Práctica 1.a

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

Alejandro Palencia Blanco

DNI: 77177568X

alepalenc@correo.ugr.es

Grupo y horario de prácticas: Jueves 17:30-19:30

Índice

1. Formulación del problema

El **Problema de la Máxima Diversidad** (*Maximum Diversity Problem*, MDP) es un problema de optimización combinatoria que pertenece a la clase de complejidad NP-completo, luego su resolución es compleja. Partiendo de un conjunto inicial S de n elementos, el problema general consiste en seleccionar un subconjunto Sel de m elementos que maximice la diversidad entre los elementos escogidos. La diversidad se calcula a partir de las distancias entre los elementos, las cuales se almacenan en una matriz $D = (d_{ij})$ de dimensión $n \times n$.

Existen distintas variantes del problema que dependen de la forma en que se calcula la diversidad:

- *MaxSum*: La diversidad se calcula como la suma de las distancias entre cada par de elementos seleccionados
- *MaxMin*: La diversidad se calcula como la distancia mínima entre los pares de elementos seleccionados
- *Max Mean Model*: La diversidad se calcula como el promedio de las distancias entre los pares de elementos seleccionados
- *Generalized Max Mean*: Existen pesos asociados a los elementos empleados en el denominador al calcular el promedio de distancias (hay un orden de importancia de los elementos)

En esta práctica, trabajaremos con el criterio *MaxSum*, luego el problema se puede formular de la siguiente forma:

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

$$\text{Sujeto a } \sum_{i=1}^{n} x_i = m$$

$$x_i \in \{0, 1\} \; \forall i \in \{1, \ldots, n\}$$

Donde x es un vector binario con n componentes que indica los m elementos seleccionados y d_{ij} es la distancia entre los elementos i y j.

2. Aplicación de los algoritmos empleados al problema

Los algoritmos que emplearemos en esta práctica comparten una serie de características comunes que serán descritas en esta sección. Concretamente, aquí explicaremos tanto el esquema de representación de las soluciones como la función objetivo que vamos a usar en cada uno de ellos. Como el algoritmo Greedy no hace uso del operador de vecino de intercambio del algoritmo de Búsqueda Local, este será explicado en la siguiente sección.

El lenguaje utilizado para la implementación de la práctica ha sido `C++` y he usado las siguientes bibliotecas:

- `<iostream>`
- `<ctime>` para medir tiempos
- `<cstdlib>` para las funciones `rand` y `srand`
- `<stl>` para las estructuras de datos `vector`, `set` y `unordered_set`
- `<utility>` para la estructura de datos `pair`
- `<algorithm>` para la función `random_shuffle`

2.1. Esquema de representación de soluciones

La formulación anterior permite dar una defición matemática del problema de forma sencilla, pero en la práctica es poco eficiente. Por tanto, resolveremos el problema mediante otro problema equivalente de programación lineal entera en el que los elementos de S se representan mediante números enteros del 0 al $n-1$. Por tanto, una solución consistirá en un subconjunto $Sel \subset S = \{0, \ldots, n-1\}$ tal que $|Sel| = m$.

Por cuestiones de eficiencia que serán comentadas más adelante, en la Búsqueda Local utilizo una estructura de datos que almacena, junto con cada elemento de una solución, la suma de las distancias al resto de elementos de esa solución (lo que podríamos llamar su contribución). Además, esta estructura de datos ordena los elementos de la solución según sus contribuciones de menor a mayor. Este criterio de ordenamiento para una solución se utiliza durante la ejecución del algoritmo, pero no es relevante a la hora de devolver la solución encontrada.

2.2. Función objetivo

Para el cálculo de la función objetivo he desarrollado dos funciones, una para Greedy en la que los elementos no tienen calculada su contribución, y otra para Búsqueda Local en la que está contribución ya está calculada. Por tanto, en la variante para Greedy es necesario usar la matriz de distancias para este cálculo. Ambas variantes acaban devolviendo el mismo valor para la misma solución.

Algorithm 1: evaluateFitness (Greedy)

```
Data: solution : vector<unsigned>, matrix : vector<vector<double>>
Result: fitness : double
begin
    fitness ← 0
    foreach i, j ∈ solution, i ≠ j do
        fitness ← fitness + matrix[i][j]
    end
end
```

Algorithm 2: evaluateFitness (LocalSearch)

```
Data: solution : set<pair<unsigned,double>>
Result: fitness : double
begin
    fitness ← 0
    foreach elem ∈ solution do
        fitness ← fitness + elem.second
    end
    fitness ← fitness/2
end
```

3. Algoritmos

He implementado tres algoritmos, siendo dos de ellos Greedy y Búsqueda Local. El último algoritmo es un híbrido de estos dos anteriores que tiene como objetivo intentar mejorar sus resultados sin que el tiempo empleado aumente de forma excesiva.

