UNIVERSIDAD DE GRANADA

MH

Metaheurísticas

Curso: 3º **Grupo:** 1

Práctica 1
MDP

Autor: Mario Carmona Segovia
DNI: 45922466E **E-mail:** mcs2000carmona@correo.ugr.es

Profesor: Daniel Molina

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Curso 2020 - 2021

Índice

Índice de figuras

Índice de tablas

1. Descripción del problema

El Problema de la Máxima Diversidad (en ingles, Maximum Diversity Problem, MDP) consiste en la elección de un subconjunto de M puntos, de un conjunto mayor de N puntos. Este subconjunto debe maximizar la diversidad, Z; es decir, maximizar la distancia entre los elementos del subconjunto.

La fórmula para obtener la diversidad de un subconjunto es la siguiente:

$$Z = \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} d_{ij}$$

Siendo:

- i y j índices de los elementos del conjunto
- d_{ij} la distancia entre los puntos con índice i y j

2. Descripción de los aspectos comunes

2.1. Representación de las soluciones

En primer lugar cada elemento del conjunto total se ha etiquetado con un número entero positivo, empezando por el cero.

Como una solución a este problema es un subconjunto de este conjunto total, he representado el subconjunto como un lista de enteros.

En concreto esta lista está implementada con una estructura *Set*, ya que todas la operaciones que se realizan sobre la lista son muy eficientes con esta estructura de datos. Operaciones:

- Eliminación de un elemento en concreto
- Inserción de un elemento

Además esta estructura nos garantiza uno de los requisitos que debe cumplir una solución al problema, que es no tener elementos repetidos. Además la lista se encuentra siempre ordenada.

El resto de elementos no seleccionado como solución también se encuentran guardados en una lista con las mismas características.

2.2. Función objetivo

La función objetivo valora como de buena es una solución. La calidad se mide como la suma total de todas las distancias entre los elementos que forman la solución. Una solución es de mayor calidad cuanto mayor es la distancia total, ya que a mayor distancia mayor diversidad en la solución.

Algorithm 1 Función objetivo

Input: *solucion*: lista con la etiquetas de los elementos que forman la solución.
Input: *distancias*: matriz que guarda la distancia entre cada par de elementos del conjunto total.
Output: *distancia_total*: es la distancia total que existe en la *solucion*.

```
1:
2: distancia_total ← 0.0
3: /* Calculamos la suma de todas la distancias entre los elementos
4:  de la solución */
5: solucion_size ← número de elementos en la solucion
6: for i ∈ {0, ..., solucion_size − 1} do
7:     for j ∈ {i + 1, ..., solucion_size} do
8:         distancia_total ← sumar distancia entre i y j
9: return distancia_total
```

Además de esta versión de la función objetivo se ha creado otra función objetivo que además de calcular la distancia total de una solución, calcula la contribución de cada elemento en la distancia total, es decir, calcula la distancia total de cierto elemento con el resto de elementos que conforman la solución. La distancia entre el elemento x e y será considerada tanto para la contribución de x como de y, de esta manera se aprovechan las operaciones realizadas para calcular la función objetivo.

Algorithm 2 Función objetivo (extendida)

Input: *solucion*: lista con la etiquetas de los elementos que forman la solución.
Input: *distancias*: matriz que guarda la distancia entre cada par de elementos del conjunto total.
Output: *distancia_total*: es la distancia total que existe en la *solucion*.
Output: *contribuciones*: vector de flotantes que indican la contribución de cada elemento a la *distancia_total*.

```
2: distancia_total ← 0.0
   /* Calculamos la suma de todas la distancias entre los elementos
4:  de la solución */
   solucion_size ← número de elementos en la solucion
6: for i ∈ {0, ..., solucion_size − 1} do
       for j ∈ {i + 1, ..., solucion_size} do
8:         distancia_total ← sumar distancia entre i y j
           contribuciones ← sumar contribución al elemento i
10:        contribuciones ← sumar contribución al elemento j
   return distancia_total, contribuciones
```

2.3. Lectura de datos

Los datos utilizados para el problema están guardados en un archivo de texto plano. Este archivo contiene los siguiente datos:

- Nº de elementos del conjunto
- Nº de elementos que debe tener la solución
- Distancia entre cada par de elementos del conjunto

