Práctica 1.a
Técnicas de Búsqueda Local y Algoritmos Greedy para el Problema de la Mínima Dispersión Diferencial

Juan Antonio Martínez Sánchez

Curso 2021/22

?abstractname?

En este documento comentaremos técnicas de Búsqueda Local y algoritmos Greedy para resolver problemas de Mínima Dispersión Diferencial estudiados en la asignatura de Metaheurísticas.

?contentsname?

1 Introducción

El objetivo de esta práctica es estudiar el funcionamiento de las técnicas de Búsqueda Local y de los Algoritmos Greedy en la resolución del problema de la mínima dispersión diferencial (MDD).

El problema de la mínima dispersión diferencial (en inglés, minimum differential dispersion problem, MDD) es un problema de optimización combinatoria consistente en seleccionar un subconjunto M de m elementos (|M|=m) de un conjunto inicial N de n elementos (con n>m) de forma que se minimice la dispersión entre los elementos escogidos. El MDD se puede formular como:

$$Minimize\ Max_{i\in M}\{\sum_{j\in M} d_{ij}\} - Min_{i\in M}\{\sum_{j\in M} d_{ij}\}$$
$$Subjetc\ to\ M \subset N, |M| = m$$

donde:

- M es una solución al problema que consiste en un vector binario que indica los m elementos seleccionados.
- d_{ij} es la distancia existente entre los elementos i y j.

Se utilizarán 50 casos seleccionados de varios de los conjuntos de instancias todas disponibles en la MDPLIB, pertenecientes al grupo GDK-b con distancias aleatorias reales con, n entre $\{25, 50, 75, 100, 125, 150\}$, y m entre 2 y 45 (GDK-bGKD-b_1_n25_m2.txt a GKD-b_50_n150_m45.txt)

Para el problema MDD, se calcula la dispersión como:

1. Para cada punto elegido v se calcula (v) como la suma de las distancias de este punto al resto.
   $\Delta(v) = \sum_{u\in S} d_{uv}$
2. La dispersión de una solución, denotada como $diff(S)$ se define como la diferencia entre los valores extremos:
   $diff(S) = max_{u\in S}\Delta(u) - min_{v\in S}\Delta(v)$
3. El objetivo es minimizar dicha medida de dispersión:
   $S^\star = argmin_{S\subset V_m} diff(S)$
   donde S es el conjunto solución al problema.

2 Aplicación de los Algoritmos

2.1 Greedy

El algoritmo greedy del MDD se basa en la heurística de ir seleccionando los elementos que reduzcan o que aumenten lo mínimo la dispersión entre los distintos elementos elegidos. Los dos primeros elementos se escogen aleatoriamente ya que la dispersión entre dos puntos es 0 y así daremos una mayor aleatoriedad. En los m-2 pasos siguientes se va escogiendo el elemento que implique una menor dispersión de los ya seleccionados hasta el momento y el nuevo elemento considerado.

Pseudocódigo Greedy:
ALGORITMO GREEDY:
void $Greedy(Sel,\ n,\ m,\ MatrizDistancias,\ dispersion)$
1 : $distancias \leftarrow \emptyset$
2 : $distancias[0] \leftarrow M[Sel[0]][Sel[1]]$
3 : $distancias[1] \leftarrow M[Sel[1]][Sel[0]]$
4 : **while** $Sel.size() < m$ **do**
5 : $Sel,\ distancias,\ dispersion \leftarrow CalculaMejorPunto(n,\ M)$
6 : **end while**

2.2 Búsqueda Local

Como algoritmo de BL para el MDD consideraremos el esquema del primer mejor. La representación será en forma de un conjunto de elementos seleccionados. Para ser una solución candidata válida, tiene que satisfacer las restricciones (ser un conjunto de tamaño m):

- No puede tener elementos repetidos.
- Ha de contener exactamente m elementos.
- El orden de los elementos no es relevante.