3.1. Greedy

El algoritmo Greedy persigue la fórmula heurística de añadir secuencialmente el elemento no seleccionado que aporte mayor diversidad a los ya seleccionados hasta ese momento. Como ya hemos visto, podemos seguir distintos criterios para calcular la diversidad. Por ejemplo, el algoritmo Greedy de Glover hace uso del concepto de elemento más central de un conjunto, es decir, su baricentro. Pero para seguir este criterio necesitaríamos

los valores concretos de los elementos, de los cuales no disponemos, luego modificamos ligeramente el algoritmo para que el criterio que use solo dependa de las distancias entre los elementos.

El algoritmo empieza con el conjunto de elementos seleccionados, *sel*, vacío y el conjunto de elementos no seleccionados, *not_sel*, conteniendo a todos los elementos. El conjunto *sel* ha sido implementado mediante la estructura `vector<unsigned>`, pues solo necesito realizar inserciones mediante el método `push_back`. En cambio, *not_sel* es un `unordered_set`, pues el orden de sus elementos es irrelevante y busco que el borrado de elementos sea rápido, para lo cual uso el método `erase`.

Elegimos como primer elemento a añadir a aquel cuya suma de sus distancias al resto de elementos sea máxima. Llamamos a esto la distancia acumulada de un elemento al resto y la calculamos con la función `acumDist`.

Algorithm 3: acumDist

```
Data: matrix : vector<vector<double>>, element : unsigned
Result: acum_dist : double
begin
    acum_dist ← 0
    for i ∈ {0, ..., n − 1} do
        acum_dist ← acum_dist + matrix[element][i]
    end
end
```

Cuando ya hemos añadido este primer elemento a *sel*, además de eliminarlo de *not_sel*, el algoritmo entra en un bucle *while* del que no se sale hasta que *sel* tenga m elementos (el número de elementos necesario para ser considerado una solución). En cada iteración, calculamos la distancia de cada uno de los elementos no seleccionados al conjunto *sel*, que es el mínimo de las distancias del elemento no seleccionado a cada uno de los elementos de *sel*. Esta distancia la calculamos con la función `distToSel`. A continuación, añadimos a *sel* aquel elemento cuya distancia al mismo sea máxima y lo eliminamos de *not_sel*.

Algorithm 4: distToSel

```
Data: matrix : vector<vector<double>>, element : unsigned, vector<unsigned> sel
Result: min_dist : double
begin
    min_dist ← matrix[element][sel[0]]
    for i ∈ {1, ..., |sel| − 1} do
        dist ← matrix[element][sel[i]]
        if dist < min_dist then
            min_dist ← dist
        end
    end
end
```

Algorithm 5: greedy

```
Data: matrix : vector<vector<double>>, m : unsigned
Result: sel : vector<unsigned>
begin
    non_sel ← {0, 1, ..., n − 1}
    best_dist ← 0
    for elem ∈ not_sel do
        dist ← acumDist(matrix, elem)
        if dist > best_dist then
            best_dist ← dist
            best_elem ← elem
        end
    end
    sel ← sel ∪ {best_elem}
    not_sel ← not_sel − {best_elem}
    while |sel| < m do
        best_dist ← 0
        for elem ∈ not_sel do
            dist ← distToSel(matrix, elem, sel)
            if dist > best_dist then
                best_dist ← dist
                best_elem ← elem
            end
        end
        sel ← sel ∪ {best_elem}
        not_sel ← not_sel − {best_elem}
    end
end
```

3.2. Búsqueda local

En este algoritmo, partimos de una solución generada aleatoriamente y almacenada en Sel (es decir, m elementos no repetidos cuyo orden no es relevante). La búsqueda local la haremos con el operador de vecino de intercambio, que consiste en intercambiar un elemento de la solución actual por otro que no pertenece a ella. Como el número de elementos de una solución es m y el número de elementos que no pertenecen a ella es $n-m$, entonces el tamaño del entorno es $m(n-m)$ (que se corresponde con el número de soluciones diferentes a las que podemos llegar aplicando el operador desde una concreta).

