]
    for i in range(n):
        if p1[i]==p2[i]:
            h1[i] = h2[i] = p1[i]
        else:
            if random.bool():
                h1[i] = p1[i]
            else:
                h1[i] = p2[i]
            if random.bool():
                h2[i] = p1[i]
            else:
                h2[i] = p2[i]
    repare(h1)
    repare(h2)
    return h1,h2
```

Como podemos ver, este operador no tiene por qué generar soluciones factibles (no tiene por qué haber m elementos seleccionados), por eso, se debe implementar también el operador de reparación.
No incluyo el código de este operador ya que podemos encontrarlo tal cual en las diapositivas del seminario 3, pero la idea básica es:

- Si sobran elementos: ir quitando el elemento i seleccionado, tal que delta[i] se aleje más de la media de los deltas

- Si faltan elementos: ir metiendo el elemento j no seleccionado, tal que delta[j] se acerque más a la media de los deltas

2.4.2. Cruce por posición

El primer paso del cruce por posición es similar al uniforme: quedarnos con las posiciones comunes. Sin embargo, para las posiciones no comunes nos quedamos con los elementos no seleccionados de uno de los padres y los añadimos a los hijos de forma aleatoria:

```
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
```

3. Descripción de los algoritmos

3.1. Algoritmos Genéticos

3.1.1. Modelo Generacional

Como ya hemos comentado en alguna ocasión a lo largo del documento, el modelo generacional se basa en:

1. Selección de M padres mediante M torneos binarios

2. Cruce (mediante los operadores descritos) de $P_c * M/2$ parejas aleatorias que nos dan lugar a $P_c * M/2$ hijos que formarán parte de la nueva población. El resto de hijos hasta completar los M, serán los padres que no se han cruzado entre ellos

3. Mutación mediante el operador descrito de $P_m * M$ individuos de la población

4. Reemplazamiento de la totalidad de la población, excepto del mejor de los padres, que se mantiene si es mejor que el peor de los hijos

Para la implementación de este modelo se han tenido en cuenta los aspectos respecto a la generación de números aleatorios de la que se habló en el seminario 3. Por tanto, el pseudocódigo de este modelo quedaría tal que así:

```
// P0 es la población inicial generada con
// el generador aleatorio ya comentado
def generación(P0, Pc, Pm):
    // M torneos binarios
    torneos = shuffle(range(0,2M))
    j=0
    mejor_padre = 0
    seleccion = new Solution [M]
    for i in range(M):
        seleccion[i] = torneo(P0[torneos[j]%M], P0[torneos[j+1]%M])
        j+=2
        // Aprovecho para quedarme con el mejor
        if seleccion[0].get_diff()>seleccion[i].get_diff():
            mejor_padre = i

    // Cruces
    cruces = shuffle(range(0,M))
    hijos = new Solution [M]
    for i in range(Pc*M/2):
        hijos[j],hijos[j+1] = cruce_...(
            seleccion[cruces[j]],seleccion[cruces[j+1]])
        j+=2
    // no cruces
    for i in range(Pc*M/2, M):
        hijos[i] = seleccion[i]

    // Mutaciones
    for i in range(Pm*M):
        hijos[i] = mutacion_...(hijos[i])

    // Reemplazamiento y elitismo para mantener el mejor padre
```

```
    peor_hijo = 0
    for i in range(hijos.size()):
        if hijos[peor_hijo].get_diff() < hijos[i].get_diff():
            peor_hijo = i
    if hijos[peor_hijo].get_diff > seleccion[mejor_padre].get_diff():
        hijos[peor_hijo] = seleccion[mejor_padre]

    P0 = hijos
    return P0
```

Siendo este el pseudocódigo para una generación, bastaría con repetirlo para hacer tantas generaciones como quisiéramos.

3.1.2. Modelo Estacionario

El modelo estacionario, a diferencia del generacional, no realiza tantos cruces, sino que únicamente realiza un cruce entre dos padres escogidos por dos torneos binarios, para dar lugar a 2 hijos que competirán por entrar en la población:

```
def estacionario(P0, Pm):
    // Buscamos los peores padres
    peores_padres = (0,0)
    for i in range(M):
        if P0[i].get_diff() > P0[peores_padres[0]].get_diff():
            peores_padres[0] = i
    for i in range(M):
        if P0[i].get_diff() > P0[peores_padres[1]].get_diff() && i!=peores_padres[0]:
            peores_padres[1] = i
    // Hacemos dos cruces
    p1,p2,p3,p4 = P0[random.int(0,M)],P0[random.int(0,M)],P0[random.int(0,M)],P0[random.in
    hijo1, hijo2 = cruce_...(torneo(p1,p2),torneo(p3,p4))
    // Mutación (igual que generacional)
    ...
    // Reemplazamiento
    // Comparamos cuál es mejor
    padre_peor = P0[peores_padres[0]]
    padre_malo = P0[peores_padres[1]]
    if hijo1.diff < hijo2.diff:
        if hijo1.diff < padre_peor.diff:
            P0[peores_padres[0]] = hijo1
            if hijo2.diff < padre_malo.diff:
                P0[peores_padres[1]] = hijo2
    else:
        if hijo2.diff < padre_peor.diff:
            P0[peores_padres[0]] = hijo2
```

```
            if hijo1.diff < padre_malo.diff:
                P0[peores_padres[1]] = hijo1
    return P0
```

Siendo este el pseudocódigo para una generación, bastaría con repetirlo para hacer tantas generaciones como quisiéramos.

3.2. Algoritmos Meméticos

3.2.1. AM-10-1

Este esquema realiza una combinación de un algoritmo basado en poblaciones con una búsqueda local. Concretamente, esta variante 10-1 realiza: cada 10 generaciones del algoritmo de poblaciones (en mi caso he escogido un modelo generacional con el cruce uniforme), realiza una búsqueda local sobre todos los elementos de la población (que son 10):

```
def AM-10-1(P0,Pc,Pm):
    generacion = 0
    while (!condicion_parada):
        if generacion%10=0 and generacion/10>0:
            for i in range(M):
                P0[i] = busqueda_local(P0[i])
            generacion /= 10
        else:
            ++generacion
            P0 = generacional(P0,Pc,Pm)
    return P0
```

3.2.2. AM-10-0.1

Este esquema es exactamente igual que el anterior, solo que únicamente se realiza la búsqueda local sobre el 10 % de la población cada 10 generaciones:

```
def AM-10-1(P0,Pc,Pm):
    generacion = 0
    while (!condicion_parada):
        if generacion%10=0 and generacion/10>0:
            bls = shuffle(random(M))
            for i in range(0.1*M):
                P0[bls[i]] = busqueda_local(P0[bls[i]])
            generacion /= 10
        else:
            ++generacion
            P0 = generacional(P0,Pc,Pm)
    return P0
```

3.2.3. AM-10-0.1mej

Este esquema es similar al anterior pero el 10 % de la población sobre la que se realiza la búsqueda será el 10 % que mejores resultados obtenga

```
def AM-10-1(P0,Pc,Pm):
    generacion = 0
    while (!condicion_parada):
        if generacion%10=0 and generacion/10>0:
            // Buscamos 10% mejor
            visitados = new bool [M] (false)
            for i in range(0.1*M):
                mejor = 0
                for j in range(M):
                    // Buscamos el mejor de los no visitados
                    if P0[mejor].diff()>P0[j].diff() && !visitados[j]:
                        mejor = j
                P0[j] = busqueda_local(P0[j])
                visitados[j] = true
            generacion /= 10
        else:
            ++generacion
            P0 = generacional(P0,Pc,Pm)
    return P0
```

3.3. Algoritmo de Búsqueda Local

Nuestro algoritmo de Búsqueda Local comenzará con un vector de puntos seleccionados de forma aleatoria del conjunto N. 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.

3.3.1. 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
```

3.3.2. 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
```

3.3.3. 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 < 100000):
        intercambios = []
        vecindario_explorado = true
        for saca in actual:
            for mete in N:
                intercambios.add((saca,mete))
        intercambios.shuffle()

        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
            break
        // Cuando explore todo un vecindario sin encontrar mejora sale del while
        if vecindario_explorado:
            break
    return actual
```

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

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 P_c, P_m, P_{ls} son los indicados en el enunciado.