El entorno de una solución Sel está formado por las soluciones accesibles desde ella a través de un movimiento de intercambio.
Se utilizará una función de itercambio de puntos $IntercambiarPunto((i,j), Sel, d_{Sel}, MatrizDistancias)$ donde se intercambiará el punto $i \in Sel$, por el punto $j \notin Sel$, y al mismo tiempo se actualizarán las distancias entre los puntos seleccionados d restando las distancias con i y añadiendo la nueva distancia con j a través de la $MatrizDistancias$. A esto último lo denominamos *factorización* de la función objetivo en todos los casos. Una vez realizado el movimiento, se actualiza la solución actual y los valores de contribución de los elementos seleccionados al coste de dicha solución, y se comienza a explorar el nuevo entorno. La exploración del entorno se realizará en orden aleatorio.

En cada ejecución de la BL, se partirá de una solución inicial aleatoria y se detendrá la ejecución cuando no se encuentre mejora en todo el entorno o cuando se hayan realizado 100000 evaluaciones de la función objetivo, es decir, en cuanto se cumpla alguna de las dos condiciones.

La BL del Mejor explora todo el vecindario, las soluciones resultantes de los $m(n-m)$ intercambios posibles. Una vez generado el vecindario, aleatoriamente se irán seleccionando las soluciones resultantes y se comprobará si la dispersión mejora y si es así actualizaremos la solución actual por la nueva obtenida y volveremos a generar el nuevo vecindario y repetir el proceso. En cambio, si no mejora en ningún caso pararemos el proceso y nos quedaremos con la última solución obtenida.

Pseudocódigo BL:

ALGORITMO BL:

void $BL(Sel,\ n,\ m,\ MatrizDistancias,\ dispersion)$
1 : $restantes \leftarrow$ *todo* $p \notin Sel$
2 : $distancias \leftarrow SumaDistancias(Sel,\ m,\ M)$
3 : $dispersion \leftarrow CalculaDispersion(distancias)$
4 : $Sel_{aux} \leftarrow Sel$
5 : $distancias_{aux} \leftarrow distancias$
6 : **while** $evaluaciones < 100000$ && $!mejora$ **do**
7 : **for** *para cada punto* $u \in Sel$
8 : **for** *para cada punto* $v \in restantes$
9 : $Vecindario \leftarrow$ *tupla* (u, v)
10 : $MezclarVecindario()$
11 : **for** *para cada tuplat* $(u, v) \in Vecindario$
12 : $IntercambiarPunto(t_i,\ Sel_{aux},\ distancias_{aux},\ MatrizDistancias)$
13 : $dispersion_{candidata} \leftarrow CalculaDispersion(distancias_{aux})$
14 : **if** $dispersion_{candidata} > dispersion$
15 : $Sel \leftarrow Sel_{aux}$
16 : $distancias \leftarrow distancias_{aux}$
17 : $dispersion \leftarrow dispersion_{candidata}$
18 : $restantes \leftarrow restantes \bigcup \{u\}$
19 : $restantes \leftarrow restantes \setminus \{v\}$
20 : **else**
21 : $Sel_{aux} \leftarrow Sel$
22 : $distancias_{aux} \leftarrow distancias$
23 : $evaluaciones++$
24 : **end while**

A continuación se profundizará en la funcionalidad de estos dos algoritmos y en las funciones auxiliares en las que se apoyan.

3 Estructura de los Algoritmos empleados

3.1 Estructura Greedy

La función Greedy() que implementa el algoritmo Greedy primero el mejor, recibe como parámetros un $vector < int > \ puntos$ con los puntos seleccionados que formarán el conjunto Sel, un $int\ n$ que es el número de puntos que hay, un $int\ m$ que es el número de puntos necesarios para que Sel se considere una posible solución, una $double\ **M$ que es una matriz dinámica que contiene todas las distancias entre todos los puntos y $double\ dispersion$ que guardará la mejor dispersión obtenida. Entre estos parámetros pasaremos por referencia $puntos$ y $dispersion$ ya que los iremos actualizando progresivamente y nos será útil al finalizar el procedimiento y dar la solución.

Datos iniciales necesarios para introducir en el algoritmo:

- Dos puntos distintos seleccionados aleatoriamente entre 0 y $n-1$.
- El número de puntos n y el número de puntos a seleccionar m.
- Una matriz con las distancias entre los puntos siendo la posición (i, j) la distancia de i a j siendo la misma que la posición (j, i) debido a que es simétrica.