El formato de la información es el siguiente:

```
1 {Número de elementos del conjunto} {Número de elementos que debe tener la solución}
2 {Elemento 0} {Elemento 1} {Distancia entre elemento 0 y elemento 1}
3 {Elemento 0} {Elemento 2} {Distancia entre elemento 0 y elemento 2}
4 ...
```

Para obtener esta información y colocarla en una estructura de datos más manejable para el programa, he creado la siguiente función:

Algorithm 3 Leer archivo

Input: *nombre_archivo*: nombre del archivo de texto plano que contiene la información.
Output: *num_elem_selec*: número de elementos que debe tener la solución al problema.
Output: *distancias*: matriz de distancias entre pares de elementos del conjunto.

```
    archivo ← abrir el archivo con nombre "nombre_archivo"
 3: numElemTotal ← obtener el número de elementos del conjunto completo
    num_elem_selec ← obtener el número de elemento que debe tener la solución
    distancias ← crear la matriz de distancias
 6: while !final_archivo do
        i, j ← obtener el par de elementos
        distancias ← añadir distancia entre el par de elementos
 9: return num_elem_selec, distancias
```

3. Descripción de la Búsqueda Local

3.1. Funciones comunes

3.1.1. Operador de generación de vecino

Como operador de generación de vecino se ha utilizado el intercambio, que consiste en elegir un elemento de la solución y otro elemento no seleccionado para la solución, e intercambiarlos de listas. Su implementación en pseudocódigo es la siguiente:

Algorithm 4 Intercambio

Input: *solucion_actual*: lista con las etiquetas de los elementos que forman la solución actual.
Input: *elemento_a_sustituir*: índice del elemento que se quiere sustituir.
Input: *no_seleccionados*: lista con las etiquetas de los elementos que no forman parte de la solución.
Input: *elemento_a_incluir*: índice del elemento que se quiere incluir en la solución.
Input: *distancias*: matriz que guarda la distancia entre cada par de elementos del conjunto total.
Input: *contribuciones_actual*: vector de flotantes que contiene las contribuciones de cada elemento de la solución actual.
Output: *nueva_solucion*: lista con las etiquetas de los elementos que forman la nueva solución.
Output: *nuevas_contribuciones*: vector de flotantes que contiene las contribuciones de cada elemento de la nueva solución.

```
contri_elem_a_incluir ← 0
nuevas_contribuciones ←contribuciones_actual
/* Modificar las contribuciones de los elementos que no son
intercambiados */
for i ∈ solucion_actual do
    if i != elemento_a_sustituir then
        nuevas_contribuciones ← restar distancia del elemento i al elemento_a_sustituir
        nuevas_contribuciones ← sumar distancia del elemento i al elemento_a_incluir
        contri_elem_a_incluir ← añadir la distancia sumada a la contribución del nuevo elemento
nueva_solucion ← intercambiamos el nuevo elemento por el elemento elegido para ser sustituido
nuevas_contribuciones ← intercambiar también las contribuciones de los elementos intercambiados
return nueva_solucion, nuevas_contribuciones
```

3.1.2. Factorización de la BL

Dado que que la creación de un nuevo vecino sólo modifica un elemento de la solución original, por lo que el coste de la solución sólo es alterado por las distancias de ambos elementos involucrados en el operador de intercambio, parece mejor opción realizar una factorización de la función objetivo; es decir, en vez de volver a calcular todas las distancias de la nueva solución, sólo tenemos que restar las distancias del elemento sustituido al resto de elementos de la solución y

sumar las distancias del elemento incluido al resto de elementos de la solución. Su implementación en pseudocódigo es la siguiente:

Algorithm 5 Función objetivo factorizada

Input: *solucion*: lista con las etiquetas de los elementos que forman la solución.
Input: *elemento_sustituido*: índice del elemento que se ha sustituido.
Input: *elemento_incluido*: índice del elemento que se ha incluido en la solución.
Input: *distancias*: matriz que guarda la distancia entre cada par de elementos del conjunto total.
Input: *coste_actual*: suma de la distancia total de la solución actual.
Output: *nuevo_coste*: suma de la distancia total de la solución modificada.
Output: *mejorado*: variable booleana que indica si ha mejorado el coste de la función.
Output: *elemento_incluido*: índice del elemento que se ha incluido en la solución.

```
    nuevo_coste ←coste_actual
    /* Modificar el coste de la nueva solución */
    for i ∈ solucion do
5:      if i != elemento_sustituido then
            nuevo_coste ← restardistanciadelelementosialelemento_sustituido
            nuevo_coste ← sumardistanciadelelementosialelemento_incluido)
    /* Indicar si se ha mejorado el coste o no */
    if nuevo_coste > coste_actual then
10:     mejorado ← true
    else
        mejorado ← false
    return mejorado, elemento_incluido, nuevo_coste
```

