Metaheurística grupo 2 (2021-2022)
Grado en Ingeniería Informática
Universidad de Granada

Práctica 1
Mínima dispersión diferencial

José María Ramírez González
jmramirez@correo.ugr.es
32899118X

4 de abril de 2022

Índice

1 Descripción del problema **4**

2 Elementos comunes empleados en la resolución del problema **5**
2.1 Representación de los datos 5
2.2 Función objetivo 6
2.3 Selección de los datos de entrada 6
2.4 Aleatoriedad en los algoritmos 7
2.5 Medición del tiempo de ejecución 7

3 Algoritmo greedy **8**
3.1 Funcionamiento del algoritmo 8
3.2 Orden de tiempo del algoritmo 9

4 Algoritmo de búsqueda local **10**
4.1 Funcionamiento del algoritmo 10
4.2 Orden de tiempo del algoritmo 11

5 Desarrollo y uso del software **12**
5.1 Implementación 12
5.2 Manual de uso 12

6 Experimentos realizados y análisis de resultados **14**

Índice de figuras

2.1 Función objetivo 6
6.1 Desviación y tiempos medios para todos los archivos 14
6.2 dispersión y tiempo por archivo 14
6.4 Diferencia mejor-peor caso por archivo en búsqueda local 15
6.3 Valores medios obtenidos en la ejecución por algoritmo 16
6.5 Mejores y peores valores obtenidos por algoritmo 17

Listings

1. Estructura de los ficheros proporcionados 5
2. Generalización de la lectura de los datos 5
3. Versión greedy 6
4. Versión búsqueda local 6
5. Versión búsqueda local 6
6. Tiempo de ejecución de un programa 7
7. Funcionamiento del algoritmo greedy 8
8. Funcionamiento del de búsqueda local 11

9. Formato del archivo results.txt 12

1. Descripción del problema

El problema escogido es el conocido como *mínima dispersión diferencial* o *minimun differential dispersion* en inglés.
Este problema trata de intentar minimizar la dispersión entre un subconjunto M de elementos de un conjunto C. Es un problema **NP-completo**, por lo que resulta inviable resolverlo de manera óptima en un tiempo decente en casos donde $|C|$ o $|M|$ sean grandes.
Nuestro objetivo es encontrar $M \subset C$, y se nos proporcionarán varios casos, cada uno con un C y un $m = |M|$. En cada caso nos dan, además, la distancia de un elemento al resto de ellos, tal que podamos calcular el Δ-*value* de cada elemento como $\Delta\text{-value}_i = \sum_{j \in M} d_{ij} \quad \forall\, i \in M$, donde d_{ij} es la distancia del elemento i al elemento j.
Para calcular la dispersión, realizamos el siguiente cálculo $disp = max(\Delta\text{-value}) - min(\Delta\text{-value})$. Esa sería la dispersión asociada a una posible solución M, el máximo de sus Δ-values menos el mínimo de ellos. Así pues, nuestro función a minimizar es esa.
Es importante destacar, que la dispersión de cada conjunto solución M depende de cada uno de los elementos $w \in M$ y que, cambiando tan solo uno de ellos, esta puede cambiar radicalmente.

2. Elementos comunes empleados en la resolución del problema

Para resolver este problema, hemos implementado dos algoritmos distintos:

- Un algoritmo **Greedy**.
- Un algoritmo de **Búsqueda local**.

Más adelante estudiaremos estos algoritmos, pero antes, vamos a comentar algunos elementos comunes a ambos.

