Metaheurísticas(2019-2020)
Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Técnicas de Búsqueda Local y Algoritmos Greedy para el Problema de la Máxima Diversidad (MDP)

Alberto Jesús Durán López
DNI: 54142189-M
albduranlopez@gmail.com

Grupo Jueves, 17:30 - 19:30

21 de marzo de 2020

Índice

1. Introducción

Durante el transcurso de la asignatura trabajaremos con el problema de la máxima diversidad *(Max Diversity Problem)*.
En particular, en esta práctica estudiaremos el funcionamiento de *Técnicas de Búsqueda Local y de los Algoritmos Greedy* para la resolución del problema en cuestión.
Comentaremos todos los pasos y problemas encontrados, detallando minuciosamente todos los detalles y solución a los mismos.

Además, se incorporarán tablas para mostrar los resultados de ambas ejecuciones y gráficas para contrastar ambos modelos (optimización, costes y desviación).

2. Problema de la máxima Diversidad

El problema de la máxima diversidad (*maximum diversity problem*, MDP) es un problema de optimización combinatoria consistente en seleccionar un subconjunto de m elementos ($|M| = m$) de un conjunto inicial N de n elementos (con $n > m$) de forma que se maximice la diversidad entre los elementos escogidos.

El **MDP** se puede formular como:

$$\text{Maximizar: } z_{MS}(x) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_i x_j$$

$$\text{Sujeto a: } \sum_{i=1}^{n} x_i = m \quad \text{con } x_i = \{0,1\},\ i = 1, ..., n \quad \text{donde:}$$

- x 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

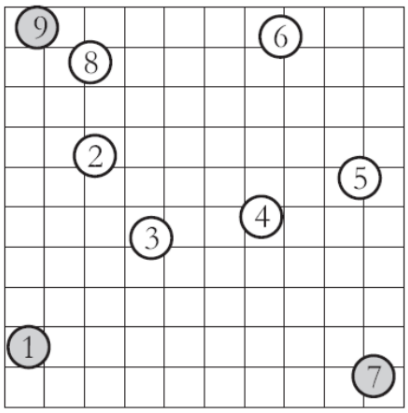


Figura 2.1: MDP-Maximum Diversity Problem

3. Datos y casos considerados

Las ejecuciones se han realizado en un ordenador `Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz, 16GB RAM, 512 SSD`.

Para la realización de las prácticas usaremos el lenguaje de programación `C++` ya que se deben probar muchos ejemplos y la ejecución es más rápida al ser un lenguaje de programación compilado.

Se utilizarán **30 casos** seleccionados de varios de los conjuntos de instancias disponible en la MDPLIB, 10 pertenecientes al grupo `GKD` con distancias Euclideas, n=500 y m=50 *(GKD-c_11_n500_m50 a GKD- c_20_n500_m50)*, 10 del grupo `MDG` con distancias reales en [0,1000], n=500 y m=50 *(MDG-b_1_n500_m50 a MDG-b_10_n500_m50)*; y otras 10 del grupo `MDG` con distancias enteras en 0,10, n=2000 y m=200 *(MDG-a_31_n2000_m200 a MDG- a_40_n2000_m200)*.

Tras la ejecución de los algoritmos se han rellenado dos tablas con sus respectivos resultados *(MDP_greedy.ods, MDP_BL.ods)* y, además, un archivo extra donde se ha añadido un resumen acompañado con gráficas que explicaremos a lo largo de la práctica. *(gráficas.ods)*. Adicionalmente, se han añadido a la entrega dos ficheros: *Output.txt* y *Output_O2.txt* donde se han volcado los resultados de la ejecución de todos los ficheros requeridos con y sin optimización `-O2`, respectivamente.

4. Algoritmo

En la siguiente sección, estudiaremos los algoritmos implementados, *Greedy* y *Búsqueda Local*.

