Práctica 1.b: Técnicas de Búsqueda Local y Algoritmos Greedy para el Problema del Agrupamiento con Restricciones

Paula Villanueva Núñez

49314567Z

pvillanunez@correo.ugr.es

Tercer Curso del Grado en Ingeniería Informática Curso 2020-2021 Grupo 1 Jueves 17:30-19:30

Universidad de Granada

Índice

1 Descripción o formulación del problema

Esta práctica consiste en encontrar una solución al **Problema del Agrupamiento con Restricciones (PAR)**, una generalización del problema del agrupamiento clásico incorporando un nuevo tipo de información (las restricciones).

Este problema se basa en agrupar un conjunto de datos dado de tal forma que se cumplan las máximas restricciones posibles y se minimice la distancia media entre las instancias del mismo grupo.

Para ello, usaremos el término **cluster** para referirnos a cada grupo. Cada cluster tiene un **centroide**, que es el centro geométrico de todos los datos que conforman el cluster. Cada instancia solo puede pertenecer a un cluster y su distancia al centroide de su cluster debe ser menor que al resto de centroides.

Además, distinguiremos dos tipos de restricciones: las restricciones fuertes y las restricciones débiles.

Las **restricciones fuertes** deben satisfacerse en la partición del conjunto de datos. Estas son:

1. Todos los clusters deben contener al menos una instancia.
2. Cada instancia debe pertenecer a un solo cluster.
3. La unión de los clusters debe ser el conjunto de datos.

En cuanto a las **restricciones débiles**, nuestra solución tiene que incumplir el mínimo número de restricciones. Tenemos dos tipos:

- **Must-Link (ML).** Estas restricciones indican las parejas de instancias que deben pertenecer al mismo grupo.
- **Cannot-Link (CL).** Estas restricciones indican las parejas de instancias que deben pertenecer a distintos grupos.

El objetivo de este problema es minimizar la siguiente función

$$fitness(solucion) = distancia_{intra-cluster}(solucion) + \lambda \cdot infeasibility(solucion)$$

Donde

- $distancia_{intra-cluster}(solucion)$ es la media de las desviaciones intra-cluster de cada grupo. Cada desviación intra-cluster de cada grupo se calcula como la media de las instancias que lo forman a su centroide.
- $\lambda \in [\ 0, 1)$ es el cociente entre la distancia máxima que hay en el conjunto de datos y el número de restricciones presentes en el problema.
- $infeasibility(solucion)$ es el número de restricciones débiles que incumple nuestra solución.

2 Descripción de la aplicación de los algoritmos empleados al problema

El lenguaje de programación que he utilizado para resolver este problema ha sido C++.

2.1 Esquema de representación

He implementado la clase `PAR` que cuenta con los siguientes atributos:

- `vector< vector<int> > restricciones`
  Es una matriz simétrica de enteros que contiene las restricciones. Para dos posiciones distintas de datos i, j podemos acceder a esta matriz de tal forma que `restricciones[i][j]` devuelve un valor, que indica el tipo de restricción entre dichos datos: 0 (sin restricciones), 1 (restricción ML) o -1 (restricción CL).
- `vector< vector<int> > restriccionesML`
  Es una lista de enteros que contiene las restricciones Must-Link. Con `restriccionesML[i][0]` obtenemos una posición en `datos` tiene una restricción ML con la posición `restriccionesML[i][1]`, y viceversa, donde i varía entre 0 y el número máximo de restricciones ML.
- `vector< vector<int> > restriccionesCL`
  Es una lista de enteros que contiene las restricciones Cannot-Link. Con `restriccionesCL[i][0]` obtenemos una posición en `datos` que tiene una restricción CL con la posición `restriccionesCL[i][1]`, y viceversa, donde i varía entre 0 y el número máximo de restricciones CL.
- `vector< vector<double> > datos`
  Es una matriz simétrica de números reales que contiene las instancias de los datos en un espacio de d dimensiones.
- `vector<int> clusters`
  Es un vector de enteros que contiene los índices a los clusters de cada punto. Esto es, dada una posición i, obtenemos el cluster al que pertenece accediendo a `clusters[i]`. Los valores están comprendidos entre 0 y *número de clusters - 1*.
- `vector< vector<double> > centroides`
  Es una matriz de números reales que tiene los centroides de cada cluster. Podemos acceder al centroide del cluster i accediendo a `centroides[i]`, que nos devuelve un vector de dimensión d.
- `vector< vector<double> > distancias`
  Es una matriz simétrica de números reales que contiene las distancias entre los datos. Para dos posiciones distintas de datos i, j podemos acceder a la distancia entre estos con `distancias[i][j]`.
- `int num_clusters`
  Indica el número de clusters en los que agrupamos los datos.
- `double lambda`
  Parámetro que toma valores en $[0, 1)$ para asegurar que el factor `infeasibility` tiene la suficiente importancia.