3.1.3. Generación de soluciones aleatorias

Tanto las soluciones de la Búsqueda Local del Primero Mejor como de la Búsqueda Local del Mejor se crean de forma aleatoria.

En el caso del Mejor se crea de forma aleatoria la solución inicial, y no se crea de forma aleatoria las soluciones vecinas, ya que en este algoritmo se examina todo el entorno, por lo que es absurdo gastar tiempo en crear una solución aleatoria cuando se van a examinar todas.

Para crear la solución inicial se van eligiendo elementos del conjunto de forma aleatoria hasta tener el número de elementos necesario para forma una solución. El resto de elementos del conjunto se introducen en la lista de no seleccionados.

En el caso del Primero Mejor se crea de forma aleatoria tanto la solución inicial, como las soluciones vecinas, ya que en este caso se cogerá como vecina la primera solución que mejora la actual, por lo que no se exploran siempre todos los vecinos.

La solución inicial se genera de la misma forma que en del Mejor. La solución vecina se crea sustituyendo el elemento que menos aporta por uno de los elementos no seleccionados, para que la creación sea aleatoria antes de empezar a generar vecinos para ver cuál mejora la solución

actual, se baraja de forma aleatoria los elementos no seleccionados, para que a la hora de recorre los elementos de forma iterativa sean aleatorios los elementos que nos vayamos encontrando, esta acción se realiza con la función *shuffle*.

3.2. Búsqueda Local del Primero Mejor

3.2.1. Método de exploración del entorno

En esta variante de la Búsqueda Local se van generando vecinos hasta encontrar uno que mejore la solución actual, si se encuentra uno que la mejore se continua con la búsqueda; si no se encuentra un vecino que la mejore, se para el proceso de búsqueda y nos quedamos con la solución actual. Además se puede parar el proceso si se llega a un límite de iteraciones, y se incrementa una iteración por cada llamada a al función objetivo.

Algorithm 6 Método de exploración del entorno (BL_PM)

```
solucion_actual ← generar solución inicial aleatoria
iteraciones ← 0
hay_mejora ← true
while iteraciones ≤ 100000 & hay_mejora do
    noSeleccionadosNuevo ← barajar aleatoriamente la lista de elementos no seleccionados
    hay_mejora, elegido ← obtener el primer vecino que mejora el coste de la solución
    /* Si se encuentra un vecino que mejore el coste */
    if hay_mejora then
        solucion_modificada ← intercambiar el elemento elegido por el elemento de la solución que menos
        contribuciones_modi ← actualizar contribuciones con el nuevo elemento de la solución
```

3.3. Búsqueda Local del Mejor

3.3.1. Método de exploración del entorno

En esta variante de la Búsqueda Local se genera todo el entorno de una solución, y se elige la solución que consiga una mayor mejora de la solución actual, si no se alcanza ninguna mejor se para el proceso de búsqueda. Además se puede parar el proceso si se llega a un límite de iteraciones, y se incrementa una iteración por cada llamada a al función objetivo. Su implementación en pseudocódigo es la siguiente:

Algorithm 7 Método de exploración del entorno (BL_M)

solucion_actual ← *generar solución inicial aleatoria*
iteraciones ← 0
hay_mejora ← *true*
while iteraciones ≤ 100000 & *hay_mejora* **do**
    noSeleccionadosNuevo ← *barajar aleatoriamente la lista de elementos no seleccionados*
    hay_mejora, elegido ← *obtener el vecino que mejora en mayor medida el coste de la solución*
    /* Si se encuentra un vecino que mejore el coste */
    if *hay_mejora* **then**
        solucion_modificada ← *intercambiar el elemento elegido por el elemento de la solución que menos*
        contribuciones_modi ← *actualizar contribuciones con el nuevo elemento de la solución*

4. Descripción del Greedy

4.1. Algoritmo de comparación

Para elegir que elemento incluir en la solución en cada momento se tiene en cuenta sus distancias con el resto de elementos.