4.1. Descripción de la Función Objetivo

Tanto el algoritmo *Greedy* como *Búsqueda Local* hacen uso de la misma función objetivo, la comentada en el punto (2) anterior y cuyo pseudocódigo mostramos a continuación:

Datos: CosteEstimado(*vector sel, matriz distancias)*
inicio
  $suma \leftarrow 0$;
  para i *in* $|sel| - 1$ **hacer**
    para $j \leftarrow i + 1$ *in* $|sel|$ **hacer**
      $suma \leftarrow distancias[i][j] + suma$;
    fin
  fin
  devolver *suma*
fin

Algoritmo 1: Evaluar solución

Sin embargo, en el algoritmo *Búsqueda Local* realizamos una pequeña modificación ya que es necesario calcular la contribución de cada elemento independientemente, y por tanto, el coste de la solución se podría llamando a la anterior función `CosteEstimado` o bien sumando las contribuciones independientes de todos los elementos que forman la solución.

Datos: ContribucionIndep(*int ind, vector sel, matriz distancias)*
inicio
  suma $\leftarrow$ *0* ;
  para *j in sel* **hacer**
    suma $\leftarrow$ *distancias*[*ind*][*j*] + *suma* ;
  fin
  devolver *suma*
fin

Algoritmo 2: Contribución Independiente

4.2. Greedy

El algoritmo *greedy* del MDP se basa en la heurística de ir seleccionando los elementos más lejanos a los previamente seleccionados. Para ello, se parte del elemento más alejado del resto en el conjunto de elementos. En los $m-1$ pasos siguientes se va escogiendo el elemento más lejano a los elementos seleccionados hasta el momento. Cabe destacar que en nuestros casos del problema no disponemos de los valores concretos de los elementos sino solo de las distancias entre ellos, luego el algoritmo está levemente modificado respecto al original.

Añadimos primeramente el código que devuelve la distancia entre los dos conjuntos que utiliza esta heurística (`Dist2Conj`), es decir, el conjunto de *Seleccionados*, de tamaño m y el conjunto de *No Seleccionados*, de tamaño $n-m$. A partir del vector de seleccionados y no seleccionados, recorremos ambos contenedores para obtener así el valor mínimo de la distancia de cada s_i a s_j con $s_i \in Unsel$ y $s_j \in Sel$. Una vez llenado el vector con estos valores, devolvemos el máximo de todos ellos.

Datos: Dist2Conj(*vector Sel, vector unSel, matriz Distancias)*
inicio
  double suma $\leftarrow$ *0* ;
  para *it in unSel* **hacer**
    para *it2 in Sel* **hacer**
      suma $\leftarrow$ *distancias*[$*it$][$*it2$] + *suma* ;
    fin
    si *suma*>*maximo* **entonces**
      maximo $\leftarrow$ *suma*;
      indice $\leftarrow$ $*it$;
    fin
    suma $\leftarrow$ 0 ;
  fin
  devolver *indice*
fin

Algoritmo 3: Dist2conj (Distancia entre los dos conjuntos, Sel-UnSel)

Ahora bien, nos centramos en la función principal de nuestro algoritmo, la llamaremos *Greedy*. Dividiremos su implementación en dos partes:

1. Insertamos en el vector **acum** la distancia acumulada de cada elemento de la matriz de distancias al resto.

   Claramente, la matriz de distancias resultante tras leer los ficheros `.txt` proporcionados es simétrica y por ende, sólo es necesario rellenar la diagonal superior o inferior. Sin embargo, para facilitar las operaciones, la rellenamos entera y, la distancia acumulada se puede calcular con un doble bucle `for`.

2. Introducimos en el vector de seleccionados aquellos elementos que devuelvan la distancia entre el vector de *seleccionados* y *no seleccionados*. Estos índices se obtienen llamando a la función `Dist2Conj`, previamente explicada.