Seleccionamos dos puntos iniciales aleatorios ya que la dispersión entre dos puntos siempre es 0, por tanto si solo hubiera un punto el algortimo cogería como mejor el siguiente mejor punto el primero. Debido a esto decidí coger inicialmente dos aleatorios y ya que el algoritmo empiece a buscar un punto que mejore o que menos empeore la dispersión actual y que esta sea distinto de 0.

Siguendo el pseudocódigo de la sección 2.1. anterior, el algoritmo primero calcula las distancias entre los dos puntos iniciales y seguidamente entrará en el $while$ y llamará a la función $CalculaMejorPunto()$ hasta que el número de puntos seleccionados hasta el momento sea igual a m y finalmente obtendremos los puntos con menos dispersion entre sí y a su vez la dispersión que tienen.

Mi implementación del algoritmo se basa en la función $CalculaMejorPunto()$ que tiene la funcionalidad de dado un conjunto de puntos seleccionados añada el primer punto que mejore la dispersion o el punto que menos la empeore. A continuación explicaré con pseudocódigo su implementación:

Pseudocódigo Función CalculaMejorPunto:

CalculaMejorPunto:

void CalculaMejorPunto(distancias, n, Sel, M, dispersion)

1 : $mejora \leftarrow False$
2 : **for** $i < n$ && $!mejora$
3 : **if** $i \notin Sel$
4 : $distancias_{aux} \leftarrow distancias$
5 : **for** $j < Sel.size()$
6 : $distancias_{aux}[i] \leftarrow distancias_{aux}[i] \bigcup d_{i,j}$
7 : $distancias_{aux}[Sel.size()] \leftarrow distancias_{aux}[Sel.size()] \bigcup d_{j,i}$
8 : $dispersion_{aux} \leftarrow CalculaDispersion(distancias)$
9 : **if** $dispersion_{aux} < dispersion$
10 : $distancias \leftarrow distancias_{aux}$
11 : $Sel \leftarrow Sel \bigcup \{i\}$
12 : $dispersion \leftarrow dispersion_{aux}$
13 : $mejora \leftarrow True$
14 : **else if** $dispersion_{aux} < dispersion_{candidata}$
15 : $distancias_{candidata} \leftarrow distancias_{aux}$
16 : $punto_{candidato} \leftarrow i$
17 : $dispersion_{candidata} \leftarrow dispersion_{aux}$
18 : **if** $!mejora$
19 : $distancias \leftarrow distancias_{candidata}$
20 : $Sel \leftarrow Sel \bigcup \{punto_{candidato}\}$
21 : $dispersion \leftarrow dispersion_{candidata}$

La función consiste en ir punto por punto comprobando que no esté en seleccionados (Sel) y calcular la distancia desde ese punto i al resto e ir sumando la distancia correspondiente a los demás con respecto al punto i. Una vez realizado lo anterior calcularemos la dispersion con la función $CalculaDispersion()$ que la veremos más adelante, y compararemos si es mejor que la dispersion actual actualizaremos los seleccionados añadiendo el punto i, quedandonos con las nuevas distancias y con la nueva mejor dispersion, a su vez como ha mejorado la dispersion terminaremos con el procedimiento ya que calculamos el mejor punto. En caso contrario comprobaremos si mejora la dispersion candidata que representa la dispersion que menos empeora a la actual, si es así guardaremos el punto i como posible candidato y las distancias respecto a ese punto y su dispersion como candidata.

Una vez terminado de examinar todos los posibles puntos a seleccionar nos quedaremos con el marcado como candidato y lo añadiremos al conjuntode seleccionados Sel y actualizaremos la dispersion actual y las nuevas distancias.

De esta manera nos quedaremos con el primer punto que mejore la dispersion o con el que menos la empeore.

3.2 Estructura Búsqueda Local

La función BL() que implementa el algoritmo de Búsqueda Local recibe como parámetros un $vector < int > \ puntos$ con los puntos seleccionados que formarán el conjunto Sel, un $int\ n$ que es el número de puntos que hay, un $int\ m$ que es el número de puntos necesarios para que Sel se considere una posible solución, una $double\ **M$ que es una matriz dinámica que contiene todas las distancias entre todos los puntos y $double\ dispersion$ que guardará la mejor dispersión obtenida. Entre estos parámetros pasaremos por referencia *puntos* y *dispersion* ya que los iremos actualizando progresivamente y nos será útil al finalizar el procedimiento y dar la solución.