En primer lugar al insertar el primer elemento se tiene en cuenta que el elemento insertado sea el que mayor distancia acumulada tenga.

Algorithm 8 Elegir primer elemento a insertar en la solución

Input: *distancias*: matriz que guarda la distancia entre cada par de elementos del conjunto total.
Input: *noSeleccionados*: lista de elementos no incluidos en la solución.
Output: *elegido*: elemento elegido para ser insertado.

```
distanciasAcu ← inicializar vector a 0.0
noSeleccionados_size ← número de elementos no seleccionados
/* Calcular la distancia acumulada de cada elemento no seleccionado */
for i ∈ {0, ..., noSeleccionados_size − 1} do
    for j ∈ {i + 1, ..., noSeleccionados_size} do
        distanciasAcu ← aumentar distancia acumulada del elemento i
        distanciasAcu ← aumentar distancia acumulada del elemento j
distAcuMax ← 0.0
elegido ← vacio
/* Elegir el elemento con mayor distancia acumulada */
for i ∈ {noSeleccionados} do
    if distAcuMax < {distancia acumulada de i} then
        disAcuMax ← asigna la distancia acumulada del elemento i
        elegido ← i
return elegido
```

En el resto de elementos a insertar lo que que se tiene en cuenta es que tenga la mayor distancia de las mínimas distancias entre los elementos de la solución y los elementos no seleccionados.

Algorithm 9 Elegir el resto de elementos a insertar en la solución

Input: *distancias*: matriz que guarda la distancia entre cada par de elementos del conjunto total.
Input: *solucion*: lista de elementos incluidos en la solución.
Input: *noSeleccionados*: lista de elementos no incluidos en la solución.
Output: *elegido*: elemento elegido para ser insertado.

```
    distanciaMin ← inicializar vector al máximo valor de un flotante
    /* Calcular la distancia mínima entre cada elemento de la solución y el resto de elementos
    no seleccionados */
    for i ∈ solucion do
        for j ∈ noSeleccionados do
            distanciaMin ← quedarse con lamínima distancia al elemento i
    distMax ← −1
    elegido ← vacio
    /* Elegir el elemento con mayor distancia mínima */
 9: for i ∈ solucion do
        if distMax < {distancia mínima de i} then
            distMax ← asigna la distancia mínima del elemento i
            elegido ← i
    return elegido
```

Se sigue el proceso de inserción de elementos hasta que la solución alcanza su tamaño necesario para ser válida como solución.

5. Desarrollo de la práctica

La práctica la he implementado en C++, sin hacer uso de frameworks.

Para tomar los tiempos he utilizado el código del timer que se nos proporciona con la práctica. El resto de código está totalmente implementado por mi.

5.1. Compilación

Para compilar utilizo un archivo Makefile. Al ejecutar "make" en el terminal, se lanza su regla general, que genera todos los ejecutables, los ficheros objeto, y las carpetas necesarias. En el caso de los ejecutables crea dos por cada algoritmo, uno para depurar, y el otro para tomar tiempos con la opción de compilación -O2.

Para más información del Makefile abrir el archivo makefile y ver los comentarios y reglas que hay en él, el makefile se encuentra en la raíz del proyecto.

5.2. Limpiar ficheros derivados de la compilación

Para eliminar todos estos ficheros también hago uso del Makefile con la regla "make clean".

5.3. Obtener resultados

Para obtener los resultados he creado un script bash que ejecuta cada algoritmo con todos los casos y da como resultado un archivo CSV con los costes y tiempos del algoritmo en cada caso.

Para obtener los resultados de todos los algoritmos hay que ejecutar "./script_tiempos.sh" o ejecutar "make tiempos". Recomiendo esta última ya que da permisos al archivo para poder ejecutarse.

Y si se quiere obtener sólo el resultado de un algoritmo en concreto, se puede hacer ejecutando "./script_tiempos.sh <nombre del algoritmo sin extensión>".