Si llevásemos a cabo una búsqueda local del mejor, tendríamos que explorar en cada iteración todo el entorno de la solución actual para realizar el intercambio hacia el mejor vecino. Este procedimiento funciona, pero es muy lento incluso para casos no demasiado grandes como $n = 500$ y $m = 50$, en el que el entorno tendría tamaño $m(n-m) = 25000$, luego optamos por implementar una búsqueda local del primer mejor.

En cada iteración, el algoritmo genera soluciones vecinas mediante el operador de intercambio y, cuando llegue a una que mejora a la actual, se queda con ésta y avanza a la siguiente iteración. El algoritmo para cuando se ha explorado todo el vecindario y no se ha encontrado una solución que mejore a la actual. La exploración del vecindario no se hace de forma totalmente aleatoria, sino que se ordenan los elementos de una solución en orden creciente según su contribución a la solución actual. De esta forma, primero aplicamos intercambios con aquellos elementos cuya contribución a la solución actual es menor. El orden en el que se escoge el otro elemento involucrado en el intercambio (aquel que no pertenece a la solución actual) sí es aleatorio.

Para evitar tener que calcular en cada iteración las contribuciones de los elementos de la solución actual, en mi implementación utilizo una estructura de datos en la que guardo junto con cada elemento su contribución. Por tanto, cada vez que se aplica el operador de intercambio, es necesario actualizar todas las contribuciones restando la distancia al elemento que eliminamos de la solución y sumando la del que añadimos. De esta forma, solo hay que calcular desde cero la contribución del elemento añadido a la nueva solución actual. Esta estructura es `set<pair<unsigned,double>,contribution_comp>`, donde `contribution_comp` es un `struct` que contiene un operador que compara las contribuciones de dos elementos con el objetivo de insertarlos en orden creciente según sus contribuciones.

Algorithm 6: contribution_comp

Data: p1 : pair<unsigned,double>, p2 : pair<unsigned,double>
Result: is_p1_less_than_p2 : bool
begin
  if $p1.second \neq p2.second$ **then**
    $is\_p1\_less\_than\_p2 \leftarrow p1.second < p2.second$
  else
    $is\_p1\_less\_than\_p2 \leftarrow p1.first < p2.first$
  end
end

El conjunto *not_sel* es de nuevo un `unordered_set<unsigned>` para garantizar que la forma de recorrer sus elementos mediante iteradores es aleatoria y que las inserciones y los borrados son rápidos.

El primer paso que realiza la búsqueda local es generar una solución aleatoria. Para ello, he implementado la función `generateRandomSolution`, que además de devolver en *sel* una solución aleatoria, devuelve en *not_sel* el resto de elementos no seleccionados. Primero, se introduce el conjunto $\{0, \ldots, n-1\}$ en un vector y lo barajamos con `random_shuffle`. A continuación, extraemos de éste m elementos aleatoriamente haciendo uso de `rand`, calculamos sus respectivas contribuciones una vez que todos ellos han sido seleccionados y los insertamos en *sel*. Por último, insertamos el resto de elementos en *not_sel*.

Algorithm 7: generateRandomSolution

Data: matrix : vector<vector<double>>, m : unsigned
Result: sel : set<pair<unsigned,double>,contribution_comp>, not_sel : unordered_set<unsigned>
begin
  $sel \leftarrow \emptyset$
  $not\_sel \leftarrow \emptyset$
  $v \leftarrow \{0, \ldots, n-1\}$
  $random\_shuffle(v)$
  for $i \in \{0, \ldots, m-1\}$ **do**
    $random\_element \leftarrow i + (rand()\,\%(|v| - i))$
    $aux \leftarrow v[i]$
    $v[i] \leftarrow v[random\_element]$
    $v[random\_element] \leftarrow aux$
  end
  $v\_contribs \leftarrow \{0, \overset{(m)}{\ldots}, 0\}$
  for $i \in \{0, \ldots, m-1\}$ **do**
    for $j \in \{i+1, \ldots, m-1\}$ **do**
      $v\_contribs[i] \leftarrow matrix[v[i]][v[j]]$
      $v\_contribs[j] \leftarrow matrix[v[i]][v[j]]$
    end
    $sel \leftarrow sel \cup \{make\_pair(v[i], v\_contribs[i])\}$
  end
  for $i \in \{m, \ldots, n-1\}$ **do**
    $not\_sel \leftarrow not\_sel \cup \{v[i]\}$
  end
end

Con la solución aleatoria ya generada, el algoritmo ya puede empezar la búsqueda local propiamente dicha. Para calcular la contribución que tendría un elemento no seleccionado en una solución en la que se pretende eliminar otro elemento (será ignorado en el cálculo de la contribución, pero todavía no ha sido eliminado de *sel*), he creado la función `calculateContribution`.

