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

Memoria Práctica 1: Técnicas de Búsqueda Local y Algoritmos Greedy

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

5 de abril 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

En primer lugar, hay que elegir una forma de modelar las soluciones. En mi caso, para este problema:

- 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 $S = \{s_0, s_1, ..., s_m\}$ 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
```

Sin embargo, en nuestro caso, los dos 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.
Estas modificaciones se indicarán en los apartados correspondientes a cada algoritmo.

2. Descripción de los algoritmos

2.1. Algoritmo Greedy

La primera opción para solucionar este problema con metaheurísticas, será la aplicación de un algoritmo greedy. Como vimos en la asignatura de algorítmica, los algoritmos Greedy son aquellos que, de entre todas las decisiones disponibles en un momento, escojan la más óptima en ese momento, aunque esto después pueda llevar a una solución peor.
Por tanto, para el algoritmo greedy a desarrollar para este problema, necesitaremos previamente definir algunos aspectos:

2.1.1. Factorización de la función objetivo

Como he indicado antes, cada uno de los algoritmos contendrá modificaciones a la función objetivo para hacerla más eficiente y no tener que calcular en cada paso la función objetivo desde 0.
Para el algoritmo greedy, la función objetivo será un parámetro de la función y se almacenarán los valores de delta para todos los elementos que ya componen la solución, así como los valores máximos y mínimos de estos deltas, actualizándolos según vayamos

añadiendo elementos a la solución.
Esto es, cuando se incluya un punto nuevo `u`, habrá que recalcular los nuevos deltas, añadiendo la distancia de cada elemento que ya formaba parte de la solución al nuevo elemento `u`. Ya como último paso, añadiremos al vector de deltas, el valor del delta para el punto `u` en el conjunto.
Antes, voy a definir una función que compruebe si para un nuevo valor de delta, este es mayor que el anterior mayor o menor que el anterior menor.

```
check_limits(new_delta):
    if (new_delta>max_delta):
        max_delta = new_delta
    if ((new_delta<min_delta) or (selected.size == 2))
        min_delta = new_delta
```

Como detalle a destacar, tengo que añadir la condición de que si solo tengo dos puntos seleccionados, establezca un valor de mínimo delta, sea cual sea, ya que este se inicializa ya 0 y si no, nunca se actualizaría este valor ya que los valores de delta siempre van a ser positivos y por tanto mayores que 0.
Ahora sí, puedo definir ya la factorización de la función objetivo

```
update_diff(S, u):
    // Actualizamos los deltas ya existentes
    deltas[0] = deltas[0]+d[selected[0]][u]
    max_delta = min_delta = deltas[0]
    for (int i=1; i<deltas.size();i++):
        deltas[i] = deltas[i]+d[selected[i]][u]
        check_limits(deltas[i])
    // Añado el delta para el punto u
    deltas.add(delta(S, u))
    diff = max_delta - min_delta
```

De esta forma, tendremos que llamar a esta función cada vez que añadamos un punto nuevo a la solución y esta nos actualizará el valor de la función objetivo en un tiempo lineal ($O(m)$) frente a un tiempo cuadrático ($O(m^2)$) que suponía la función objetivo sin factorizar.

2.1.2. Función de selección

Por tanto, como nuestro algoritmo greedy debe escoger en cada momento el punto que minimice la dispersión, debemos escoger m veces el punto u que aún no haya sido seleccionado (es decir que no esté ya en S) cuyo valor de $diff(S \cup u)$ sea el mínimo.
Para esto, implementaremos la función selección que recorra el conjunto de puntos no seleccionados y nos elija el que menor valor de diff nuevo nos genere
Para esto, crearemos el método añade que usará la función objetivo factorizada anteriormente definida