2.1. Representación de los datos

Los datos de entrada se nos proporcionan en un fichero con la estructura que podemos apreciar en el listing 1.

```
n m
0 1 value
0 2 value
...
n-2 n-1 value
```

Listing 1: Estructura de los ficheros proporcionados

Donde $m = |M|$, $n = |C|$ y *value* es el coste de ir de la posición que aparece primero a la posición que aparece en segundo lugar.
En nuestro caso, guardaremos esta información en forma de matriz cuadrada $n \times n$, donde rellenamos el triangulo superior y el inferior, para no tener que comprobar índices, y dejaremos 0s en la diagonal principal.
El funcionamiento general sería el que observamos en el listing 2.

```
archivo = open(archivoConDatos)
n, m = archivo.get(0,1)

# Nueva matriz de tam n*n inicializada a 0
matriz = nuevaMatriz(n x n, 0)

for linea in archivo:
    pos1 = linea[0]
    pos2 = linea[1]
    valor = linea[2]
    matriz[pos1][pos2] = valor
    matriz[pos2][pos1] = valor
```

Listing 2: Generalización de la lectura de los datos

2.2. Función objetivo

Nuestro objetivo es minimizar la dispersión, calculada como $disp = max(\Delta\text{-value}) - min(\Delta\text{-value})$.
Para ahorrar tiempo de procesado, no vamos a calcular los Δ-value de cada conjunto solución, si no que iremos recalculando por cada elemento que incluyamos (versión *greedy*) o cambiemos (versión *búsqueda local*) en el conjunto solución. Es decir, si antes teníamos X Δ-values, ahora que hemos cambiado/añadido este elemento, modificamos cada Δ-value $\in X$ según corresponda.
Esto acelerará los cálculos al no tener que calcular todos los valores de 0.
Así pues, cambiar un elemento tal que la nueva dispersión $disp' < disp$ o añadir uno nuevo tal que $disp'$ sea lo más cercana a $disp$ posible, es decir, no aumente mucho la dispersión.
En la figura 2.1, tenemos los pseudocódigos para ambos casos del funcionamiento de la función objetivo.

```
hasta que |Sol|=m:
    para elem en elementos:
        si elem no en Sol:
            calc. valor
    Sol += elem con min(valor)
```

Listing (3) Versión greedy

```
hasta que no varie Sol, para todos elems:
    cambiar elem i opor elem j
    calcular nuevo valor
    si nuevo valor < valor ant:
        aceptar cambio
```

Listing (4) Versión búsqueda local

Figura 2.1: Función objetivo

2.3. Selección de los datos de entrada

Para seleccionar los datos de entrada, hacemos uso de la librería *os* con la que, introduciendo el nombre de la carpeta donde tenemos todos los archivos con los datos de entrada, nos leerá los nombres de los mismos.
Una vez tengamos los nombres, los ordenamos basándonos en los nombres de los mismos con una función que hemos creado y pasamos a ejecutar un bucle que realiza ambos algoritmos (*búsqueda local* y *greedy*) para cada uno.
Podemos encontrar el pseudocódigo en el listing 5.

```
archivos = leerCarpeta(nombreCarpeta)
ordenar(archivos)
para cada archivo en archivos:
    leer(archivo)
    Ejecutar algoritmos
```

Listing 5: Versión búsqueda local

2.4. Aleatoriedad en los algoritmos

Como se nos explica en el guión de la práctica, necesitamos incluir cierta aleatoriedad en los algoritmos.
Para controlarla y que los experimentos realizados puedan repetirse, hemos establecido las semillas previamente, que son similares para cada ejecución de cada algoritmo, es decir, para cada conjunto de datos, tenemos 5 (en nuestro caso) semillas, una para ejecución que realizaremos.
Así pues, para cada archivo que leemos, realizamos **5** iteraciones con los dos algoritmos, cada una con una semilla distinta.
La aleatoriedad se incluye de forma similar en cada algoritmo, siendo esta necesaria tan solo al comienzo del mismo.
En el caso del algoritmo *greedy*, se usa para seleccionar los 2 elementos iniciales con los que comenzaremos la ejecución.
En el caso del algoritmo de *búsqueda local*, se usa para seleccionar el conjunto solución inicial, del que partiremos para ejecutar el algoritmo, y para barajar el conjunto de elementos que no están en el conjunto solución.