Cada vez que se encuentre un elemento no perteneciente a la solución actual cuya contribución devuelta por `calculateContribution` sea mayor a la del elemento ignorado, aplicaremos el operador de vecino de intercambio para mejorar nuestra solución actual. Este operador está implementado en la función `exchangeNeighbour`.

Algorithm 8: calculateContribution

```
Data: element : unsigned, element_to_ignore : unsigned,
      sel : set<pair<unsigned,double>,contribution_comp>,
      matrix : vector<vector<double>>
Result: contribution : double
begin
    contribution ← −matrix[element][element_to_ignore]
    for i ∈ sel do
        contribution ← contribution + matrix[element][i]
    end
end
```

Algorithm 9: exchangeNeighbour

```
Data: sel : set<pair<unsigned,double>,contribution_comp>,
      not_sel : unordered_set<unsigned>,
      matrix : vector<vector<double>>,
      old_element : pair<unsigned, double>,
      new_element : unsigned,
      new_element_contrib : double
Result: sel : set<pair<unsigned,double>,contribution_comp>,
      not_sel : unordered_set<unsigned>
begin
    sel ← sel − {old_element}
    not_sel ← not_sel ∪ {old_element}
    sel_aux ← ∅
    for i ∈ sel do
        updated_contrib ← i.second + matrix[i.first][new_element] − matrix[i.first][old_element]
        sel_aux ← sel_aux ∪ {make_pair(i.first, updated_contrib)}
    end
    sel ← sel_aux ∪ {make_pair(new_element, new_element_contrib)}
    not_sel ← not_sel − {new_element}
end
```

Algorithm 10: localSearch

```
Data: matrix : vector<vector<double>>, m : unsigned
Result: sel : set<pair<unsigned,double>,contribution_comp>
begin
    sel, not_sel ← generateRandomSolution(matrix, m)
    do
        exchange ← false
        for i ∈ sel and !exchange do
            for j ∈ not_sel and !exchange do
                new_element_contrib ← calculateContribution(j, i.first, sel, matrix)
                if new_element_contrib > i.second then
                    sel, not_sel ← exchangeNeighbour(sel, not_sel, matrix, i, j, new_element_contrib)
                    exchange ← true
                end
            end
        end
    while exchange;
end
```

3.3. Híbrido

Este algoritmo es una hibridación de los dos algoritmos anteriores que pretende quedarse con las bondades de cada uno de ellos. Por un lado, Greedy es un algoritmo rápido que permite generar soluciones relativamente buenas en poco tiempo. Por otro, Búsqueda Local es capaz de explorar el espacio de búsqueda de una forma más exhaustiva hasta llegar a un máximo local. Si en lugar de generar una solución aleatoria al inicio de la Búsqueda Local, esta solución fuese construida mediante Greedy, entonces la exploración del espacio de soluciones comenzaría desde una solución de una calidad razonable. Si esta solución inicial ya está cerca de un máximo local razonablemente bueno, esto puede llegar a ser beneficioso tanto para el tiempo como para la calidad de la solución final.

En pseudocódigo, el algoritmo Híbrido es similar a Búsqueda Local cambiando la función `generateRandomSolution` por `greedy`.

4. Procedimiento de desarrollo de la práctica

Cada algoritmo ha sido implementado en un fichero independiente que incluye todas las funciones que necesita. Los códigos fuentes se encuentran en el directorio `src`. Las 30 instancias proporcionadas para este problema se encuentran en el directorio `input`, (por motivos de espacio, en la entrega de la práctica este directorio está inicialmente vacío, pero si se quiere ejecutar los algoritmos sobre dichas instancias solo hay que introducirlas en él). Además, el proyecto también dispone de un directorio `bin` para los binarios y otro llamado `output` para guardar los resultados devueltos por los algoritmos. Todos estos directorios se encuentran dentro del directorio `software`.