5.1. Algoritmos Genéticos

5.1.1. Modelo Generacional

Cruce uniforme

Caso	Coste obtenido	Desv	Tiempo
GKD-b_1_n25_m2	0	0	396
GKD-b_2_n25_m2	0	0	368
GKD-b_3_n25_m2	0	0	389
GKD-b_4_n25_m2	0	0	372
GKD-b_5_n25_m2	0	0	371
GKD-b_6_n25_m7	20.3955	0.376433037	366
GKD-b_7_n25_m7	28.421	0.503931952	360
GKD-b_8_n25_m7	21.8265	0.232071564	357
GKD-b_9_n25_m7	30.2173	0.435117962	358
GKD-b_10_n25_m7	26.7197	0.129285508	473
GKD-b_11_n50_m5	14.3311	0.865599989	574
GKD-b_12_n50_m5	10.0244	0.788412274	520
GKD-b_13_n50_m5	17.9332	0.868271697	425
GKD-b_14_n50_m5	16.4247	0.898737876	440
GKD-b_15_n50_m5	11.3549	0.748731385	489
GKD-b_16_n50_m15	92.7317	0.539038107	1011
GKD-b_17_n50_m15	61.0236	0.211655655	1043
GKD-b_18_n50_m15	117.746	0.633141763	934
GKD-b_19_n50_m15	86.7231	0.464820215	917
GKD-b_20_n50_m15	128.395	0.628372522	911
GKD-b_21_n100_m10	52.2861	0.735455121	1199
GKD-b_22_n100_m10	53.9619	0.746778004	1308
GKD-b_23_n100_m10	45.4282	0.662205855	1209
GKD-b_24_n100_m10	55.2255	0.843538945	1192
GKD-b_25_n100_m10	58.4556	0.70575086	1291
GKD-b_26_n100_m30	222.058	0.24015532	3283

Caso	Coste obtenido	Desv	Tiempo
GKD-b_27_n100_m30	148.486	0.144045499	3142
GKD-b_28_n100_m30	170.533	0.376195868	3110
GKD-b_29_n100_m30	229.991	0.402354179	3431
GKD-b_30_n100_m30	181.719	0.29847875	3129
GKD-b_31_n125_m12	40.6302	0.710925863	1950
GKD-b_32_n125_m12	55.416	0.66094756	2077
GKD-b_33_n125_m12	75.3112	0.753933014	1904
GKD-b_34_n125_m12	71.5956	0.727799893	1798
GKD-b_35_n125_m12	50.5613	0.641773056	1972
GKD-b_36_n125_m37	228.489	0.319727558	5329
GKD-b_37_n125_m37	284.023	0.299723544	4963
GKD-b_38_n125_m37	274.7	0.315737059	4973
GKD-b_39_n125_m37	249.421	0.324073755	5179
GKD-b_40_n125_m37	333.231	0.465254613	5491
GKD-b_41_n150_m15	63.5051	0.632374723	2656
GKD-b_42_n150_m15	79.3508	0.662391557	2962
GKD-b_43_n150_m15	74.1896	0.639377083	2956
GKD-b_44_n150_m15	87.3342	0.703030542	2927
GKD-b_45_n150_m15	59.8978	0.53632671	3196
GKD-b_46_n150_m45	307.069	0.258312269	7509
GKD-b_47_n150_m45	294.091	0.222679715	7939
GKD-b_48_n150_m45	318.977	0.289148308	8024
GKD-b_49_n150_m45	257.82	0.121830696	8525
GKD-b_50_n150_m45	301.392	0.174309139	7233

Cruce por posición

Caso	Coste obtenido	Desv	Tiempo
GKD-b_1_n25_m2	0.00	0.00	194
GKD-b_2_n25_m2	0.00	0.00	192
GKD-b_3_n25_m2	0.00	0.00	186
GKD-b_4_n25_m2	0.00	0.00	203
GKD-b_5_n25_m2	0.00	0.00	186
GKD-b_6_n25_m7	25.89	0.51	311
GKD-b_7_n25_m7	28.67	0.51	319
GKD-b_8_n25_m7	42.42	0.60	334
GKD-b_9_n25_m7	25.65	0.33	315
GKD-b_10_n25_m7	26.88	0.13	318
GKD-b_11_n50_m5	15.82	0.88	351
GKD-b_12_n50_m5	11.92	0.82	337
GKD-b_13_n50_m5	7.44	0.68	346