2.5. Medición del tiempo de ejecución

Para medir los tiempos de ejecución de cada algoritmo, hemos hecho uso de la librería *time*, la cual nos proporciona el tiempo de CPU que tarda en ejecutarse una determinada parte del código con una implementación similar a la del listing 6.

```
tiempoinicio = tiempoActual()

...
#Ejecucion del algoritmo
...

tiempoFinal = tiempoACtual()
tiempoEjecucion = tiempoFinal - tiempoInicio
```

Listing 6: Tiempo de ejecución de un programa

3. Algoritmo greedy

Como primer algoritmo, hemos implementado uno de tipo voraz, este deberá ofrecer tiempos de ejecución realmente bajos, pero también soluciones que son, en algunos casos, bastante alejadas de la óptima.
El funcionamiento general por el que hemos optado consiste en seleccionar dos elementos aleatorios de la muestra e ir, a partir de estos dos, añadiendo nuevos elementos que nos garanticen la menor dispersión posible. Una vez hayamos llegado al tamaño m que se nos pide, terminamos la ejecución.

3.1. Funcionamiento del algoritmo

El algoritmo vamos a ejecutarlo en bucle, de forma que se ejecute hasta que tenemos m elementos en la solución, como hemos señalado anteriormente. Por cada iteración del mismo, calcularemos los Δ-values con los elementos que llevemos en la solución e iteraremos por todos aquellos elementos que no están en nuestra solución actual, comparando que pasaría con la dispersión si añadimos cada elemento a la solución actual, quedándonos al final con aquel que aumente lo menos posible la dispersión.
Podemos ver el pseudocódigo del algoritmo en el listing 7.

```
Hasta que longitud(Sol) distinto de m:
    calcular Delta-values

    por cada elemento disponible que no este en Sol:
        posibleSolucion = Sol + elemento
        posiblesDelta = actualizarDelta(posibleSolucion)

        si dispersion(posiblesDelta) < minimoActual:
            minimoActual = dispersion(posiblesDelta)
            elementoSeleccionado = elemento

    Sol += elementoSeleccionado
```

Listing 7: Funcionamiento del algoritmo greedy

En lo referente a la implementación, no contamos con funciones como sería *dispersión* o *actualizarDelta*, y lo hemos implementado directamente en el bucle, ya que es una operación que no resulta demasiado costosa y no ocupa un tamaño de código excesivo, no obstante, se podrían declarar como función si se quisiera.
Una vez que acabemos el bucle, puesto que no guardamos los Δ-values actualizados cuando añadimos el elemento, sería necesario recalcularlos todos. Se podría guardar este valor durante la ejecución del bucle, pero la mejora en tiempo de ejecución sería despreciable.

3.2. Orden de tiempo del algoritmo

En base al listing 7, si lo observamos con detalle, podemos destacar **dos** bucles importantes en el mismo:

- Un bucle *while*.
- Un bucle *for*.

A grandes rasgos, esto nos daría un tiempo teórico del orden $O\Big(m \cdot (n-m)\Big)$, no obstante, más adelante haremos un estudio empírico del mismo para verificar estos tiempo de ejecución.

4. Algoritmo de búsqueda local

Como se nos pide, hemos llevado a cabo también un algoritmo de búsqueda local, que debería ofrecer unos resultados más cercanos al óptimo teniendo un impacto algo mayor en el tiempo de ejecución en comparación con el voraz, pero sin resultar extremadamente lento.