Para automatizar todo el proceso, he creado un script en `bash` llamado `executeAlgorithms.sh` y un archivo `makefile`. El primero ejecuta los tres algoritmos con cada una de las 30 instancias del problema y guarda los resultados obtenidos por cada algoritmo en un fichero `.dat` dentro del directorio `output`. Por otro lado, el archivo `makefile` dispone de los siguientes comandos:

- `bin/<algoritmo>`: compila el código fuente con el algoritmo especificado (greedy, localSearch, hybrid)
- `example<algoritmo>`: compila el algoritmo especificado y lo ejecuta con tres instancias del problema con distintos tamaños, mostrando los resultados por la terminal
- `compile_all`: compila los tres algoritmos
- `examples`: compila los tres algoritmos y realiza sus ejemplos de ejecución
- `all`: compila los tres algoritmos y ejecuta el script `executeAlgorithms.sh`
- `clean`: elimina el contenido de los directorios `bin` y `output`

El ordenador en el que se han realizado los experimentos tiene 13.7 GiB de memoria RAM y un procesador AMD Ryzen 7 3700u. El sistema operativo que tiene instalado es Ubuntu 20.04.2.

5. Experimentos y análisis de resultados

5.1. Experimentos

Para evaluar el rendimiento de cada algoritmo y compararlos entre sí, los ejecutamos sobre los 30 casos proporcionados. Estos casos se pueden clasificar en 3 grupos en función del comienzo del nombre del fichero que los contiene. Cada uno de estos grupos está formado por 10 casos y tienen las siguientes características:

- MDG-a: $n = 500$, $m = 50$, distancias racionales
- MDG-b: $n = 2000$, $m = 200$, distancias racionales

- MDG-c: $n = 3000$, $m \in \{300, 400, 500, 600\}$, distancias enteras

Como podemos ver, con cada letra los valores de n y m aumentan, luego también es esperable que aumente el tiempo de ejecución.

En cuanto a las semillas utilizadas, he utilizado la sentencia `srand(1)` para fijar la semilla tanto en Búsqueda Local como en Híbrido. En Greedy no es necesario fijar ninguna semilla, ya que es determinista y siempre vamos a obtener la misma solución para un caso concreto.

5.2. Resultados

Para cada caso, los algoritmos devuelven el coste de la solución final y el tiempo que han empleado en calcularla, medido en segundos. Con estos resultados, he elaborado tres tablas. En la tabla 1, podemos ver los costes obtenidos junto con el óptimo asociado a cada caso. A continuación, tenemos la tabla 2 con los tiempos obtenidos. Por último, he calculado para cada algoritmo dos estadísticos a partir de las tablas anteriores. Por un lado está la media de las desviaciones y por otro la media de los tiempos. Definimos la desviación mediante el siguiente cociente y multiplicado por 100 para verlo en forma de porcentaje:

$$100 \cdot \frac{coste_{algoritmo} - coste_{optimo}}{coste_{algoritmo}}$$

Estos estadísticos se encuentran en las tablas 3 y 4, respectivamente, y han sido calculados tanto para el conjunto total de caso como para cada uno de los tres subconjuntos anteriormente mencionados.