Caso	Coste obtenido	Desv	Tiempo
GKD-b_14_n50_m5	12.05	0.86	340
GKD-b_15_n50_m5	2.94	0.03	341
GKD-b_16_n50_m15	96.07	0.56	747
GKD-b_17_n50_m15	78.12	0.38	769
GKD-b_18_n50_m15	78.54	0.45	761
GKD-b_19_n50_m15	83.35	0.44	737
GKD-b_20_n50_m15	89.70	0.47	739
GKD-b_21_n100_m10	35.08	0.61	793
GKD-b_22_n100_m10	34.14	0.60	799
GKD-b_23_n100_m10	61.57	0.75	788
GKD-b_24_n100_m10	35.03	0.75	785
GKD-b_25_n100_m10	36.21	0.52	785
GKD-b_26_n100_m30	385.73	0.56	2140
GKD-b_27_n100_m30	388.87	0.67	2129
GKD-b_28_n100_m30	290.48	0.63	2116
GKD-b_29_n100_m30	288.73	0.52	2155
GKD-b_30_n100_m30	215.89	0.41	2163
GKD-b_31_n125_m12	65.92	0.82	1109
GKD-b_32_n125_m12	32.99	0.43	1134
GKD-b_33_n125_m12	33.59	0.45	1111
GKD-b_34_n125_m12	43.40	0.55	1110
GKD-b_35_n125_m12	54.40	0.67	1124
GKD-b_36_n125_m37	439.24	0.65	3118
GKD-b_37_n125_m37	425.69	0.53	3110
GKD-b_38_n125_m37	402.46	0.53	3118
GKD-b_39_n125_m37	533.93	0.68	3118
GKD-b_40_n125_m37	375.37	0.53	3152
GKD-b_41_n150_m15	89.04	0.74	1527
GKD-b_42_n150_m15	69.37	0.61	1504
GKD-b_43_n150_m15	81.84	0.67	1516
GKD-b_44_n150_m15	69.03	0.62	1525
GKD-b_45_n150_m15	74.46	0.63	1510
GKD-b_46_n150_m45	613.57	0.63	4290
GKD-b_47_n150_m45	419.93	0.46	4313
GKD-b_48_n150_m45	574.13	0.61	4314
GKD-b_49_n150_m45	499.39	0.55	4283
GKD-b_50_n150_m45	641.73	0.61	4291

5.1.2. Modelo Estacionario

Cruce uniforme

Caso	Coste obtenido	Desv	Tiempo
GKD-b_1_n25_m2	0	0.00	68.00
GKD-b_2_n25_m2	0	0.00	69.00
GKD-b_3_n25_m2	0	0.00	72.00
GKD-b_4_n25_m2	0	0.00	72.00
GKD-b_5_n25_m2	0	0.00	65.00
GKD-b_6_n25_m7	38.5196	0.67	64.00
GKD-b_7_n25_m7	32.207	0.56	67.00
GKD-b_8_n25_m7	33.7809	0.50	69.00
GKD-b_9_n25_m7	37.3619	0.54	62.00
GKD-b_10_n25_m7	31.4356	0.26	73.00
GKD-b_11_n50_m5	34.2494	0.94	75.00
GKD-b_12_n50_m5	14.4448	0.85	83.00
GKD-b_13_n50_m5	11.8061	0.80	76.00
GKD-b_14_n50_m5	13.0263	0.87	114.00
GKD-b_15_n50_m5	8.44495	0.66	105.00
GKD-b_16_n50_m15	81.101	0.47	264.00
GKD-b_17_n50_m15	88.9064	0.46	208.00
GKD-b_18_n50_m15	116.48	0.63	211.00
GKD-b_19_n50_m15	114.59	0.59	178.00
GKD-b_20_n50_m15	102.85	0.54	199.00
GKD-b_21_n100_m10	57.8125	0.76	276.00
GKD-b_22_n100_m10	60.5789	0.77	254.00
GKD-b_23_n100_m10	47.3681	0.68	227.00
GKD-b_24_n100_m10	71.7784	0.88	398.00
GKD-b_25_n100_m10	66.095	0.74	384.00
GKD-b_26_n100_m30	271.689	0.38	740.00
GKD-b_27_n100_m30	234.673	0.46	662.00
GKD-b_28_n100_m30	234.222	0.55	812.00
GKD-b_29_n100_m30	218.469	0.37	661.00
GKD-b_30_n100_m30	195.691	0.35	866.00
GKD-b_31_n125_m12	68.0995	0.83	446.00
GKD-b_32_n125_m12	68.3298	0.73	377.00
GKD-b_33_n125_m12	42.9446	0.57	377.00
GKD-b_34_n125_m12	72.6896	0.73	333.00
GKD-b_35_n125_m12	52.1148	0.65	444.00