4.1. Funcionamiento del algoritmo

En este caso, lo que realizaremos será comenzar con una solución aleatoria, e ir explorando esta solución realizando cambios entre elementos con aquellos que no formen parte de esta para mejorar el resultado. Estos cambios entre elementos se realizan uno a uno, no siendo posible intercambiar dos o más elementos a la vez.
Podemos verlo como explorar un árbol hasta llegar a un nodo hoja desde el que no podemos mejorar más la solución. En este árbol, cada nodo tendrá $m \cdot n$ padres y $m \cdot n$ hijos, siendo cada uno de ellos el resultado de intercambiar un elemento concreto por otro. Importante destacar que el padre de un nodo, también es hijo del mismo, ya que sería el resultado de volver a intercambiar los mismos elementos.
Una vez que pasemos a un hijo, no podremos volver al padre, es decir, no podemos realizar *backtracking*, resultando en un algoritmo del tipo "el primer mejor", ya que una vez que encontremos un hijo mejor que el padre, pasaremos a ese directamente.
A su vez, limitaremos la ejecución del algoritmo a un máximo de 100000 iteraciones, para evitar un bucle infinito.
Podemos ver el pseudocódigo del mismo, que corresponde a nuestra implementación, en el listing 8.

```
Barajar(posiblesOpciones)
Sol = Seleccionar(m, posiblesOpciones)

Mientras cambie y no superemos las maximas iteraciones:
    Barajar(posiblesOpciones):
    Para cada elementoSol en Sol:
        Para cada elemento en posibles opciones:
            nuevaSol = Int(Sol, elementoSol, elemento)
            nuevoCoste = RecalcularCoste(Sol, elementoSol, elemento)

            Si nuevoCoste menor que costeAnterior:
                Sol = nuevaSol
                costeAnterior = nuevoCoste
                posiblesOpciones.quitar(elemento)
                posiblesOpciones.meter(elementoSol)
                SalirBuclesFor

            iteracion += 1
```

Listing 8: Funcionamiento del de búsqueda local

La función *Int* que observamos en el listing anterior se usa para intercambiar el elemento *elementoSol*, presente en *Sol* por el elemento *elemento*.
La función *RecalcularCoste* no se encuentra implementada en nuestro programa, pero consta de varias líneas que lo que hacen es actualizar los costes de la nueva solución tan sólo realizando los cambios necesarios en los valores, resultando en una función bastante eficiente a nivel de tiempo.

4.2. Orden de tiempo del algoritmo

En base al listing 8, si lo observamos con detalle, podemos destacar **tres** bucles importantes en el mismo:

- Un bucle *while*.
- Dos bucle *for*.

A grandes rasgos, esto nos daría un tiempo teórico del orden $O\Big(100000 \cdot m \cdot (n-m)\Big)$, no obstante, más adelante haremos un estudio empírico del mismo para verificar estos tiempo de ejecución.

5. Desarrollo y uso del software

En esta sección vamos a comentar cómo hemos desarrollado el software para la práctica, a la vez que indicaremos como ejecutar el mismo.

5.1. Implementación

A la hora de implementar ambos algoritmos, hemos optados por usar *Python* debido a la sencillez del código, la simplicidad de uso, unos tiempos de ejecución decentes (más lentos que si usamos un lenguaje como *C*, pero correctos para este tipo de software) y, lo más importante, las librerías disponibles y el fácil manejo de tipos de datos como las listas y las matrices.
Ambos algoritmos están escritos en el mismo programa, por lo que no nos tendremos que preocupar de ejecutar varios archivos.
Junto con la memoria, incluiremos un archivo de requisitos en formato *.txt*, de forma que se pueda hacer uso de *pip* para instalar las librerías pertinentes, aunque a excepción de *NumPy*, las otras que usamos vienen instaladas por defecto, por lo que sólo tendríamos que instalar esta.
Es importante destacar que el ejecutable escribe a disco, dejando los resultados en un archivo de nombre *results.txt*, con el formato que se observa en el listing 9.

```
Greedy y BL: DispMedia DispMin DispMax TMedio TMin TMax
Para cada archivo en orden

Ejemplo:

45 25 60 0.7 0.45 1.1 45 25 60 0.7 0.45 1.1
45 25 60 0.7 0.45 1.1 45 25 60 0.7 0.45 1.1
45 25 60 0.7 0.45 1.1 45 25 60 0.7 0.45 1.1
```