Datos iniciales necesarios para introducir en el algoritmo:

- m puntos aleatorios seleccionados previamente y que pertenezcan al conjunto de puntos.
- El número de puntos n y el número de puntos a seleccionar m.
- Una matriz con las distancias entre los puntos siendo la posición (i, j) la distancia de i a j siendo la misma que la posición (j, i) debido a que es simétrica.

Llegados a este punto podemos observar respecto a los parámetros, el parecido con *Greedy*. Sin embargo, en la Búsqueda Local partimos de una solución inicial generada aleatoriamente con el objetivo de ir minimizando la dispersión intercambiando puntos del *vecindario* con los seleccionados hasta que superemos 100000 evaluaciones de la función objetivo o que no se encuentre ningún *vecino* que mejore.

Analizando el pseudocódigo del algoritmo en la sección 2.2. podemos destacar la función $IntercambiaPunto()$ como la función clave del algoritmo ya que es la responsable del intercambio de un punto de seleccionados por otro del *vecindario* y además de realizar la factorización de la función objetivo como observaremos en el siguiente pseudocógico:

Pseudocódigo Función IntercambiaPunto:
IntercambiaPunto:
$void\ \ IntercambiaPunto(tupla(u, v),\ Sel,\ distancias,\ M)$
1 : **for** $i < Sel.size()$
2 : **if** $Sel[i] == u$
3 : $d_u \leftarrow distancias[i]$
4 : **else**
5 : $distancias[i] \leftarrow distancias[i] - d_{Sel[i],u}$
6 : $distancias[i] \leftarrow distancias[i] + d_{Sel[i],v}$
7 : $dtotal_v \leftarrow dtotal_v + d_{Sel[i],v}$
8 : $Sel \leftarrow Sel \backslash \{u\}$
9 : $Sel \leftarrow Sel \bigcup \{v\}$
10 : $distancias \leftarrow Sel \backslash \{d_u\}$
11 : $distancias \leftarrow Sel \bigcup \{dtotal_v\}$

La función consiste en dada una tupla (u, v) siendo $u \in Sel$ y $v \notin Sel$, un vector de distancias, un vector de seleccionados (Sel) y la matriz de distancias que intercambie u por v y actualice las distancias utilizando al factorización, es decir, que a la distancia total de cada punto con el resto se le vaya restando la distancia con el punto u y se le sume la correspondiente con v a través de la matriz de distancias M.

Entendiendo esta función podremos explicar el funcionamiento de la función de Búsqueda Local. Al comienzo generaremos el conjunto de puntos no seleccionados iniciales llamado *restantes*, seguidamente calcularemos la suma de las distancias de todos los puntos de Sel con todos mediante la función $SumaDistancias()$ que veremos más tarde, y calcularemos la dispersión con la función $CalculaDispersion()$.

Mientras no lleguemos a 100000 evaluaciones o no mejoremos la dispersión repetiremos el siguiente proceso:

1. Crearemos el *Vecindario* con todas las posibles combinaciones de intercambios a través de un vector de tuplas (u, v) $(u \in Sel$ y $v \notin Sel)$, donde u será intercambiado en el conjunto de seleccionados (Sel) por v.

2. Mezclaremos las tuplas con la función $random\_shuffle$ de la libreria *algorithm* para que en el siguiente paso al seleccionar tuplas haya aleatoriedad.

3. Seleccionaremos una tupla hasta que tras llamar a $IntercambiarPunto$ y seguidamente a $CalculaDispersion$ comprobemos si la dispersión mejora.

    - En el caso de que mejore actualizaremos la dispersión actual a la última calculada, el conjunto Sel al nuevo con el intercambio de la tupla y el vector nuevo de distancias. Cambiamos mejora a $True$ para seguir generando el nuevo *Vecindario* y paremos de examinar el actual. Finalmente a *restantes* añadiremos v y eliminaremos u.
    - En el caso de que no mejore restableceremos los cambios realizados en las variables auxiliares Sel_{aux} y $distancias_{aux}$ con la última solución.

4. Por cada tupla que examinamos le aplicamos la función de evaluación, es decir, hemos realizado una evaluación por tanto incrementamos su variable en 1.