Caso	Coste obtenido	Desv	Tiempo
GKD-b_36_n125_m37	342.292	0.55	1099.00
GKD-b_37_n125_m37	279.755	0.29	1287.00
GKD-b_38_n125_m37	304.964	0.38	1102.00
GKD-b_39_n125_m37	306.888	0.45	1458.00
GKD-b_40_n125_m37	302.979	0.41	1213.00
GKD-b_41_n150_m15	93.5337	0.75	737.00
GKD-b_42_n150_m15	49.2019	0.46	566.00
GKD-b_43_n150_m15	94.4523	0.72	597.00
GKD-b_44_n150_m15	124.51	0.79	706.00
GKD-b_45_n150_m15	73.6504	0.62	845.00
GKD-b_46_n150_m45	373.352	0.39	1952.00
GKD-b_47_n150_m45	436.097	0.48	1681.00
GKD-b_48_n150_m45	382.2	0.41	1764.00
GKD-b_49_n150_m45	435.931	0.48	1819.00
GKD-b_50_n150_m45	452.48	0.45	1448.00

Cruce por posición

Caso	Coste obtenido	Desv	Tiempo
GKD-b_1_n25_m2	0	0.00	35.00
GKD-b_2_n25_m2	0	0.00	32.00
GKD-b_3_n25_m2	0	0.00	30.00
GKD-b_4_n25_m2	0	0.00	39.00
GKD-b_5_n25_m2	0	0.00	31.00
GKD-b_6_n25_m7	38.0517	0.67	50.00
GKD-b_7_n25_m7	40.4639	0.65	50.00
GKD-b_8_n25_m7	22.2758	0.25	53.00
GKD-b_9_n25_m7	25.0145	0.32	52.00
GKD-b_10_n25_m7	39.0557	0.40	52.00
GKD-b_11_n50_m5	15.7994	0.88	59.00
GKD-b_12_n50_m5	17.6905	0.88	57.00
GKD-b_13_n50_m5	21.2616	0.89	57.00
GKD-b_14_n50_m5	10.3943	0.84	61.00
GKD-b_15_n50_m5	9.70514	0.71	59.00
GKD-b_16_n50_m15	96.574	0.56	138.00
GKD-b_17_n50_m15	131.796	0.63	138.00
GKD-b_18_n50_m15	123.777	0.65	125.00
GKD-b_19_n50_m15	170.17	0.73	122.00

Caso	Coste obtenido	Desv	Tiempo
GKD-b_20_n50_m15	168.118	0.72	121.00
GKD-b_21_n100_m10	48.9394	0.72	137.00
GKD-b_22_n100_m10	70.4884	0.81	134.00
GKD-b_23_n100_m10	62.7512	0.76	131.00
GKD-b_24_n100_m10	48.1461	0.82	140.00
GKD-b_25_n100_m10	62.2343	0.72	131.00
GKD-b_26_n100_m30	424.663	0.60	354.00
GKD-b_27_n100_m30	424.468	0.70	365.00
GKD-b_28_n100_m30	471.603	0.77	356.00
GKD-b_29_n100_m30	501.241	0.73	363.00
GKD-b_30_n100_m30	301.188	0.58	366.00
GKD-b_31_n125_m12	91.8083	0.87	189.00
GKD-b_32_n125_m12	60.7382	0.69	194.00
GKD-b_33_n125_m12	84.2331	0.78	187.00
GKD-b_34_n125_m12	96.5715	0.80	197.00
GKD-b_35_n125_m12	47.301	0.62	186.00
GKD-b_36_n125_m37	590.469	0.74	533.00
GKD-b_37_n125_m37	775.537	0.74	521.00
GKD-b_38_n125_m37	660.304	0.72	531.00
GKD-b_39_n125_m37	630.388	0.73	555.00
GKD-b_40_n125_m37	610.443	0.71	542.00
GKD-b_41_n150_m15	86.8142	0.73	267.00
GKD-b_42_n150_m15	113.087	0.76	256.00
GKD-b_43_n150_m15	95.647	0.72	257.00
GKD-b_44_n150_m15	102.572	0.75	260.00
GKD-b_45_n150_m15	118.366	0.77	259.00
GKD-b_46_n150_m45	843.622	0.73	732.00
GKD-b_47_n150_m45	732.999	0.69	731.00
GKD-b_48_n150_m45	695.153	0.67	730.00
GKD-b_49_n150_m45	728.821	0.69	729.00
GKD-b_50_n150_m45	790.805	0.69	732.00