Ejemplo:

- ./script_tiempos.sh BL_M
- ./script_tiempos.sh BL_PM
- ./script_tiempos.sh Greedy

Además de los CSV hay un archivo Excel de cada algoritmo que calcular la desviación de cada caso, la desviación media y el tiempo medio. Este archivo hay que modificarlo a mano, es decir, copiar y pegar los resultados del CSV.

Si se quiere ejecutar el archivo ejecutable de alguno de los algoritmos se debe ejecutar de la siguiente forma:

./<nombre del ejecutable> <seed> <nombre del fichero de datos con su ruta>

5.4. Distribución de carpetas

El proyecto está divido en las siguientes carpetas:

- src ← contiene los archivos fuente
- include ← contiene los archivos de cabecera
- data ← contiene los archivos de datos de los distintos casos
- tablas ← contiene los CSV y Excel con los resultados
- bin ← contiene los ejecutables
- obj ← contiene los ficheros objeto

Además en la raíz del proyecto se encuentra el Makefile y el script de bash.

6. Experimentos y Análisis de resultados

6.1. Descripción de los casos y valores de parámetros

Los casos utilizados se agrupan por diferente cantidad de elementos para el conjunto y cantidad de elementos para la solución. Para obtener los tiempos se han utilizado casos con los siguientes tamaños:

- 500 elementos en el conjunto, y 50 elementos para la solución
- 2000 elementos en el conjunto, y 200 elementos para la solución
- 3000 elementos en el conjunto, y 300, 400, 500, ó 600 elementos para la solución

Los argumentos pasados a los algoritmos son los siguientes:

- Nombre del caso
- Semilla

Para todas las ejecuciones de los algoritmos con los distintos casos se ha utilizado la semilla con valor 0.

6.2. Resultados obtenidos

Todos los tiempos están expresados en segundos.

6.2.1. Búsqueda Local

Algoritmo Búsqueda Local del Primero Mejor		
Caso	**Desv**	**Tiempo**
MDG-a_1_n500_m50	20,22	0,000454
MDG-a_2_n500_m50	14,48	0,001168
MDG-a_3_n500_m50	15,76	0,001055
MDG-a_4_n500_m50	13,21	0,001590
MDG-a_5_n500_m50	16,10	0,000844
MDG-a_6_n500_m50	16,43	0,000983
MDG-a_7_n500_m50	14,60	0,001132
MDG-a_8_n500_m50	15,97	0,001122
MDG-a_9_n500_m50	13,74	0,001026
MDG-a_10_n500_m50	16,85	0,000674
MDG-b_21_n2000_m200	11,34	0,008486
MDG-b_22_n2000_m200	10,23	0,017221
MDG-b_23_n2000_m200	9,24	0,019949
MDG-b_24_n2000_m200	9,98	0,012718
MDG-b_25_n2000_m200	8,41	0,020578
MDG-b_26_n2000_m200	8,84	0,025828
MDG-b_27_n2000_m200	9,81	0,020848
MDG-b_28_n2000_m200	9,66	0,018468
MDG-b_29_n2000_m200	9,17	0,024955
MDG-b_30_n2000_m200	9,05	0,018864
MDG-c_1_n3000_m300	8,42	0,040963
MDG-c_2_n3000_m300	7,23	0,057552
MDG-c_8_n3000_m400	6,47	0,075587
MDG-c_9_n3000_m400	7,08	0,068703
MDG-c_10_n3000_m400	7,07	0,042103
MDG-c_13_n3000_m500	5,88	0,068206
MDG-c_14_n3000_m500	5,73	0,111800
MDG-c_15_n3000_m500	6,22	0,083328
MDG-c_19_n3000_m600	5,25	0,102310
MDG-c_20_n3000_m600	4,54	0,172809