Listing 9: Formato del archivo results.txt

5.2. Manual de uso

Para ejecutar correctamente el ejecutable (*.py*), tan sólo tenemos que hacer uso de *pip* para instalar los requisitos proporcionados en el archivo *requirements.txt* con el comando `pip install -r /path/to/requirements.txt`.
Una vez instalado, tenemos que asegurarnos de comprobar que tenemos en una carpeta localizada los archivos a leer y **solo** esos archivos, ya que el programa no realiza comprobación sobre los archivos que lee. Esta carpeta se la pasaremos por parámetros al momento de ejecución.
Ahora podemos pasar a ejecutar el archivo con el comando `python3 <nombreArchivo>.py <ruta carpeta datos>`.
Podremos observar los resultado en tiempo real de los valores medios de tiempo y dispersión para ambos algoritmos y, si queremos también los valores mínimo y máximo de

tiempo y dispersión, podremos abrir el archivo *resulsts.txt* que aparecerá en el mismo directorio del ejecutable.
Las semillas para controlar la aleatoriedad vienen definidas en el propio código en una variable llamada *seeds*[1], que se encuentra en la línea 46. Si cambiamos los valores, obtendremos resultados diferentes a la hora de ejecutar el programa, y si queremos optar por no utilizar semillas, podemos comentar la línea 46 y la línea 67, siendo esta última la que fija la semilla a utilizar.

[1]Las semillas utilizadas son 19102001 ,20102001, 19112001, 19102002 y 21112001

6. Experimentos realizados y análisis de resultados

Para ambos algoritmos, hemos realizado 5 ejecuciones por fichero, con las cuales hemos obtenido los valores que nos resultarán de interés.
Vamos primero a observar los valores medios en la figura 6.3 y comentarlos.
Como vemos, podemos ver que los resultados a nivel de tiempo medio son algo peores en el algoritmo de búsqueda local que en el voraz. No obstante, el impacto en el tiempo tampoco es exagerado, y obtenemos unos resultado bastante mejores de media.
Si nos fijamos en los valores óptimos, podemos calcular la dispersión para cada caso y con ello la dispersión media de cada algoritmo.
En la figura 6.1 podemos ver cuáles serían estos valores.

	Greedy	**BL**
Desviación	80,32	58,12
Tiempo	1,42E-02	2,77E-01

Figura 6.1: Desviación y tiempos medios para todos los archivos

Aquí, en la tabla 6.1, se aprecia perfectamente que la desviación media respecto al óptimo de la búsqueda local es menor que en el voraz y que el efecto del mismo sobre el tiempo no resulta significativo.
Estos valores pueden variar según la máquina en la que ejecutemos los algoritmos, pero el orden será similar.
Vamos a visualizar unas gráficas en la figura 6.2, comparando tiempos de ejecución y dispersión media de cada algoritmo por archivo.

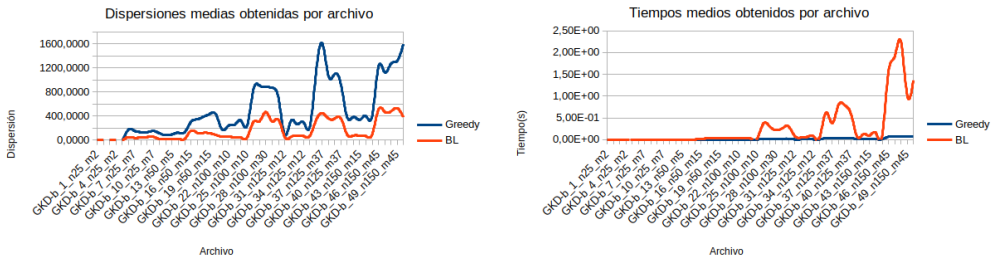

Figura 6.2: dispersión y tiempo por archivo

Como observamos, a nivel de dispersión, ambos algoritmos siguen la misma tendencia en todo momento, pero el de búsqueda local se mantiene por debajo del voraz siempre.
En cuanto a tiempos, ambos comienzan de manera similar, pero llega un punto en el que el de búsqueda local sube su tiempo de ejecución. No obstante, este aumento en el tiempo no resulta descabellado si tenemos en cuenta la mejoría en los resultados frente al voraz.