5.2. Algoritmos Meméticos

5.2.1. AM-10-1

Caso	Coste obtenido	Desv	Tiempo
GKD-b_1_n25_m2	0	0.00	539.00
GKD-b_2_n25_m2	0	0.00	544.00
GKD-b_3_n25_m2	0	0.00	533.00
GKD-b_4_n25_m2	0	0.00	517.00
GKD-b_5_n25_m2	0	0.00	522.00
GKD-b_6_n25_m7	32.7762	0.61	779.00
GKD-b_7_n25_m7	19.609	0.28	841.00
GKD-b_8_n25_m7	20.9588	0.20	847.00
GKD-b_9_n25_m7	28.6604	0.40	792.00
GKD-b_10_n25_m7	30.7019	0.24	838.00
GKD-b_11_n50_m5	8.76978	0.78	1273.00
GKD-b_12_n50_m5	8.76624	0.76	1256.00
GKD-b_13_n50_m5	9.3674	0.75	1330.00
GKD-b_14_n50_m5	7.14426	0.77	1297.00
GKD-b_15_n50_m5	7.79095	0.63	1255.00
GKD-b_16_n50_m15	66.1713	0.35	1753.00
GKD-b_17_n50_m15	87.4136	0.45	1684.00
GKD-b_18_n50_m15	66.6127	0.35	1735.00
GKD-b_19_n50_m15	76.6639	0.39	1565.00
GKD-b_20_n50_m15	69.9124	0.32	1652.00
GKD-b_21_n100_m10	35.5951	0.61	2571.00
GKD-b_22_n100_m10	34.4695	0.60	2570.00
GKD-b_23_n100_m10	30.3426	0.49	2919.00
GKD-b_24_n100_m10	36.809	0.77	2533.00
GKD-b_25_n100_m10	30.1185	0.43	2614.00
GKD-b_26_n100_m30	277.827	0.39	3836.00
GKD-b_27_n100_m30	201.864	0.37	3901.00
GKD-b_28_n100_m30	193.205	0.45	4223.00
GKD-b_29_n100_m30	191.302	0.28	3524.00
GKD-b_30_n100_m30	220.411	0.42	4077.00
GKD-b_31_n125_m12	32.2194	0.64	2967.00
GKD-b_32_n125_m12	43.1115	0.56	3666.00
GKD-b_33_n125_m12	28.4041	0.35	3396.00
GKD-b_34_n125_m12	36.6482	0.47	3445.00
GKD-b_35_n125_m12	46.2706	0.61	3350.00

Caso	Coste obtenido	Desv	Tiempo
GKD-b_36_n125_m37	295.042	0.47	4736.00
GKD-b_37_n125_m37	429.146	0.54	4319.00
GKD-b_38_n125_m37	392.706	0.52	4345.00
GKD-b_39_n125_m37	295.624	0.43	4196.00
GKD-b_40_n125_m37	373.08	0.52	4113.00
GKD-b_41_n150_m15	43.4915	0.46	4539.00
GKD-b_42_n150_m15	67.2222	0.60	4771.00
GKD-b_43_n150_m15	67.006	0.60	3663.00
GKD-b_44_n150_m15	52.2516	0.50	4132.00
GKD-b_45_n150_m15	42.8188	0.35	4064.00
GKD-b_46_n150_m45	456.844	0.50	8705.00
GKD-b_47_n150_m45	306.574	0.25	6206.00
GKD-b_48_n150_m45	391.734	0.42	7052.00
GKD-b_49_n150_m45	434.112	0.48	7404.00
GKD-b_50_n150_m45	444.392	0.44	5746.00

5.2.2. AM-10-0.1

Caso	Coste obtenido	Desv	Tiempo
GKD-b_1_n25_m2	0	0.00	278.00
GKD-b_2_n25_m2	0	0.00	258.00
GKD-b_3_n25_m2	0	0.00	263.00
GKD-b_4_n25_m2	0	0.00	258.00
GKD-b_5_n25_m2	0	0.00	257.00
GKD-b_6_n25_m7	38.6305	0.67	439.00
GKD-b_7_n25_m7	33.3063	0.58	434.00
GKD-b_8_n25_m7	24.4453	0.31	415.00
GKD-b_9_n25_m7	45.9348	0.63	442.00
GKD-b_10_n25_m7	28.251	0.18	502.00
GKD-b_11_n50_m5	10.1262	0.81	756.00
GKD-b_12_n50_m5	17.1971	0.88	789.00
GKD-b_13_n50_m5	16.0033	0.85	1064.00
GKD-b_14_n50_m5	19.0362	0.91	918.00
GKD-b_15_n50_m5	4.72534	0.40	755.00
GKD-b_16_n50_m15	119.111	0.64	1329.00
GKD-b_17_n50_m15	95.4387	0.50	1411.00
GKD-b_18_n50_m15	92.6335	0.53	1351.00
GKD-b_19_n50_m15	90.9917	0.49	1307.00
GKD-b_20_n50_m15	72.3976	0.34	1284.00
GKD-b_21_n100_m10	31.9905	0.57	2050.00
GKD-b_22_n100_m10	45.4918	0.70	2098.00