3.3 Funciones Auxiliares

En esta sección voy a comentar funciones auxiliares de las cuales $Greedy()$ y $BL()$ se apoyan en su implementación:

3.3.1 CalculaDispersion()

Pseudocódigo Función CalculaDispersion:
CalculaDispersion:
void $CalculaDispersion(distancias,\ dispersion)$
1 : $maximo \leftarrow Max(distancias)$
2 : $minimo \leftarrow Min(distancias)$
3 : $dispersion \leftarrow maximo - minimo$
Esta función utiliza las funciones $*max\_element()$ y $*min\_element()$ de la biblioteca *algorithm* para calcular el elemento máximo y mínimo del vector de distancias para luego hacer la diferencia entre ambos y obtener la dispersión.

3.3.2 SumaDistancias()

Pseudocódigo Función SumaDistancias:
SumaDistancias:
void $SumaDistancias(distancias,\ Sel,\ m,\ M)$
1 : **for** $i < m$
2 : $distancias[i] = 0$
3 : **for** $j < m$
4 : $distancias[i] \leftarrow distancias[i] + M[Sel[i]][Sel[j]]$

Esta función dado un conjunto de puntos seleccionados Sel y M la matriz simétrica de distancias calcula la distancia de todos los puntos con todos los demás. Siendo m el número de puntos de Sel.

4 Procedimiento considerado para desarrollar la práctica

Antes de comenzar la práctica decidí realizarla en C++ ya que es el lenguaje con el que estoy más familiarizado teniendo recientes los conocimientos adquiridos en la asignatura de ED y el uso de la STL, aunque también me he apoyado en algunas funciones de la librería *algorithm*.

Primero comencé con la lectura de datos en el main para asegurarme de que la lectura de datos de los archivos es correcta. En un primer momento me resulto cómodo utilizar un vector de la STL de tipo Elemento que es un struct que declaré que me guardaba la fila i y la columna j de la matriz y su valor que en este caso es la distancia entre los puntos i y j. También me intereso guardar en Elemento en número de puntos n y el número de puntos a seleccionar m.

Una vez leído bien los datos declaré una matriz dinámica M donde inserté las distancias y a su vez la convertí en una matriz simétrica por comodida a la hora de acceder a las distancias.

Teniendo ya todos los datos y habiendo leído las diapositivas comencé a implementar el algoritmo Greedy y seguidamente la Búsqueda Local (ambos ya los he comentado en las secciones anteriores). Para la implementación no he seguido un pseudocódigo concreto, me he centrado en la idea y en el funcionamiento de cada algoritmo y las ideas que pensaba las iba implementando y probando su funcionamiento. En un principio no he tenido problemas con los conceptos que implementaba de primeras salvo error código como de acceder mal a la matriz de distancias o de reservar memoria. En Búsqueda Local inicialmente lo planteé de manera q iba sustituyendo los puntos conforme mejoraban y me pasaba al siguiente y hacia lo mismo. Esto me pasó porque no entendia muy bien el concepto de generar el vecindario pero fui a la clase de prácticas para dudas y lo acabé entendiendo y lo implementé finalmente.

Por último para comprobar el correcto funcionamiento utilicé un archivo "prueba.txt" que almacenaba los datos para 7 puntos y sus respectivas distancias ya que al ser menos puntos es más sencillo de ver paso a paso lo que hacen los algoritmos y con salidas de pantalla de resultados en las funciones y realizando un ejemplo a mano creo que ambos algoritmos tienen un funcionamiento correcto y se ajustan a sus descripciones.

Después de tener todo correcto decidí hacer un archivo para cada algoritmo y uno extra para las funciones que comparten llamado "utilidades.cpp".
A la hora de hacer las ejecuciones junto a un compañero hicimos un script para que guardara en un archivo .txt los resultados de ejecutar 5 veces cada caso y así evitar hacerlas a mano. El script devuelve el archivo resultados.txt con todas las salidas del programa que son el nombre del caso ejecutado y el coste y tiempo medio de cada algoritmo.

5 Experimentos y análisis de resultados

A continuación voy a insertar imágenes con las tablas resultantes de aplicar cada algoritmo 5 veces a cada uno de los 50 casos. En main.cpp genero un array con 5 semillas 1,2,3,4,5, para obtener los mismos resultados para cada ejecución.