```
Solucion.añade(u):
    selected.add(u)
    update_diff(Solucion, u)
```

```
// N es el conjunto de puntos del que se parte y tiene tamaño n
// En esta función N contiene el conjunto de puntos inicial menos
// los puntos que están en S
select(S, N):
    Scopia = S
    Scopia.añade(N[0])
    mejor_diff = Scopia.diff
    mejor_opcion = N[0]
    for (u in N):
        Scopia = S
        Scopia.añade(N[0])
        if (Scopia.diff < mejor_diff):
            mejor_diff = Scopia.diff
            mejor_opcion = u
    return mejor_opcion
```

2.1.3. Función solución

La función solución en un algoritmo greedy (y en cualquier algoritmo) es aquella que comprueba si lo que llevamos conseguido (el conjunto S de seleccionados) es ya una solución al problema o no lo es.
En nuestro caso, la única condición que deben cumplir nuestras soluciones será que el conjunto de seleccionados tenga m elementos.

2.1.4. Descripción algoritmo

Nuestro algoritmo greedy será aquel que escoja m veces el punto que le devuelva la función de selección. Es decir escoger m veces el punto que menos amplíe la min_diff.
Hay que añadir una consideración y es que el primer punto se elige de forma aleatoria ya que no se puede calcular la dispersión de un solo punto.
El pseudo-código para este algoritmo quedaría por tanto:

```
// N es el conjunto de puntos del que se parte y tiene tamaño n
greedyDiff(n,m):
    primero = random(0,n)
    Solution.añade(primero)
    N = N\{primero}
    repeat m-1 times:
        u = select(Solution, N)
        Solution.aniade(u)
        N = N\{u}
    return Solution
```

2.2. 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.

2.2.1. Factorización de la función objetivo

De igual modo que para el algoritmo greedy, 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 similar al del greedy 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
```

2.2.2. 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
```

2.2.3. 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(N, m):
// Se entiende que un vector aleatorio de m elementos de N sin repetir
    actual = aleatorio(N,m)
    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
```

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, 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, tanto para el algoritmo greedy como para la búsqueda local.
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).
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

4.1. Resultados Greedy

Algoritmo Greedy			
Caso	Coste medio obtenido	Desv	Tiempo (ms)
GKD-b_1_n25_m2	0.0000	0.00	1.8
GKD-b_2_n25_m2	0.0000	0.00	2
GKD-b_3_n25_m2	0.0000	0.00	2.1
GKD-b_4_n25_m2	0.0000	0.00	2
GKD-b_5_n25_m2	0.0000	0.00	2.3
GKD-b_6_n25_m7	69.1205	0.82	1.5
GKD-b_7_n25_m7	45.5274	0.69	2
GKD-b_8_n25_m7	53.7245	0.69	1.8
GKD-b_9_n25_m7	64.5842	0.74	1.8
GKD-b_10_n25_m7	78.8779	0.71	2.2
GKD-b_11_n50_m5	25.3851	0.92	6
GKD-b_12_n50_m5	26.5515	0.92	5.9
GKD-b_13_n50_m5	25.1493	0.91	6.4
GKD-b_14_n50_m5	37.2861	0.96	6

GKD-b_15_n50_m5	28.0476	0.90	6
GKD-b_16_n50_m15	182.9050	0.77	7.1
GKD-b_17_n50_m15	187.2910	0.74	7
GKD-b_18_n50_m15	162.0430	0.73	6.8
GKD-b_19_n50_m15	160.5890	0.71	6.6
GKD-b_20_n50_m15	149.0720	0.68	6.3
GKD-b_21_n100_m10	79.4898	0.83	22
GKD-b_22_n100_m10	90.3288	0.85	21.8
GKD-b_23_n100_m10	68.9124	0.78	22.2
GKD-b_24_n100_m10	56.3157	0.85	21.7
GKD-b_25_n100_m10	64.5165	0.73	22.1
GKD-b_26_n100_m30	409.0970	0.59	25.2
GKD-b_27_n100_m30	383.1220	0.67	24.5
GKD-b_28_n100_m30	427.5690	0.75	24.8
GKD-b_29_n100_m30	326.2420	0.58	25.2
GKD-b_30_n100_m30	378.2170	0.66	25.2
GKD-b_31_n125_m12	91.0639	0.87	33.6
GKD-b_32_n125_m12	71.5299	0.74	33.7
GKD-b_33_n125_m12	86.1189	0.78	33.7
GKD-b_34_n125_m12	73.7969	0.74	33.5
GKD-b_35_n125_m12	93.6831	0.81	33.6
GKD-b_36_n125_m37	585.8640	0.73	38.5
GKD-b_37_n125_m37	495.1230	0.60	38.6
GKD-b_38_n125_m37	568.8400	0.67	51.4
GKD-b_39_n125_m37	506.9880	0.67	39.3
GKD-b_40_n125_m37	427.7110	0.58	39.4
GKD-b_41_n150_m15	115.4040	0.80	48.9
GKD-b_42_n150_m15	128.1870	0.79	48.9
GKD-b_43_n150_m15	109.4280	0.76	48.8
GKD-b_44_n150_m15	136.5550	0.81	49.6
GKD-b_45_n150_m15	126.6580	0.78	51.2
GKD-b_46_n150_m45	657.9280	0.65	57.6
GKD-b_47_n150_m45	584.9460	0.61	58.3
GKD-b_48_n150_m45	631.4800	0.64	55.6
GKD-b_49_n150_m45	578.6580	0.61	56.3
GKD-b_50_n150_m45	563.7070	0.56	56