Datos: Greedy(*integer m, matriz distancias*)
inicio
  vector $sel, unSel$;
  vector $acum \leftarrow$ `DistanciaAcumulada`$(distancias)$
  integer $s_i \leftarrow$ `max`$(acum)$;
  $sel \leftarrow sel \cup s_i$;
  $unSel \leftarrow unSel - \{s_i\}$;
  mientras $|sel| < m$ **hacer**
    $s_i^* \leftarrow$ `dist2conj`$(sel, unSel, distancias)$;
    $sel \leftarrow sel \cup s_i^*$;
    $unsel \leftarrow unsel - s_i^*$;
  fin
fin

Algoritmo 4: Greedy MDP

4.3. Búsqueda Local

Para la implementación del algoritmo de Búsqueda Local, consideramos el esquema disponible en el *Seminario 2* de la asignatura. Explicaremos el algoritmo con minuciosidad adjuntando trozos de psudocódigo reflejando lo realizado. La semilla usada para todas las ejecuciones ha sido `54142189`, pudiéndose ésta indicar en la ejecución del archivo en cuestión.

1. Primeramente partimos de una solución obtenida de forma aleatoria, es decir, con un vector *sel* de tamaño $m \leq n$ donde n representa el orden de la matriz de distancias.

```
Datos: SolucionInicial(integer n, integer m)
inicio
    unordered_set sel ;
    mientras |sel| < m hacer
        random = rand() %n ;
        sel ← s_random ;
    fin
    devolver sel ;
fin
```

Algoritmo 5: SolucionInicial

   La estructura usada para almacenar los índices de los elementos en seleccionados es *unordered_set* que, como bien sabemos, no almacena repetidos y, como consecuencia, la solución inicial estará formada por m índices diferentes.

2. `Función de Generación j`: En cada iteración, seleccionamos el elemento del vector de seleccionados `i` que menor contribución aporte y lo intercambiamos por otro elemento `j` del entorno con mejor heurística.

   En el contenedor *todasJ* introducimos todos los índices `j` que se han generado, de forma que si la nueva solución no mejora respecto a la anterior, no se vuelve a generar el mismo valor `j` y, como consecuencia, no se generan valores repetidos.

```
inicio
    integer j ;
    mientras j ∈ sel   or   se ha generado antes hacer
        j ← generar nuevo j ;
    fin
    todasJ ← todasJ ∪ j ;
    devolver j ;
fin
```

Algoritmo 6: Generar índice j

Descripción del entorno: En nuestro problema MDP, el parámetro `i` indica el índice del elemento que se eliminará de la solución y el j por cual se sustituirá. Por tanto, las opciones para `i` serán `m` (los m elementos seleccionados) y las opciones para `j` serán `n-m` (los elementos disponibles en *no seleccionados*), por tanto, el entorno estará formado por `m·(n-m)` elementos.

Si el intercambio es favorable, es decir, si el coste de la nueva solución (sumando las distancias del nuevo elemento y restando las del elemento anterior) es mayor que la anterior, aceptamos el intercambio. En caso contrario, rechazamos.

Repetiremos este proceso hasta que se realicen 100.000 evaluaciones de la función objetivo o cuando no encuentre mejora en el entorno.

Retomando la explicación del contenedor *todasJ*, explicaremos su utilidad en la función para evaluar el entorno. Si este contenedor está vacío significa que el anterior intercambio(i,j) fue favorable y sólo es necesario encontrar el s_i que menor contribución aporte. En caso contrario, si *todasJ* está lleno (tiene n-m elementos) hay que pasar al **siguiente** elemento que menor contribución aporte.

Datos: EvaluaVecinos*(pair<vector,int>inicial,*
matriz distancias)
inicio
mejora ← *True*;
eval ←*0, siguienteMin* ← *0* ;

mientras *eval<100.000 and mejora* **hacer**
si $|todasJ| = 0$ **entonces**
s_i ← `min_element(contribuciones)` ;
siguienteMin ← *0* ;
fin
else si $|todasJ| \neq 0$ *and* $siguienteMin < m$ **entonces**
siguienteMin ← *siguienteMin+1* ;
s_i ← `next_min_element(contribuciones)` ;
fin
else si *entorno explorado* **entonces**
mejora ← *False* ;
return ;
fin

salir ← False, $c \leftarrow 0$;
mientras $c < n - m$ *and not salir* **hacer**
Genera elemento válido j ;
si `Contrib(`s_j`)>Contrib(`s_i`)` **entonces**
Int(sel, i, j) ;
salir ← True;
coste ← $coste + contribNueva - contribAntigua$
fin
$c \leftarrow c + 1$;
$eval \leftarrow eval + 1$;
fin
fin
fin