N° casos:	50		

Caso	*n*	*m*	Mejor coste
GKD-b_1_n25_m2	25	2	0
GKD-b_2_n25_m2	25	2	0
GKD-b_3_n25_m2	25	2	0
GKD-b_4_n25_m2	25	2	0
GKD-b_5_n25_m2	25	2	0
GKD-b_6_n25_m7	25	7	12,71796
GKD-b_7_n25_m7	25	7	14,09875
GKD-b_8_n25_m7	25	7	16,76119
GKD-b_9_n25_m7	25	7	17,06921
GKD-b_10_n25_m7	25	7	23,26523
GKD-b_11_n50_m5	50	5	1,9261
GKD-b_12_n50_m5	50	5	2,12104
GKD-b_13_n50_m5	50	5	2,36231
GKD-b_14_n50_m5	50	5	1,6632
GKD-b_15_n50_m5	50	5	2,85313
GKD-b_16_n50_m15	50	15	42,74578
GKD-b_17_n50_m15	50	15	48,10761
GKD-b_18_n50_m15	50	15	43,19609
GKD-b_19_n50_m15	50	15	46,41245
GKD-b_20_n50_m15	50	15	47,71511
GKD-b_21_n100_m10	100	10	13,83202
GKD-b_22_n100_m10	100	10	13,66434
GKD-b_23_n100_m10	100	10	15,34538
GKD-b_24_n100_m10	100	10	8,64064
GKD-b_25_n100_m10	100	10	17,20051
GKD-b_26_n100_m30	100	30	168,72959
GKD-b_27_n100_m30	100	30	127,09726
GKD-b_28_n100_m30	100	30	106,37919
GKD-b_29_n100_m30	100	30	137,45316
GKD-b_30_n100_m30	100	30	127,47974
GKD-b_31_n125_m12	125	12	11,74514
GKD-b_32_n125_m12	125	12	18,78893
GKD-b_33_n125_m12	125	12	18,5316
GKD-b_34_n125_m12	125	12	19,48833
GKD-b_35_n125_m12	125	12	18,11242
GKD-b_36_n125_m37	125	37	155,43477
GKD-b_37_n125_m37	125	37	198,89462
GKD-b_38_n125_m37	125	37	187,96703
GKD-b_39_n125_m37	125	37	168,5902
GKD-b_40_n125_m37	125	37	178,19374
GKD-b_41_n150_m15	150	15	23,34608
GKD-b_42_n150_m15	150	15	26,7895
GKD-b_43_n150_m15	150	15	26,75447
GKD-b_44_n150_m15	150	15	25,93559
GKD-b_45_n150_m15	150	15	27,77301
GKD-b_46_n150_m45	150	45	227,74931
GKD-b_47_n150_m45	150	45	228,6029
GKD-b_48_n150_m45	150	45	226,74534
GKD-b_49_n150_m45	150	45	226,40961
GKD-b_50_n150_m45	150	45	248,85662

?figurename? 1: Mejor coste por caso

Cada caso se ha ejecutado 5 veces y cada ejecución con diferente semilla. Después de las 5 ejecuciones se calcula la costes medio y el tiempo medio para poder comparar los resultados para el mismo caso de ambos algoritmos. Los resultados están representados en la Figura 2 para la Búsqueda Local y en la Figura 3 para Greedy.

Podemos observar que para los 5 primeros casos obtenemos 0 de desviación ya que el coste para dos 2 es 0 debido a la dispersión entre ellos que es 0 también.

Analizando ambos algoritmos podemos destacar que la Búsqueda Local obtiene resultados más cercanos al mejor coste que podemos observar en la Figura 1, en cambio Greedy salvo en algunos casos obtiene resultados de mayor coste medio por lo general pero con la pecularidad de que los obtiene en un tiempo mucho menor comparado al que necesita Búsqueda Local.

Esto último se debe a la filosofía que sigue cada algoritmo, aunque los dos se basan en el primero mejor, la clave está en la construcción de la solución inicial como he comentado en secciones anteriores donde Greedy solo necesita encontrar $m - 2$ puntos que mejoren la dispersión por lo cual implica menos iteracciones y Búsqueda Local realiza muchas más por lo general, ya que estará seleccionando puntos que intercambiandolos por uno de la solución mejore la dispersión hasta que no haya más que la mejore.