Tabla 1: Resultados BL_PM

Algoritmo Búsqueda Local del Mejor		
Caso	**Desv**	**Tiempo**
MDG-a_1_n500_m50	20,22	0,000473
MDG-a_2_n500_m50	14,48	0,001239
MDG-a_3_n500_m50	15,76	0,001079
MDG-a_4_n500_m50	13,21	0,001600
MDG-a_5_n500_m50	16,10	0,000869
MDG-a_6_n500_m50	16,43	0,000996
MDG-a_7_n500_m50	14,60	0,001136
MDG-a_8_n500_m50	15,97	0,001155
MDG-a_9_n500_m50	13,74	0,001042
MDG-a_10_n500_m50	16,85	0,000676
MDG-b_21_n2000_m200	11,34	0,008763
MDG-b_22_n2000_m200	10,23	0,017400
MDG-b_23_n2000_m200	9,24	0,021633
MDG-b_24_n2000_m200	9,98	0,015570
MDG-b_25_n2000_m200	8,41	0,024610
MDG-b_26_n2000_m200	8,84	0,026611
MDG-b_27_n2000_m200	9,81	0,023620
MDG-b_28_n2000_m200	9,66	0,018935
MDG-b_29_n2000_m200	9,17	0,026574
MDG-b_30_n2000_m200	9,05	0,019576
MDG-c_1_n3000_m300	8,42	0,046387
MDG-c_2_n3000_m300	7,23	0,058848
MDG-c_8_n3000_m400	6,47	0,077928
MDG-c_9_n3000_m400	7,08	0,071305
MDG-c_10_n3000_m400	7,07	0,043744
MDG-c_13_n3000_m500	5,88	0,071115
MDG-c_14_n3000_m500	5,73	0,108981
MDG-c_15_n3000_m500	6,22	0,085142
MDG-c_19_n3000_m600	5,25	0,104157
MDG-c_20_n3000_m600	4,54	0,173443

Tabla 2: Resultados BL_M

6.2.2. Greedy

Algoritmo Greedy		
Caso	**Desv**	**Tiempo**
MDG-a_1_n500_m50	24,06	0,03239
MDG-a_2_n500_m50	24,79	0,03266
MDG-a_3_n500_m50	26,66	0,03237
MDG-a_4_n500_m50	22,77	0,03249
MDG-a_5_n500_m50	26,07	0,03215
MDG-a_6_n500_m50	25,80	0,03326
MDG-a_7_n500_m50	22,48	0,03264
MDG-a_8_n500_m50	24,73	0,03482
MDG-a_9_n500_m50	25,23	0,03303
MDG-a_10_n500_m50	24,27	0,03285
MDG-b_21_n2000_m200	13,14	1,49319
MDG-b_22_n2000_m200	12,37	1,50895
MDG-b_23_n2000_m200	12,60	1,46693
MDG-b_24_n2000_m200	12,67	1,49859
MDG-b_25_n2000_m200	12,68	1,48597
MDG-b_26_n2000_m200	12,01	1,50338
MDG-b_27_n2000_m200	11,59	1,50624
MDG-b_28_n2000_m200	12,18	1,48056
MDG-b_29_n2000_m200	12,98	1,49104
MDG-b_30_n2000_m200	12,72	1,47574
MDG-c_1_n3000_m300	10,57	5,17701
MDG-c_2_n3000_m300	10,21	5,10977
MDG-c_8_n3000_m400	8,67	8,52709
MDG-c_9_n3000_m400	8,72	8,65216
MDG-c_10_n3000_m400	8,55	8,54996
MDG-c_13_n3000_m500	7,22	12,87660
MDG-c_14_n3000_m500	7,18	12,91230
MDG-c_15_n3000_m500	7,05	12,94770
MDG-c_19_n3000_m600	6,46	18,30870
MDG-c_20_n3000_m600	6,42	18,37670

Tabla 3: Resultados Greedy

6.2.3. Resultados globales

Algoritmo Greedy	Desv	Tiempo
Greedy	15,09	4,22
BL_PM	10,57	0,03
BL_M	10,57	0,04

Tabla 4: Resultados Globales

6.3. Análisis de resultados

En los resultados globales hay una clara diferencia de tiempos entre los algoritmos de Búsqueda Local y el Greedy. Sin ver la tabla lo normal sería que el Greedy tardase menos tiempo, ya que es un algoritmo rápido, pero si nos fijamos en como elige el elemento a insertar en la solución vemos que es una búsqueda muy costosa y que aumenta de forma exponencial con el tamaño del conjunto de elementos, ya que para cada elección de un elemento es necesario calcular todas las distancias acumuladas y buscar el máximo; ó calcular la distancia mínima de cada elemento de la solución y buscar el máximo. En la siguiente gráfica se puede ver como evoluciona el tiempo conforme aumenta el tamaño del conjunto de elemento:

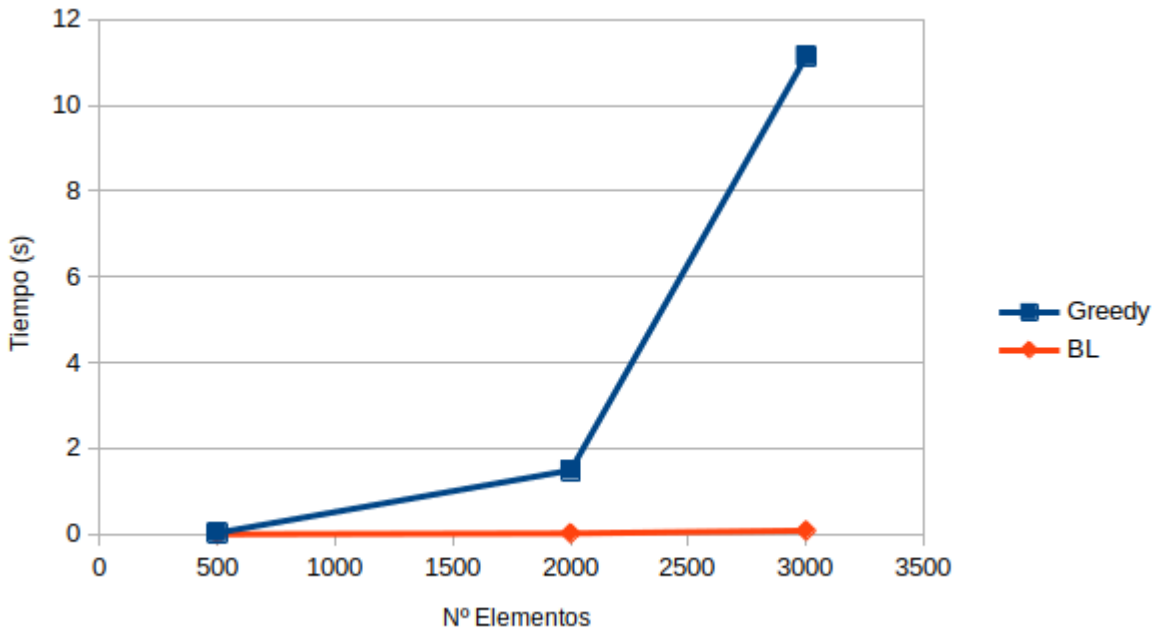

Figura 1: Evolución del tiempo - Greedy

En la gráfica se puede apreciar esa tendencia exponencial en el tiempo del Greedy. En la Búsqueda Local no ocurre debido a la factorización de la función objetivo.

Una vez explicada la causa de la diferencia de tiempo entre algoritmos, me queda explicar la

diferencia de desviación, es decir, la calidad de la solución, cuánto de cerca está la solución del algoritmo del óptimo del caso. En la Búsqueda Local se ha obtenido de media menos desviación que con Greedy, pero esto no quiere decir que sea mejor algoritmo para obtener soluciones de más calidad, ya que la calidad de la Búsqueda Local depende del valor de la semilla y del caso, a diferencia del Greedy que es independiente a la semilla y siempre obtiene la misma calidad en cada ejecución de un caso.

La dependencia de la Búsqueda Local a la semilla es debido a que la solución inicial y el orden en la comprobación de los elementos a insertar en la solución se hace de manera aleatoria. Por lo que puede que si ponemos otra semilla para esto casos se pueda obtener soluciones de menos calidad, o si mantenemos la semilla pero cambiamos de casos puede que también se obtengan peores soluciones de las esperadas.

Por lo que el Greedy nos proporciona un valor estable de calidad, mientras que la Búsqueda Local nos posibilita la obtención de soluciones de mayor calidad, pero no nos asegura obtenerlas siempre.

Por último cabe destacar que aunque parezca que los tres algoritmos mejoran la calidad de la solución conforme aumenta el tamaño de muestra, este efecto es debido a que la diferencia entre la solución obtenido con el algoritmo y la óptima cada vez es menor en proporción al coste óptimo. Por eso en los tamaños pequeños una pequeña diferencia entre ambas soluciones hace que la calidad sea baja, es decir, la desviación es alta.