Vamos ver ahora los mejores y peores tiempos de ejecución y dispersión obtenidos, para asegurarnos para observar los mejores y peores casos que obtendremos de forma general. En la figura 6.5 podemos ver los mejores y peores casos de cada algoritmo.
Es destacable que los peores casos del algoritmo de búsqueda local siguen siendo muchísimo mejores que incluso el mejor caso del voraz a nivel de dispersión.
A nivel de tiempo, obviamente gana el voraz, pero, en mi opinión, si esto se tratara de un caso de uso real, sería mucho mejor sacrificar esos segundos en función de obtener un resultado mucho más fiel al óptimo.
Si observamos en la figura 6.4 la diferencia de dispersión entre los mejores y peores casos del algoritmo de búsqueda local, podemos ver que por norma general no tienen mucha diferencia y hay algunos casos en los que coinciden. Esto es buena señal, ya que nos indica que el algoritmo va a tener, de media, buenos resultados si aumentamos el número de casos del problema.

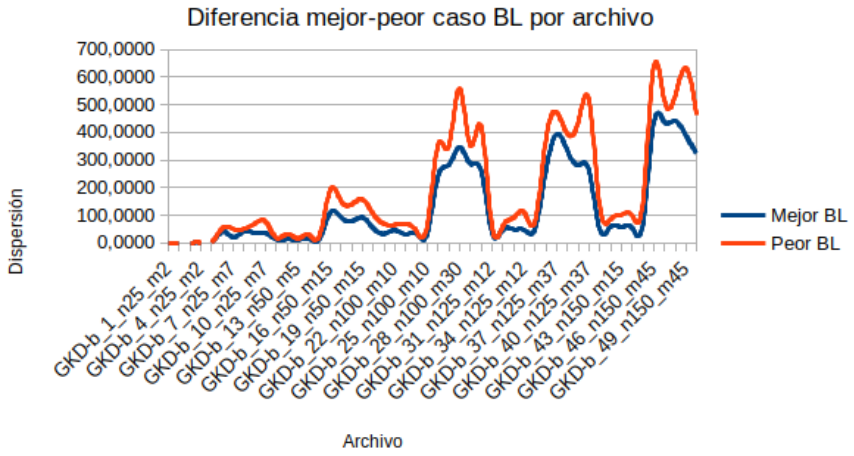

Figura 6.4: Diferencia mejor-peor caso por archivo en búsqueda local

Archivo	Optimo	Algoritmo greedy		Algoritmo BL	
		Tiempo	Dispersión	Tiempo	Dispersión
1	0	8,46E-06	0,0000	3,11E-04	0,0000
2	0	6,85E-06	0,0000	2,11E-04	0,0000
3	0	6,68E-06	0,0000	2,11E-04	0,0000
4	0	6,49E-06	0,0000	2,22E-04	0,0000
5	0	6,73E-06	0,0000	2,14E-04	0,0000
6	12,717	3,28E-04	172,9389	2,46E-03	49,0833
7	14,098	3,24E-04	151,4784	2,90E-03	34,5253
8	16,761	3,27E-04	132,5934	2,07E-03	44,7451
9	17,069	3,26E-04	131,4541	2,66E-03	50,3845
10	23,26	3,23E-04	151,2385	2,17E-03	52,6371
11	1,926	3,46E-04	110,6476	2,69E-03	15,0193
12	2,121	3,45E-04	80,4410	3,28E-03	19,5315
13	2,362	3,45E-04	93,6333	2,70E-03	14,1592
14	1,663	3,45E-04	120,3797	2,04E-03	19,9200
15	2,853	3,46E-04	129,4126	2,42E-03	19,0034
16	42,74	2,67E-03	302,2888	1,83E-02	154,7710
17	48,10	2,69E-03	344,0174	2,46E-02	120,4500
18	43,19	2,69E-03	387,2363	2,91E-02	118,8968
19	46,41	2,67E-03	432,7591	2,72E-02	113,3795
20	47,71	2,67E-03	427,7715	3,60E-02	84,4173
21	13,83	2,64E-03	171,7590	2,48E-02	53,7561
22	13,66	2,65E-03	240,6561	2,56E-02	59,2516
23	15,34	2,69E-03	253,9828	2,58E-02	44,0998
24	8,640	2,63E-03	322,2628	2,24E-02	39,4464
25	17,20	2,65E-03	238,7843	2,27E-02	42,1936
26	168,7	1,97E-02	839,3913	3,71E-01	299,9137
27	127,0	2,06E-02	901,5594	3,01E-01	315,1940
28	106,3	2,05E-02	889,6144	2,20E-01	465,3706
29	137,4	2,10E-02	865,4841	2,56E-01	310,0761
30	127,4	2,14E-02	700,4242	3,02E-01	340,4451