En conclusión a la hora de decidir que algoritmo escoger reflexionaremos si necesitamos una solución que obtenga un menor costo o si necesitamos una solución en menos tiempo. Según la decisión que tomemos en el primer caso escogeriamos Búsqueda Local y en el segundo Greedy.

Algoritmo BL			
Caso	**Coste medio obtenido**	**Desv**	**Tiempo**
GKD-b_1_n25_m2	0,0000	0,00	3,92E-05
GKD-b_2_n25_m2	0,0000	0,00	4,78E-05
GKD-b_3_n25_m2	0,0000	0,00	4,31E-05
GKD-b_4_n25_m2	0,0000	0,00	4,22E-05
GKD-b_5_n25_m2	0,0000	0,00	4,44E-05
GKD-b_6_n25_m7	39,2021	67,56	4,23E-04
GKD-b_7_n25_m7	27,9700	49,59	3,45E-04
GKD-b_8_n25_m7	38,5928	56,57	3,38E-04
GKD-b_9_n25_m7	39,8396	57,16	3,24E-04
GKD-b_10_n25_m7	38,1981	39,09	5,02E-04
GKD-b_11_n50_m5	30,2544	93,63	6,44E-04
GKD-b_12_n50_m5	14,6676	85,54	4,27E-04
GKD-b_13_n50_m5	16,8537	85,98	4,51E-04
GKD-b_14_n50_m5	16,1220	89,68	4,62E-04
GKD-b_15_n50_m5	31,6256	90,98	4,61E-04
GKD-b_16_n50_m15	121,1840	64,73	3,13E-03
GKD-b_17_n50_m15	101,3580	52,54	3,49E-03
GKD-b_18_n50_m15	101,1020	57,27	3,17E-03
GKD-b_19_n50_m15	120,8550	61,60	4,52E-03
GKD-b_20_n50_m15	110,7570	56,92	3,36E-03
GKD-b_21_n100_m10	40,4112	65,77	3,48E-03
GKD-b_22_n100_m10	31,8462	57,09	4,04E-03
GKD-b_23_n100_m10	39,2590	60,91	3,77E-03
GKD-b_24_n100_m10	40,0255	78,41	2,46E-03
GKD-b_25_n100_m10	44,3333	61,20	2,82E-03
GKD-b_26_n100_m30	396,5660	57,45	2,44E-02
GKD-b_27_n100_m30	363,1230	65,00	2,43E-02
GKD-b_28_n100_m30	370,4030	71,28	2,30E-02
GKD-b_29_n100_m30	316,0250	56,51	2,96E-02
GKD-b_30_n100_m30	300,1200	57,52	1,98E-02
GKD-b_31_n125_m12	36,1022	67,47	1,28E-02
GKD-b_32_n125_m12	43,7893	57,09	7,16E-03
GKD-b_33_n125_m12	60,8093	69,53	5,92E-03
GKD-b_34_n125_m12	44,7439	56,44	7,22E-03
GKD-b_35_n125_m12	67,8362	73,30	6,77E-03
GKD-b_36_n125_m37	330,4210	52,96	6,11E-02
GKD-b_37_n125_m37	480,3190	58,59	4,41E-02
GKD-b_38_n125_m37	542,6080	65,36	4,06E-02
GKD-b_39_n125_m37	316,5290	46,74	5,84E-02
GKD-b_40_n125_m37	419,8180	57,55	7,46E-02
GKD-b_41_n150_m15	67,3402	65,33	1,32E-02
GKD-b_42_n150_m15	78,0787	65,69	1,73E-02
GKD-b_43_n150_m15	70,8865	62,26	1,20E-02
GKD-b_44_n150_m15	73,5100	64,72	1,71E-02
GKD-b_45_n150_m15	64,3543	56,84	1,23E-02
GKD-b_46_n150_m45	570,1420	60,05	9,49E-02
GKD-b_47_n150_m45	498,8750	54,18	8,79E-02
GKD-b_48_n150_m45	505,2170	55,12	6,22E-02
GKD-b_49_n150_m45	519,4510	56,41	9,21E-02
GKD-b_50_n150_m45	496,8310	49,91	8,97E-02

Algoritmo BL	
Media Desv:	**56,71**
Media Tiempo:	**1,96E-02**

?figurename? 2: Desviación media y tiempo medio de BL.