Algoritmo 7: EvaluaVecinos

Resumidamente, el algoritmo a seguir es el siguiente:

- Antes de llamar a la función `EvaluaVecinos`, tenemos una solución aleatoria. En cada iteración tomamos el elemento s_i que menor contribución aporte de todos y el objetivo será generar el elemento s_j a intercambiar.
- Primero, la búsqueda del indice `j` se hará tras fijar i. Haciendo uso de la función `rand()`, generaremos valores que no se hayan generado para el valor i fijado. Si la contribución del nuevo elemento es mayor que la del anterior, aceptamos el cambio. El conjunto de j generados se guardaran en el vector $todasJ$ de forma que no se generen dos índices iguales y así se pueda generar todo el entorno de forma correcta y sin repetidos.
- Si para el valor i que menos contribución aporte no se encuentra ningún j cuyo intercambio sea aceptado, pasaremos al siguiente i que menor contribución aporte. Este proceso, que se indica en el algoritmo anterior en el segundo if, cuando $|todasJ| \neq 0$ y $siguienteMin < m$, es decir, para cuando el valor i no se encuentre ningún j a intercambiar y cuando no se hayan recorrido los m índices de *seleccionados*, respectivamente. Así pues, este proceso se repetirá hasta recorrer todos los valores posibles de n y m, es decir, $m \cdot (n - m)$
- Por último, cuando se haya recorrido todo el vector de *seleccionados* y no haya encontrado mejora en el entorno, saldremos de la función.

4.3.1. Optimización

Adelantamos que únicamente el algoritmo de *Búsqueda Local* con los ficheros `MDG-a` tiene un tiempo de ejecución algo superior, en torno a un segundo más que el resto. Esto es debido a que en dichos ficheros se trabaja con una matriz de distancias regular de tamaño 2000 y, por ello, el entorno es mayor.

Para reducir el tiempo de ejecución, se ha compilado el código con la opción de optimización -O2 y, además, se ha realizado una factorización del coste para obtener una mayor eficiencia.

En cada iteración, podemos calcular la diferencia de costes entre las dos soluciones recalculando todas las distancias de la función objetivo, sin embargo, esto no es necesario ya que como únicamente añadimos y quitamos 1 elemento, basta con sumar y restar la distancia del nuevo y del viejo elemento al resto de elementos seleccionados, respectivamente. Además, combinamos la factorización del coste con el cálculo de la contribución de los elementos para mejorar más aún la eficiencia.

Mostramos, por último, la función llamada `BusquedaLocal`. En ella, vemos que la estructura elegida para almacenar la solución ha sido un *pair*, donde en la primera componente guardamos el contenedor *sel* y en la segunda el coste de la solución evaluada referente a dicho *sel*.

Primero, llamamos a la función `SolucionInicial` y `CosteSol` para obtener una solución aleatoria y su coste, respectivamente; y después a la función `EvaluaVecinos`, que se encargará de encontrar la mejor solución mediante la *Búsqueda Local*.

Datos: BusquedaLocal(*integer* m, *matriz distancias*)
inicio
    $pair < vector, double > inicial$;
    $inicial.first \leftarrow$ `SolucionInicial(n,m)` ;
    $inicial.second \leftarrow$ `CosteSolGeneral(inicial.first, distancias)` ;
    `EvaluaVecinos(inicial,distancias)` ;
fin

Algoritmo 8: BusquedaLocal

5. Resultados

Mostramos un gráfico global con los resultados obtenidos tras las ejecuciones. En él, reflejamos los costes obtenidos en los algoritmos *Greedy* y *Búsqueda Local*, así como sus respectivas desviaciones (frente al mejor coste) y sus tiempos de ejecución medidos en segundos. Además, incluímos una tabla a modo resumen con los resultados globales.