La **solución** se representa con un vector de enteros que contiene los índices a los clusters de cada punto.

2.2 Descripción en pseudocódigo de la función objetivo y los operadores comunes

La **función objetivo** de esta práctica es

$$fitness(solucion) = distancia_{intra-cluster}(solucion) + \lambda \cdot infeasibility(solucion)$$

Nuestro problema debe devolver una solución que minimice esta función. Para calcular este valor, he realizado las siguientes operaciones.

```
fitness = desviación_general + lambda * infeasibility
```

Donde la **desviación general** nos indica cuánto de cerca están todos nuestros datos del centroide del cluster al que pertenecen.

```
Input: índices a los clusters {l_1, ..., l_{n-1}}, número de clusters
   num_clusters

suma = 0.0

for i ∈ {0, ..., num_clusters-1} do
  suma += distancia intra cluster del cluster i
end

return (suma/num_clusters)
```

La **distancia intra cluster** de cada cluster nos muestra, dado un cluster, cuánto de cerca están los datos de dicho cluster a su centroide.

```
Input: Conjunto de datos X de dimensión d, centroides {μ_1, ..., μ_k},
    cluster c_j, índices a los clusters {l_0, ..., l_{n-1}}

suma = 0.0
contador = 0  // número de instancias que pertenecen al cluster
    c_j

for i ∈ {0,...,n-1} do
  if (la asignación l_i pertenece al cluster c_j) do
    suma += distancia del dato x_i al centroide μ_j del cluster c_j
    contador++
  end
end

return (suma/contador)
```

Otro operador que necesitamos para calcular la función `fitness` es `lambda`. Este parámetro toma valores comprendidos entre 0 y 1, y es útil para darle suficiente relevancia al factor `infeasibility`.

```
lambda = distancia_máxima/número_total_restricciones
```

Finalmente, la `infeasibility` nos indica el número de restricciones que nuestra solución incumple. Nótese que la matriz de restricciones es simétrica, luego solo se recorre la parte triangular superior.

```
1   Input: Conjunto de restricciones R, índices a los clusters {
      l_0, ..., l_{n-1}}
2
3   infeasibility = 0
4
5   for i ∈ {0,...,n-1} do
6     for j ∈ {i+1,...,n-1} do
7       if (hay restricción ML pero l_i es distinto de l_j) do
8         infeasibility++
9       else if (hay restricción CL pero l_i es igual a l_j) do
10        infeasibility++
11      end
12    end
13  end
14
15  return infeasibility
```

3 Pseudocódigo de la estructura del método de búsqueda y operaciones relevantes de cada algoritmo

El método de búsqueda por trayectorias simples que he usado en esta práctica es la **Búsqueda Local (BL)**.

La búsqueda local consiste en, de forma aleatoria, generar una solución inicial y, al explorarla, generar posibles vecinos y comprobar si alguno puede minimizar la función objetivo actual. En caso afirmativo, esa será nuestra nueva solución.

Sin embargo, con este algoritmo obtenemos un mínimo local. Esto es, no nos garantiza alcanzar la solución óptima, aunque puede darse el caso de que el mínimo local coincida con la solución óptima.