Algoritmo Greedy			
Caso	**Coste medio obtenido**	**Desv**	**Tiempo**
GKD-b_1_n25_m2	0,0000	0,00	8,02E-07
GKD-b_2_n25_m2	0,0000	0,00	8,97E-04
GKD-b_3_n25_m2	0,0000	0,00	9,90E-04
GKD-b_4_n25_m2	0,0000	0,00	6,93E-04
GKD-b_5_n25_m2	0,0000	0,00	6,00E-07
GKD-b_6_n25_m7	49,2781	74,19	9,59E-05
GKD-b_7_n25_m7	70,0438	79,87	8,20E-05
GKD-b_8_n25_m7	42,8261	60,86	8,19E-05
GKD-b_9_n25_m7	78,3335	78,21	8,14E-05
GKD-b_10_n25_m7	80,0830	70,95	7,71E-05
GKD-b_11_n50_m5	26,7260	92,79	8,42E-05
GKD-b_12_n50_m5	29,4615	92,80	8,66E-05
GKD-b_13_n50_m5	36,7743	93,58	1,00E-04
GKD-b_14_n50_m5	24,6666	93,26	8,13E-05
GKD-b_15_n50_m5	28,2984	89,92	8,24E-05
GKD-b_16_n50_m15	181,7840	76,49	4,50E-04
GKD-b_17_n50_m15	187,2190	74,30	4,82E-04
GKD-b_18_n50_m15	168,9950	74,44	4,86E-04
GKD-b_19_n50_m15	193,3450	76,00	6,10E-04
GKD-b_20_n50_m15	173,2890	72,47	4,51E-04
GKD-b_21_n100_m10	75,3892	81,65	4,89E-04
GKD-b_22_n100_m10	71,4620	80,88	5,04E-04
GKD-b_23_n100_m10	74,1665	79,31	5,43E-04
GKD-b_24_n100_m10	69,8125	87,62	5,60E-04
GKD-b_25_n100_m10	82,0937	79,05	5,84E-04
GKD-b_26_n100_m30	439,7110	61,63	2,96E-03
GKD-b_27_n100_m30	431,9330	70,57	3,83E-03
GKD-b_28_n100_m30	457,5030	76,75	2,72E-03
GKD-b_29_n100_m30	325,5970	57,78	3,14E-03
GKD-b_30_n100_m30	455,2380	72,00	2,68E-03
GKD-b_31_n125_m12	107,1910	89,04	1,25E-03
GKD-b_32_n125_m12	69,6034	73,01	1,21E-03
GKD-b_33_n125_m12	103,7390	82,14	9,14E-04
GKD-b_34_n125_m12	63,9608	69,53	8,60E-04
GKD-b_35_n125_m12	84,8231	78,65	9,63E-04
GKD-b_36_n125_m37	458,5780	66,11	5,30E-03
GKD-b_37_n125_m37	463,8730	57,12	5,73E-03
GKD-b_38_n125_m37	545,3780	65,53	5,20E-03
GKD-b_39_n125_m37	447,0590	62,29	5,58E-03
GKD-b_40_n125_m37	415,8630	57,15	7,40E-03
GKD-b_41_n150_m15	109,1780	78,62	1,39E-03
GKD-b_42_n150_m15	142,6770	81,22	2,04E-03
GKD-b_43_n150_m15	103,7920	74,22	1,78E-03
GKD-b_44_n150_m15	137,1490	81,09	1,80E-03
GKD-b_45_n150_m15	112,6590	75,35	2,03E-03
GKD-b_46_n150_m45	538,0100	57,67	1,12E-02
GKD-b_47_n150_m45	578,9720	60,52	8,91E-03
GKD-b_48_n150_m45	448,7150	49,47	8,22E-03
GKD-b_49_n150_m45	559,0420	59,50	8,48E-03
GKD-b_50_n150_m45	643,3480	61,32	1,05E-02

Algoritmo Greedy	
Media Desv:	**66,54**
Media Tiempo:	**2,29E-03**

?figurename? 3: Desviación media y tiempo medio de Greedy.