Como era de esperar, la solución Greedy nos da un método muy eficiente, sin embargo, los resultados obtenidos no son precisamente buenos.
Como podemos observar, los valores de desviación se sitúan todos por encima del 0.5 (obviando los problemas con $m = 2$ para los cuales es obvio que la dispersión siempre va a ser 0), y la mayoría por encima del 0.7. Esto nos da a entender que la solución aportada por el algoritmo greedy se sitúa muy lejos de la mejor solución encontrada, lo que nos

hace ver que las soluciones alcanzadas por este algoritmo no son buenas.
Sin embargo, un grandísimo punto a favor es la magnífica eficiencia de este algoritmo. Tras realizar un pequeño análisis al algoritmo, se puede comprobar que la eficiencia teórica de este algoritmo es de $O(nm)$ gracias a la factorización de la función objetivo (sin esta factorización, la eficiencia sería de $O(nm^2)$). Es por esto que se consiguen valores tan buenos de tiempo.
En conclusión, con el algoritmo greedy obtenemos una solución al problema que no se acerca demasiado a la solución más óptima encontrada pero la obtenemos muy rápidamente.

4.2. Resultados BL

Algoritmo Búsqueda Local			
Caso	Coste medio obtenido	Desv	Tiempo (ms)
GKD-b_1_n25_m2	0.0000	0.00	2
GKD-b_2_n25_m2	0.0000	0.00	1.8
GKD-b_3_n25_m2	0.0000	0.00	2
GKD-b_4_n25_m2	0.0000	0.00	2
GKD-b_5_n25_m2	0.0000	0.00	2
GKD-b_6_n25_m7	31.5690	0.60	5.8
GKD-b_7_n25_m7	33.0409	0.57	4.4
GKD-b_8_n25_m7	34.5198	0.51	4.2
GKD-b_9_n25_m7	36.9938	0.54	6.2
GKD-b_10_n25_m7	35.6617	0.35	4.4
GKD-b_11_n50_m5	15.2367	0.87	17.4
GKD-b_12_n50_m5	16.9364	0.87	16.4
GKD-b_13_n50_m5	18.2877	0.87	13.4
GKD-b_14_n50_m5	15.6102	0.89	17
GKD-b_15_n50_m5	14.2439	0.80	23.6
GKD-b_16_n50_m15	122.2670	0.65	69
GKD-b_17_n50_m15	110.1590	0.56	62.8
GKD-b_18_n50_m15	89.5669	0.52	48.2
GKD-b_19_n50_m15	106.3840	0.56	49
GKD-b_20_n50_m15	87.2135	0.45	44.2
GKD-b_21_n100_m10	45.9212	0.70	100.4
GKD-b_22_n100_m10	34.4301	0.60	118.6
GKD-b_23_n100_m10	42.0480	0.64	131.8
GKD-b_24_n100_m10	31.8571	0.73	77.2
GKD-b_25_n100_m10	37.8112	0.55	117
GKD-b_26_n100_m30	382.1390	0.56	323.6
GKD-b_27_n100_m30	282.2650	0.55	423.2
GKD-b_28_n100_m30	337.9400	0.69	358.6