Archivo	Mejor coste	Greedy	Desv.	Time	BL	Desv.	Time
GKD-c_11_n500_m50	19587,12891	19583,9	0,02	0,01	19579	0,04	0,03
GKD-c_12_n500_m50	19360,23633	19349,2	0,06	0,01	19355,6	0,02	0,04
GKD-c_13_n500_m50	19366,69922	19348,8	0,09	0,01	19359,7	0,04	0,05
GKD-c_14_n500_m50	19458,56641	19458,4	0,00	0,01	19458,5	0,00	0,03
GKD-c_15_n500_m50	19422,15039	19429,3	0,01	0,01	19415,9	0,03	0,03
GKD-c_16_n500_m50	19680,20898	19677,3	0,01	0,01	19677,4	0,01	0,03
GKD-c_17_n500_m50	19331,38867	19331,4	0,00	0,01	19326	0,03	0,03
GKD-c_18_n500_m50	19461,39453	19461,4	0,00	0,01	19456	0,03	0.03
GKD-c_19_n500_m50	19477,32813	19472,8	0,02	0,01	19477,3	0,00	0,02
GKD-c_20_n500_m50	19604,84375	19604,8	0,00	0,01	19596,2	0,04	0,03
MDG-b_1_n500_m50	778030,625	754035	3,08	0,01	754287	3,05	0,16
MDG-b_2_n500_m50	779963,6875	759843	2,58	0,01	771778	1,05	0,13
MDG-b_3_n500_m50	776768,4375	757355	2,50	0,01	760587	2,08	0,12
MDG-b_4_n500_m50	775394,625	763216	1,57	0,01	765833	1,23	0,13
MDG-b_5_n500_m50	775611,0625	753857	2,80	0,01	753444	2,86	0,12
MDG-b_6_n500_m50	775153,6875	756626	2,39	0,01	758023	2,21	0,12
MDG-b_7_n500_m50	777232,875	757362	2,56	0,01	756698	2,64	0,13
MDG-b_8_n500_m50	779168,75	759994	2,46	0,01	764679	1,86	0,13
MDG-b_9_n500_m50	774802,1875	754166	2,66	0,01	763851	1,41	0,13
MDG-b_10_n500_m50	774961,3125	753517	2,77	0,01	767725	0,93	0,13
MDG-a_31_n2000_m200	114139	112505	1,43	1,22	112980	1,02	1,28
MDG-a_32_n2000_m200	114092	112425	1,46	1,19	112882	1,06	1,37
MDG-a_33_n2000_m200	114124	112375	1,53	1,18	112094	1,78	1,42
MDG-a_34_n2000_m200	114203	112262	1,70	1,32	112728	1,29	1,33
MDG-a_35_n2000_m200	114180	112558	1,42	1,51	112777	1,23	1,29
MDG-a_36_n2000_m200	114252	112355	1,66	1,46	112832	1,24	1,40
MDG-a_37_n2000_m200	114213	112453	1,54	1,55	112534	1,47	1,39
MDG-a_38_n2000_m200	114378	112495	1,65	1,43	112830	1,35	1,28
MDG-a_39_n2000_m200	114201	112283	1,68	1,34	112789	1,24	1,28
MDG-a_40_n2000_m200	114191	112709	1,30	1,42	113149	0,91	1,28