Caso	Coste obtenido	Desv	Tiempo
GKD-b_23_n100_m10	32.7032	0.53	2073.00
GKD-b_24_n100_m10	32.8518	0.74	2068.00
GKD-b_25_n100_m10	34.7214	0.50	2052.00
GKD-b_26_n100_m30	365.781	0.54	3025.00
GKD-b_27_n100_m30	248.497	0.49	3072.00
GKD-b_28_n100_m30	358.086	0.70	2889.00
GKD-b_29_n100_m30	199.642	0.31	2872.00
GKD-b_30_n100_m30	214.335	0.41	2764.00
GKD-b_31_n125_m12	22.3705	0.47	2529.00
GKD-b_32_n125_m12	39.8888	0.53	2579.00
GKD-b_33_n125_m12	46.4742	0.60	2566.00
GKD-b_34_n125_m12	43.8519	0.56	2621.00
GKD-b_35_n125_m12	42.6398	0.58	2640.00
GKD-b_36_n125_m37	324.426	0.52	3765.00
GKD-b_37_n125_m37	321.983	0.38	3799.00
GKD-b_38_n125_m37	316.264	0.41	3857.00
GKD-b_39_n125_m37	416.467	0.60	3691.00
GKD-b_40_n125_m37	322.534	0.45	3798.00
GKD-b_41_n150_m15	54.5989	0.57	3358.00
GKD-b_42_n150_m15	50.6438	0.47	3246.00
GKD-b_43_n150_m15	72.2577	0.63	3377.00
GKD-b_44_n150_m15	67.2141	0.61	3248.00
GKD-b_45_n150_m15	48.7672	0.43	3365.00
GKD-b_46_n150_m45	475.952	0.52	4589.00
GKD-b_47_n150_m45	454.049	0.50	4374.00
GKD-b_48_n150_m45	555.139	0.59	4741.00
GKD-b_49_n150_m45	467.271	0.52	4569.00
GKD-b_50_n150_m45	445.136	0.44	4370.00

5.2.3. AM-10-0.1mej

Caso	Coste obtenido	Desv	Tiempo
GKD-b_1_n25_m2	0	0.00	250.00
GKD-b_2_n25_m2	0	0.00	250.00
GKD-b_3_n25_m2	0	0.00	249.00
GKD-b_4_n25_m2	0	0.00	245.00
GKD-b_5_n25_m2	0	0.00	237.00
GKD-b_6_n25_m7	35.6182	0.64	452.00
GKD-b_7_n25_m7	30.3901	0.54	471.00
GKD-b_8_n25_m7	16.7612	0.00	469.00
GKD-b_9_n25_m7	28.5003	0.40	474.00