31	11,74	4,61E-03	86,3618	8,40E-02	51,3877
32	18,78	4,79E-03	316,8829	4,40E-02	52,1606
33	18,53	4,64E-03	262,1155	6,05E-02	74,9053
34	19,48	4,65E-03	298,9516	7,77E-02	66,0790
35	18,11	4,80E-03	214,9209	5,70E-02	64,5465
36	155,4	3,73E-02	1094,4853	6,23E-01	322,4048
37	198,8	3,72E-02	1606,5565	3,78E-01	448,1244
38	187,9	3,76E-02	1066,3734	7,96E-01	358,6058
39	168,5	3,75E-02	1097,4133	7,91E-01	355,2638
40	178,1	3,73E-02	924,1819	5,92E-01	359,0492
41	23,34	8,62E-03	378,9201	6,72E-02	90,0941
42	26,78	8,56E-03	383,6978	1,25E-01	73,7698
43	26,75	8,65E-03	335,2620	9,42E-02	74,5123
44	25,93	8,71E-03	399,9048	1,49E-01	65,7905
45	27,77	8,38E-03	405,1055	1,01E-01	95,5109
46	227,7	6,52E-02	1214,1638	1,55E+00	503,3408
47	228,6	6,48E-02	1125,7093	1,91E+00	472,3337
48	226,7	6,48E-02	1265,8778	2,25E+00	475,1340
49	226,4	6,43E-02	1316,0476	1,02E+00	530,0055
50	248,8	6,50E-02	1592,4987	1,36E+00	380,9427

Figura 6.3: Valores medios obtenidos en la ejecución por algoritmo

Algoritmo greedy			
Mejores		Peores	
Tiempo	Disp	Tiempo	Disp
6,31E-06	0,0	1,41E-05	0,0
6,42E-06	0,0	7,13E-06	0,0
6,17E-06	0,0	6,89E-06	0,0
6,11E-06	0,0	7,26E-06	0,0
6,14E-06	0,0	7,45E-06	0,0
3,32E-04	114,7	3,46E-04	298,4
3,28E-04	91,2	3,99E-04	208,1
3,33E-04	85,3	3,74E-04	171,3
3,34E-04	73,2	3,99E-04	213,9
3,37E-04	121,9	3,46E-04	199,7
3,57E-04	65,9	4,00E-04	153,5
3,56E-04	47,7	3,70E-04	107,1
3,51E-04	47,6	4,11E-04	156,5
3,62E-04	57,9	3,68E-04	192,6
3,60E-04	88,0	3,70E-04	168,5
2,75E-03	229,6	2,82E-03	382,4
2,76E-03	275,0	2,95E-03	399,7
2,83E-03	316,5	2,90E-03	451,3
2,75E-03	326,7	2,94E-03	538,8
2,74E-03	288,8	2,79E-03	526,6
2,71E-03	109,4	2,83E-03	266,6
2,76E-03	142,7	2,85E-03	334,9
2,76E-03	154,9	2,99E-03	353,0
2,73E-03	259,3	2,82E-03	356,3
2,79E-03	84,2	3,16E-03	373,1
2,07E-02	655,8	2,17E-02	972,7
2,04E-02	731,8	2,15E-02	1064,3
2,07E-02	702,0	2,10E-02	1310,1
2,08E-02	747,2	2,20E-02	1100,5
2,06E-02	576,0	2,18E-02	798,1
4,90E-03	55,7	5,06E-03	141,4
4,84E-03	134,1	4,95E-03	473,7
4,91E-03	197,4	5,16E-03	427,3
4,87E-03	247,8	4,93E-03	335,2
4,89E-03	158,7	4,94E-03	273,9
3,90E-02	901,0	4,30E-02	1294,5
3,85E-02	1463,5	3,97E-02	1732,8
3,84E-02	931,6	3,93E-02	1447,0
4,00E-02	955,9	4,07E-02	1438,1
3,95E-02	828,7	4,02E-02	1070,7
8,94E-03	345,2	9,49E-03	452,4
9,04E-03	329,7	9,42E-03	434,8
9,01E-03	189,9	9,88E-03	469,2
8,89E-03	317,9	9,01E-03	488,9
9,02E-03	321,7	9,53E-03	497,0
6,87E-02	977,1	7,20E-02	1396,1
6,72E-02	991,9	6,89E-02	1337,8
6,68E-02	1187,9	6,84E-02	1397,1
6,70E-02	920,6	6,88E-02	1575,1
6,73E-02	1500,9	7,01E-02	1829,5