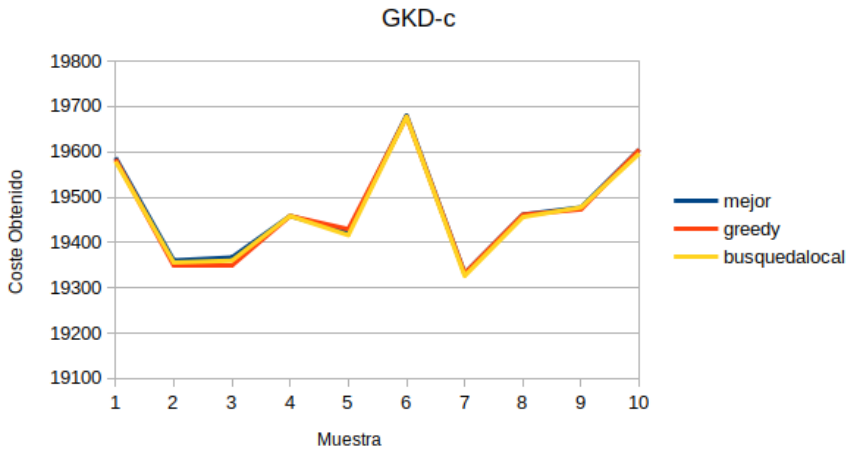
Tabla 5.1: Tabla Resultados Globales

Algoritmo	*Desv*	*Tiempo*
Greedy	1,37	0,46
BL	1,07	0,49

5.1. Comparación de costes obtenidos

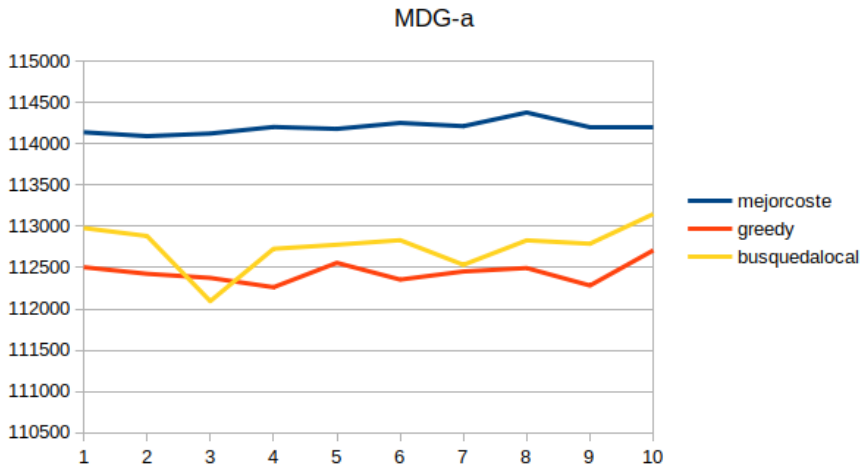
Recordemos los casos seleccionados para la ejecución de los algoritmos:
Tomamos 10 pertenecientes al grupo `GKD` con distancias Euclideas, 10 del grupo `MDG` con distancias reales y otras 10 del grupo `MDG` con distancias enteras. Mostramos los resultados obtenidos en 3 gráficas para reflejar mejor las diferencias:

- `GKD-c`: Conjunto de datos, referentes al grupo de distancias Euclideas con $n = 500$, $m = 50$, en el que mejor resultados se han obtenido, tanto el *greedy* como la *Búsqueda Local* llegan al mejor coste. Hay convergencia con ambos algoritmos y con todos los archivos, de hecho, se aprecia que las tres líneas están superpuestas, obteniendo prácticamente una desviación nula.