Caso	Coste obtenido	Desv	Tiempo
GKD-b_10_n25_m7	51.4246	0.55	564.00
GKD-b_11_n50_m5	18.4376	0.90	1085.00
GKD-b_12_n50_m5	21.2907	0.90	841.00
GKD-b_13_n50_m5	18.2319	0.87	740.00
GKD-b_14_n50_m5	10.8287	0.85	718.00
GKD-b_15_n50_m5	7.87207	0.64	733.00
GKD-b_16_n50_m15	102.974	0.58	1261.00
GKD-b_17_n50_m15	125.241	0.62	1320.00
GKD-b_18_n50_m15	130.219	0.67	1282.00
GKD-b_19_n50_m15	144.056	0.68	1229.00
GKD-b_20_n50_m15	95.6783	0.50	1274.00
GKD-b_21_n100_m10	56.6047	0.76	2064.00
GKD-b_22_n100_m10	52.6683	0.74	2036.00
GKD-b_23_n100_m10	33.5508	0.54	2076.00
GKD-b_24_n100_m10	31.2726	0.72	2004.00
GKD-b_25_n100_m10	33.9231	0.49	2051.00
GKD-b_26_n100_m30	259.604	0.35	2985.00
GKD-b_27_n100_m30	283.339	0.55	2933.00
GKD-b_28_n100_m30	278.741	0.62	2983.00
GKD-b_29_n100_m30	193.639	0.29	2944.00
GKD-b_30_n100_m30	243.192	0.48	2974.00
GKD-b_31_n125_m12	40.1804	0.71	2700.00
GKD-b_32_n125_m12	62.2194	0.70	2515.00
GKD-b_33_n125_m12	48.6608	0.62	2494.00
GKD-b_34_n125_m12	31.6692	0.38	2600.00
GKD-b_35_n125_m12	50.6629	0.64	2487.00
GKD-b_36_n125_m37	341.892	0.55	3532.00
GKD-b_37_n125_m37	346.528	0.43	3642.00
GKD-b_38_n125_m37	378.514	0.50	3514.00
GKD-b_39_n125_m37	399.569	0.58	3709.00
GKD-b_40_n125_m37	538.038	0.67	3608.00
GKD-b_41_n150_m15	47.0852	0.50	3273.00
GKD-b_42_n150_m15	47.6191	0.44	3341.00
GKD-b_43_n150_m15	42.9248	0.38	3211.00
GKD-b_44_n150_m15	57.1604	0.55	3249.00
GKD-b_45_n150_m15	47.0668	0.41	3204.00
GKD-b_46_n150_m45	436.965	0.48	4511.00
GKD-b_47_n150_m45	449.855	0.49	4381.00
GKD-b_48_n150_m45	445.189	0.49	4331.00
GKD-b_49_n150_m45	470.723	0.52	4757.00
GKD-b_50_n150_m45	643.234	0.61	4668.00

5.3. Comparación y valoración de resultados

Se muestra la tabla resumen incluyendo los valores de la práctica anterior:

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

Analizando los resultados obtenidos podemos sacar algunas conclusiones:

Dentro de los algoritmos genéticos, podemos observar que el modelo generacional obtiene bastantes mejores resultados que el modelo estacionario. Esto se puede deber a varias causas:

- La primera de ellas es que para un mismo número de generaciones, el modelo generacional explora una mayor cantidad de soluciones, ya que sustituye a la mayor parte de individuos de la población en cada generación, mientras que el modelo estacionario, solo cambia, a lo sumo, dos individuos.
- La búsqueda de los dos peores padres en el modelo estacionario consume una gran cantidad de ejecuciones de la función objetivo, lo que hace que, al estar esta factorizada, el tiempo se reduzca muchísimo, pero también limite mucho el espacio de búsqueda. Una opción sería aumentar el número máximo de llamadas a la función objetivo para contrarrestar estas búsquedas.

Por otro lado, podemos observar también que en ambos modelos, el cruce uniforme nos da unos mejores resultados. Esto se puede deber a que este cruce (pese a ser más costoso en tiempo) mantiene más información de los padres, lo que hace que padres que sean buenas soluciones den lugar más frecuentemente a hijos que también lo sean. En cambio, esta relación no se da con el cruce por posición, que mantiene menos información de los padres.

Analizando ahora los diferentes modelos de meméticos, podemos comprobar que la opción que mejor resultado nos da es la que aplica la búsqueda local en toda la población. Esto se puede deber al hecho de que realizar una búsqueda local en tan solo 1 individuo (como hacen los otros dos modelos), de una forma tan limitada (400 evaluaciones de la función objetivo o un vecindario sin mejora), haga que esta apenas tenga influencia (aunque se haga unicamente en la mejor opción) y por eso, los resultados de estos algoritmos son tan parecidos a los del algoritmo genético sin búsqueda local que hemos utilizado

(AGG-uniforme).

Ya por último, en comparación con los algoritmos de la práctica anterior, podemos observar que todos obtienen un mejor resultado que el greedy, sin embargo ninguno puede si quiera compararse en eficiencia a este. En cuanto a la búsqueda local simple, hay algoritmos que consiguen mejorarla aumentando mucho el tiempo de ejecución y aquellos que mantienen aproximadamente sus resultados (como el AGE-uniforme), también mantienen sus tiempos de ejecución.

En definitiva, los resultados obtenidos nos hacen ver en mi opinión, que se pueden realizar bastantes modificaciones a los diferentes modelos con el objetivo de tener unos mejores resultados y sobre todo, más equilibrados en resultado-tiempo en el modelo estacionario, donde conseguimos tiempos muy bajos pero esto penaliza mucho el resultado.

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.