Caso	Greedy	Búsqueda Local	Híbrido	Óptimo
MDG-a_1_n500_m50	6865.94	7573.54	7675.99	7833.83252
MDG-a_2_n500_m50	6769.68	7581.39	7666.75	7771.66162
MDG-a_3_n500_m50	6741.60	7606.58	7673.53	7759.35986
MDG-a_4_n500_m50	6841.59	7560.53	7511.62	7770.2417
MDG-a_5_n500_m50	6740.34	7618.90	7587.41	7755.23096
MDG-a_6_n500_m50	6952.95	7594.95	7548.83	7773.70996
MDG-a_7_n500_m50	6637.46	7617.86	7610.91	7771.73096
MDG-a_8_n500_m50	6946.28	7683.66	7518.42	7750.88135
MDG-a_9_n500_m50	6925.26	7545.83	7670.74	7770.0708
MDG-a_10_n500_m50	6848.57	7689.11	7618.71	7780.35059
MDG-b_21_n2000_m200	10314568.353374	11177930.880744	11185597.100425	11299894.85774
MDG-b_22_n2000_m200	10283328.499965	11200340.821013	11208399.011322	11286775.58894
MDG-b_23_n2000_m200	10224214.160396	11197381.663520	11192499.907093	11299940.86847
MDG-b_24_n2000_m200	10263575.472059	11196394.022819	11196566.502954	11290874.15963
MDG-b_25_n2000_m200	10250090.790876	11192378.358757	11169645.213420	11296066.6607
MDG-b_26_n2000_m200	10196189.880494	11174081.565327	11175165.277229	11292295.88629
MDG-b_27_n2000_m200	10358195.606871	11203562.071010	11187397.894291	11305676.84757
MDG-b_28_n2000_m200	10277383.165226	11179878.720191	11197682.128372	11279916.05595
MDG-b_29_n2000_m200	10291258.670388	11170642.525669	11213117.471060	11297188.33456
MDG-b_30_n2000_m200	10263859.329650	11214788.480507	11169241.897007	11296414.92982
MDG-c_1_n3000_m300	23016526	24731400	24761452	24884110
MDG-c_2_n3000_m300	23003023	24698216	24707185	24905330
MDG-c_8_n3000_m400	40498138	43212902	43248688	43437261
MDG-c_9_n3000_m400	40416360	43256869	43168926	43437861
MDG-c_10_n3000_m400	40641371	43263928	43259095	43476251
MDG-c_13_n3000_m500	62891574	66784629	66758671	67014051
MDG-c_14_n3000_m500	63055195	66759987	66795495	66979606
MDG-c_15_n3000_m500	63044103	66704810	66785908	66992877
MDG-c_19_n3000_m600	90718810	95289727	95317567	95633549
MDG-c_20_n3000_m600	90728662	95327159	95258256	95643586

Cuadro 1: Costes

Caso	Greedy	Búsqueda Local	Híbrido
MDG-a_1_n500_m50	0.000939	0.008005	0.008678
MDG-a_2_n500_m50	0.000858	0.008371	0.008945
MDG-a_3_n500_m50	0.000860	0.009346	0.008592
MDG-a_4_n500_m50	0.000901	0.008850	0.009982
MDG-a_5_n500_m50	0.000933	0.009367	0.008323
MDG-a_6_n500_m50	0.000924	0.006883	0.006677
MDG-a_7_n500_m50	0.000842	0.010098	0.009200
MDG-a_8_n500_m50	0.001578	0.009809	0.010152
MDG-a_9_n500_m50	0.000865	0.007454	0.008199
MDG-a_10_n500_m50	0.001350	0.009011	0.007449
MDG-b_21_n2000_m200	0.197770	2.151856	2.246905
MDG-b_22_n2000_m200	0.198127	2.233203	2.200428
MDG-b_23_n2000_m200	0.196726	2.127762	2.121615
MDG-b_24_n2000_m200	0.197402	2.178484	2.221656
MDG-b_25_n2000_m200	0.202940	1.984376	2.072369
MDG-b_26_n2000_m200	0.198144	2.114353	2.085053
MDG-b_27_n2000_m200	0.201107	2.207967	2.240059
MDG-b_28_n2000_m200	0.200148	2.181586	2.167887
MDG-b_29_n2000_m200	0.199529	2.162907	2.270365
MDG-b_30_n2000_m200	0.198674	2.275851	2.111233
MDG-c_1_n3000_m300	0.671329	8.020046	7.978911
MDG-c_2_n3000_m300	0.667153	7.695218	8.181308
MDG-c_8_n3000_m400	1.114676	13.341589	14.326703
MDG-c_9_n3000_m400	1.101780	14.025788	13.900844
MDG-c_10_n3000_m400	1.104032	14.182871	14.095741
MDG-c_13_n3000_m500	1.612960	20.265749	20.567244
MDG-c_14_n3000_m500	1.591374	19.659483	20.039913
MDG-c_15_n3000_m500	1.610086	19.566369	20.441362
MDG-c_19_n3000_m600	2.165338	26.940439	28.079696
MDG-c_20_n3000_m600	2.134681	26.847583	28.957537

Cuadro 2: Tiempos (seg)

	Greedy	Búsqueda Local	Híbrido
Media Desv MDG-a	12.18	2.14	2.13
Media Desv MDG-b	9.05	0.92	0.93
Media Desv MDG-c	6.36	0.47	0.45
Media Desv	9.20	1.17	1.17

Cuadro 3: Medias Desv

	Greedy	Búsqueda Local	Híbrido
Media Tiempos MDG-a	0.001005	0.008719	0.008620
Media Tiempos MDG-b	0.385110	4.416238	4.565554
Media Tiempos MDG-c	1.617179	20.212612	20.868905
Media Tiempos	0.525801	6.408356	6.613101