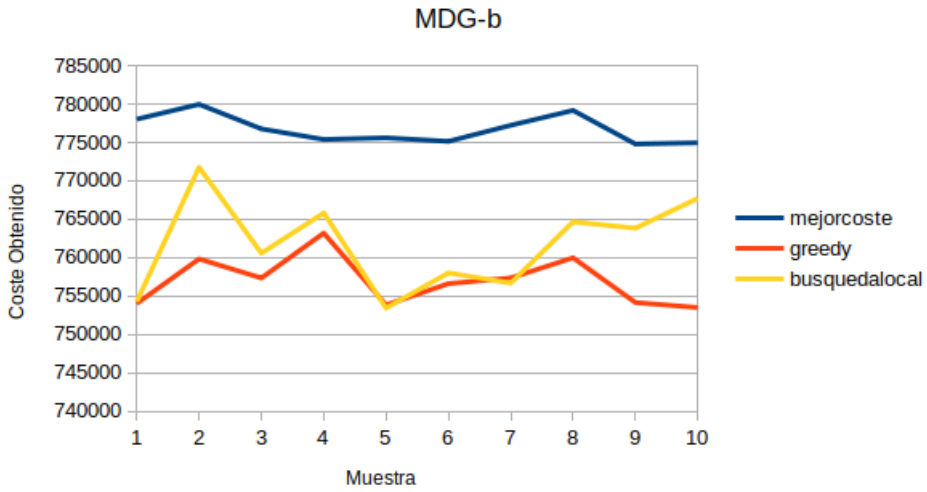


- `MDG-a`: Conjunto de datos correspondiente al grupo de distancias enteras y tamaño de $n = 2000$ y $m = 200$. En todas las muestras los costes obtenidos en el algoritmo de *Búsqueda Local* están por encima de los de *Greedy*, a excepción del archivo *MDG-a_ 33*

  No hay convergencia en el algoritmo de *BusquedaLocal*, de hecho, éste termina tras realizar 100.000 evaluaciones en la función objetivo. Para explorar el entorno completo se necesitarían $m \cdot (n - m)$ evaluaciones, es decir casi 4 veces más. Sin embargo, el tiempo se hubiera multiplicado por 4 y , por tanto, la calidad del algoritmo sería peor.

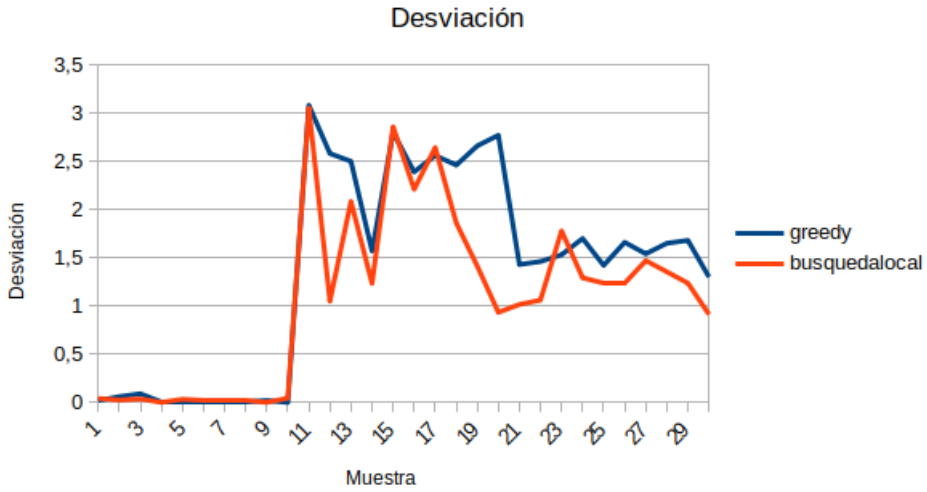


- `MDG-b`: Conjunto de datos correspondiente al grupo de distancias reales y tamaño $n = 500$ y $m = 50$. Al igual que en el caso anterior, el algoritmo de *Búsqueda Local* mejora los costes del algoritmo *greedy*. En este caso, en los archivos centrales del diagrama anterior, la *Búsqueda Local* no mejora los resultados del *greedy*, de hecho, son prácticamente iguales. Esto se debe a la existencia de óptimos locales en nuestro entorno, no permitiendo a nuestro algoritmo de BL salir de éste.



5.2. Desviación

Por último, reflejados en el siguiente gráfico, mostramos en las lineas azules la desviación de la ejecución de cada una de las muestras del algoritmo *Greedy* y, en las líneas naranjas, lo obtenido en la ejecución del algoritmo *Búsqueda Local* que, como bien vemos, se encuentra por debajo (menos en la Muestra 23).



Por otro lado, el área que deja el contorno de la línea naranja es menor que la de la línea azul, pero esto dependerá mucho de la existencia de óptimos locales, que depende a su vez de la solución inicial obtenida. En resumen, podemos considerar que la calidad del algoritmo *Búsqueda Local* es mejor porque en media obtiene soluciones más cercanas al mejor valor conocido.

Referencias

-Seminario 1
-Seminario 2
-Guión de prácticas