Algoritmo BL			
Mejores		Peores	
Tiempo	Disp	Tiempo	Disp
1,63E-04	0,0	8,95E-04	0,0
2,05E-04	0,0	2,19E-04	0,0
2,07E-04	0,0	2,18E-04	0,0
2,09E-04	0,0	2,63E-04	0,0
2,12E-04	0,0	2,18E-04	0,0
1,77E-03	40,8	4,14E-03	54,4
1,41E-03	20,8	3,72E-03	50,4
1,81E-03	40,0	2,36E-03	50,8
1,22E-03	35,9	4,79E-03	70,8
1,41E-03	35,1	3,24E-03	76,4
1,96E-03	12,1	3,67E-03	17,7
1,88E-03	12,7	5,10E-03	32,9
2,02E-03	10,7	3,32E-03	16,2
1,30E-03	14,7	3,39E-03	29,7
1,59E-03	13,8	3,80E-03	26,8
1,05E-02	110,1	3,53E-02	191,4
1,07E-02	93,1	5,17E-02	152,7
9,72E-03	79,2	4,40E-02	139,1
1,58E-02	91,7	3,71E-02	157,0
2,10E-02	52,3	5,01E-02	103,2
1,23E-02	33,2	4,39E-02	68,5
2,02E-02	45,5	3,19E-02	65,3
1,23E-02	31,6	4,15E-02	70,2
1,17E-02	33,7	3,99E-02	42,2
1,07E-02	31,7	3,77E-02	63,5
1,87E-01	240,7	9,01E-01	354,8
1,29E-01	280,9	4,86E-01	351,2
6,72E-02	346,8	6,27E-01	560,0
1,29E-01	285,7	3,90E-01	351,4
2,09E-01	260,0	4,29E-01	421,3
2,31E-02	37,4	1,59E-01	62,6
2,33E-02	47,3	5,95E-02	60,7
2,40E-02	49,2	1,06E-01	92,0
5,02E-02	46,3	1,03E-01	109,3
2,86E-02	53,0	9,05E-02	79,7
2,69E-01	260,2	1,59E+00	373,2
1,79E-01	393,9	5,70E-01	472,8
2,01E-01	332,4	2,23E+00	393,9
1,79E-01	281,7	1,24E+00	437,7
1,94E-01	260,9	9,09E-01	514,6
3,45E-02	48,5	1,13E-01	135,1
8,57E-02	59,4	1,56E-01	86,2
3,92E-02	58,5	1,60E-01	101,9
4,97E-02	53,9	2,45E-01	99,4
6,15E-02	68,5	1,68E-01	149,6
7,65E-01	431,3	2,59E+00	634,2
6,11E-01	438,3	4,40E+00	515,3
8,71E-01	441,2	3,24E+00	533,7
7,12E-01	387,9	1,43E+00	632,7
2,35E-01	322,4	2,35E+00	462,4

Figura 6.5: Mejores y peores valores obtenidos por algoritmo