GKD-b_29_n100_m30	346.5230	0.60	354.2
GKD-b_30_n100_m30	310.0490	0.59	335.4
GKD-b_31_n125_m12	33.6075	0.65	443.8
GKD-b_32_n125_m12	47.8165	0.61	185
GKD-b_33_n125_m12	47.0347	0.61	297.8
GKD-b_34_n125_m12	46.1035	0.58	190.2
GKD-b_35_n125_m12	37.8909	0.52	355
GKD-b_36_n125_m37	374.5570	0.59	783
GKD-b_37_n125_m37	429.5050	0.54	787
GKD-b_38_n125_m37	487.8030	0.61	717.4
GKD-b_39_n125_m37	363.0150	0.54	713
GKD-b_40_n125_m37	431.6080	0.59	556.6
GKD-b_41_n150_m15	64.0391	0.64	422.8
GKD-b_42_n150_m15	62.8639	0.57	387.2
GKD-b_43_n150_m15	81.3328	0.67	451.2
GKD-b_44_n150_m15	62.9713	0.59	431.6
GKD-b_45_n150_m15	60.1306	0.54	432
GKD-b_46_n150_m45	469.5230	0.51	1707.4
GKD-b_47_n150_m45	489.8720	0.53	1183.8
GKD-b_48_n150_m45	423.9480	0.47	1588.4
GKD-b_49_n150_m45	498.7450	0.55	1481
GKD-b_50_n150_m45	642.4160	0.61	1197.4

Como podemos observar, los valores de desviación se sitúan en torno al 0.5 al 0.7 normalemente (aunque hay casos más extremos, por lo que podemos ver que nos estamos acercando bastante al mejor valor de dispersión encontrado. Dado que este es un algoritmo de búsqueda local bastante simple, podemos concluir que los resultados obtenidos están bastante bien, ya que el algoritmo que nos da la mejor solución obtenida contiene infinitud de mejoras para alcanzar una solución mejor.
En cuanto al tiempo de ejecución, si bien es cierto que la función objetivo factorizada para este caso tiene solo eficiencia $O(m)$ y no $O(m^2)$ como si no estuviera factorizada, la exploración de un vecindario completo supone $(n-m)m$ iteraciones, por lo que la eficiencia teórica de este algoritmo quedaría $O(nm^2)$. Esto nos supone por tanto que, aunque sigamos resolviendo el problema en un tiempo polinomial, el tiempo de ejecución crezca bastante según los valores de entrada llegando a tardar en torno a 1 segundo o 1.5 segundos en los casos más grandes.

4.3. Comparación

Por tanto, una vez analizados los datos obtenidos para ambos algoritmos, incluyo la tabla resumen y comparativa y procedo a analizar las ventajas y desventajas de cada uno.