Cuadro 4: Medias Tiempos

5.3. Análisis

5.3.1. Análisis por costes

Si observamos la tabla 1 de costes, vemos que los resultados que obtiene Greedy siempre son inferiores a los obtenidos por Búsqueda Local e Híbrido para cada caso. Esto era esperable, pues Greedy construye una solución añadiendo sucesivamente el elemento que parece más prometedor según su contribución al resto de elementos ya seleccionados, lo cual no garantiza que la solución obtenida sea realmente buena. La información aportada por las medias de las desviaciones de la tabla 3 también dan respaldo a esto último. Vemos que la desviación obtenida en Greedy es casi ocho veces mayor que las desviaciones de Búsqueda Local e Híbrido, lo que significa que explorar localmente el espacio de soluciones es un procedimiento de búsqueda mucho más exhaustivo que construir una solución siguiendo una filosofía voraz.

Analizando las desviaciones por subconjuntos de casos, vemos que en el subconjunto de casos con un menor número de elementos, MDG-a, las desviaciones son mayores y que, a medida que aumenta el tamaño de los mismos, la desviación disminuye. Por ejemplo, Greedy consigue disminuir su desviación a la mitad, pasando de 12.18 para los casos de MDG-a a 6.36 para MDG-c. En los otros dos algoritmos, esta disminución es incluso mayor, pues consiguen reducir su desviación a más de la cuarta parte al pasar de MDG-a a MDG-c.

Si comparamos los resultados obtenidos por Búsqueda Local e Híbrido, vemos que son muy similares en casi todos los casos; en algunos Búsqueda Local supera a Híbrido y en otros pasa lo contrario. Además, sus medias de desviaciones son prácticamente idénticas. Estos resultados no deberían de sorprendernos, pues estos algoritmos solo se diferencian en la solución inicial de la que parten. En unos casos, Greedy es capaz de darnos una solución inicial que nos lleve a un mejor máximo local tras aplicar Búsqueda Local, pero en otros, es posible que la solución inicial aleatoria sea más prometedora.

5.3.2. Análisis por tiempos

En la tabla 2 de tiempos, podemos ver que Greedy obtiene en todos los casos unos tiempos mucho menores a los obtenidos por Búsqueda Local e Híbrido. Esto es debido a que Greedy no realiza una exploración sobre el espacio de soluciones. Búsqueda Local e Híbrido sí realizan esta exploración, la cual consume una mayor cantidad de tiempo. Si analizamos las medias en tiempos de la tabla 4, vemos que las obtenidas por Greedy tanto por subconjuntos como en total también son muy inferiores a las de Búsqueda Local e Híbrido. Concretamente, su media total es doce veces menor a la de los otros dos algoritmos. Todo esto demuestra que Greedy es el algoritmo que obtiene los mejores tiempos.

Como cabía esperar, todos los algoritmos sufren un aumento considerable en sus tiempos de ejecución a medida que aumenta el tamaño de los casos. Para MDG-a, la media de tiempos para todos ellos no supera la centésima de segundo, mientras que para MDG-c, las medias ya son superiores a un segundo en Greedy y superiores a 20 segundos en Búsqueda Local e Híbrido.

Si comparamos de nuevo, esta vez por tiempos, a Búsqueda Local e Híbrido, comprobamos que también son muy similares en la mayoría de los casos. Excluyendo la media del subconjunto MDG-a, las medias de Búsqueda Local son ligeramente inferiores a las de Híbrido. Esto nos da otro argumento que refuerza la postura de que, partir de una solución inicial Greedy, no aporta beneficios significativos a la Búsqueda Local con respecto a una generada aleatoriamente.

5.3.3. Conclusiones

Podemos concluir que, al menos para este problema, un algoritmo de Búsqueda Local proporciona resultados bastante mejores que uno Greedy. Sin embargo, Greedy garantiza tiempos mucho más bajos que Búsqueda Local.

Separando los casos por tamaño, hemos visto que para aquellos con mayor tamaño se obtienen resultados ligeramente superiores, aunque esto también supone un incremento considerable en el tiempo de ejecución.

Por último, el análisis anterior refleja que comenzar la Búsqueda Local desde una solución Greedy no aporta mejoras significativas con respecto a una generada aleatoriamente. Incluso puede llegar a ser perjudicial, pues

se ha visto que se obtienen tiempos ligeramente superiores para alguons casos.