Algoritmo	Desv	Tiempo
Greedy	0.67	24.496
BL	0.55	340.948

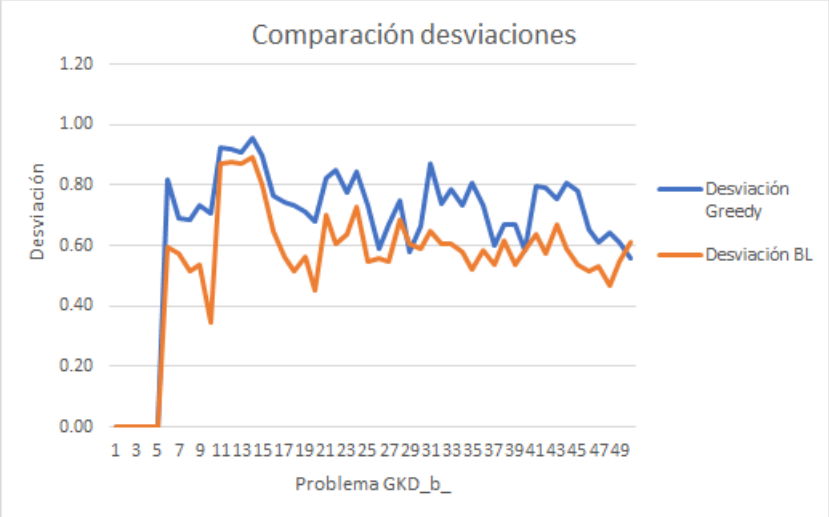

Figura 4.1: journalctl

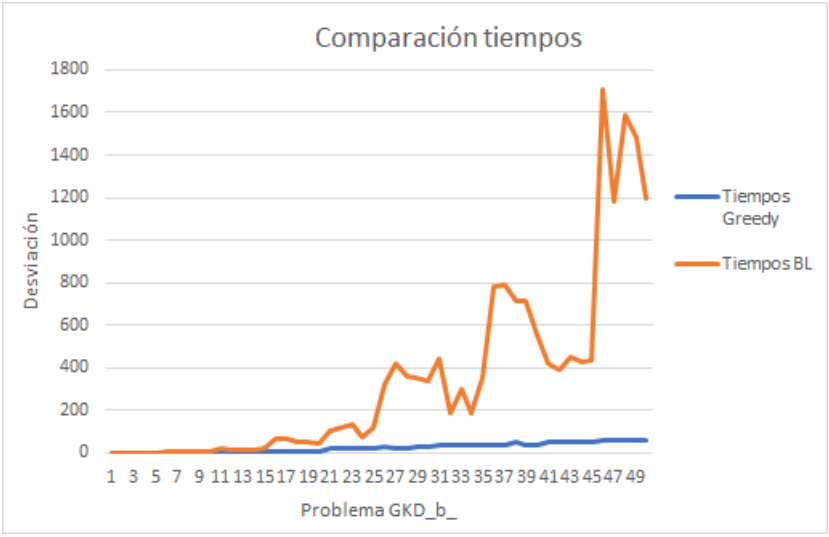

Figura 4.2: journalctl

Como podemos ver y como ya hemos analizado antes, el algoritmo greedy, por lo general, consigue soluciones bastante alejadas del óptimo, sin embargo, las consigue de una manera muy rápida. Por otro lado, la búsqueda local, sacrifica un poco de eficiencia por

encontrar soluciones mejores.
Por tanto, ¿qué algoritmo sería mejor que el otro? Pues como siempre, no hay una solución única, el algoritmo que utilicemos se debe adecuar a las circunstancias de nuestro problema. En el caso de que necesitáramos utilizar un algoritmo que nos diera una respuesta especialmente rápida y no nos importara sacrificar un poco la calidad de la solución, deberíamos emplear greedy. Si nos podemos permitir un poco más de tiempo, la búsqueda local nos daría una solución mucho mejor.
También cabe destacar que los tiempos obtenidos en la búsqueda local no son para nada descabellados, estando siempre por debajo de los dos segundos para los casos del experimento, por lo que realmente no sería una locura utilizar este algoritmo siempre para tamaños similares a estos.
Lo que sí está claro es que se trata de dos versiones muy simplificadas de estos algoritmos, y que aceptan infinidad de mejoras. Por poner un ejemplo, el algoritmo greedy podría, en lugar de escoger de forma aleatoria el primer punto, probar a escoger todos los puntos del conjunto N, y quedarse con la mejor solución que encontrara. Esto empeoraría su eficiencia obviamente, pero sería objeto de estudio analizar si esta pérdida de eficiencia compensa con la ganancia en calidad de la solución.

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 5 veces con cada uno de los problemas, así como de dar la media del coste y el tiempo de ejecución en el fichero `resultados/[greedy|bl].csv`.
- El directorio `datos` contiene los diferentes problemas.
- El directorio `resultados` contiene los ficheros de salida de las ejecuciones de los dos 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.
- El directorio `bin` que incluye ya las versiones compiladas de los dos experimentos.

Para la ejecución de los dos archivos, he añadido dos 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/[greedy|bl].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 para cada una de las 5 ejecuciones. 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.