3.1 Generación de soluciones aleatorias

El algoritmo BL comienza generando una solución aleatoria de tamaño n. Para ello, inicializo el atributo `clusters`, que es un vector de enteros con índices a los clusters asociados a cada instancia, con valores aleatorios entre 0 y $k-1$, donde k es el número de agrupamientos. Una vez inicializado, compruebo que no ha quedado ningún cluster sin instancia asignada. Si esto ha ocurrido, vuelvo a generar otra solución inicial. Finalmente, actualizo los centroides, pues hemos realizado cambios en los clusters.

```
1     Input: índices a los clusters {l_0, ..., l_{n-1}}
2
3     do
4       recalcular = false
5
6       for i ∈ {0,...,n-1} do
7         Asignar a l_i un número aleatorio entre 0 y k-1
8       end
9
10      // Comprobar que ningún cluster se ha quedado vacío
11      Inicializar el contador de clusters a 0
12
13      for i ∈ {0,...,n-1} do
14        Incrementar el contador del cluster al que l_i está asociado
15      end
16
17      for i ∈ {0,...,k-1} do
18        Comprobar si algún contador del cluster i es 0 (vacío)
19      end
20    while Haya clusters sin instancias asignadas
21
22    Actualizar los centroides
```

3.2 Operador de generación de vecino

El siguiente paso es recorrer la solución inicial calculada en el apartado anterior aleatoriamente y cambiar cada cluster por otros válidos, sus vecinos.

Para realizar este cambio, en cada iteración sobre los datos he creado un vector de pares `vecindarioVirtual` en el que el primer elemento del par indica el cluster al que pertenece el dato y el segundo elemento indica otro cluster válido por el que se puede cambiar.

```
1    Input: x_j dato en la posición j, c_j cluster asignado a x_j
2
3    for i ∈ {0,...,num_clusters-1} do
4      Si (c_j es distinto de i) do
5        Si (al asignarle el cluster i a x_j no se queda ningún
             cluster vacío) do
6          Añadir al vecindario virtual el par (c_j, i)
7        end
8      end
9    end
```

3.3 Descripción en pseudocódigo del método de exploración del entorno

Para explorar el entorno, tendremos que recorrer aleatoriamente la solución actual y en cada iteración generaremos un vecino. Comprobaremos si mejora la solución actual y de hacerlo, sera nuestra nueva solución.

```
1    fitnessActual = fitnessBL()
2    infeasibilityActual, infeasibilityNueva = infeasibilityBL()
3
4    do
5      indicesDatos <- RandomShuffle({0,...,n-1})
6
7      for i ∈ indicesDatos && no hay mejora && número de
           evaluaciones < 100000 do
8        Generar vecindario virtual del dato x_i
9        indicesVecindarios <- Barajar los índices al vecindario
10
11       for j ∈ indicesVecindarios && no hay mejora && número de
             evaluaciones < 100000 do
12         Asignar el cluster j al dato x_i
13         Decrementar infeasibilityNueva la infeasibility que
             produce el dato x_i asignado al cluster l_i
14         Incrementar infeasibilityNueva la infeasibility que
             produce el dato x_i asignado al cluster j
15         Calcular fitnessNueva con infeasibilityNueva
16         Incrementar el número de evaluaciones de la función
             fitness
17
```

```
18            Si (fitnessNueva es menor que fitnessActual) do
19              Actualizar fitnessActual e infeasibilityActual
20            else
21              Reestablecer los clusters y la infeasibilityNueva
22            end
23          end
24        end
25      while Hay mejora y número de evaluaciones < 100000
```

4 Pseudocódigo de los algoritmos de comparación

En esta práctica, he implementado el algoritmo **Greedy COPKM** para compararlo con el otro algoritmo de esta práctica (BL). Este algoritmo se basa en el algoritmo k-medias para agrupar un conjunto de datos con la diferencia de que tiene en cuenta las restricciones asociadas a dicho conjunto.

Al contrario que la Búsqueda Local, este algoritmo intenta minimizar el número de restricciones incumplidas aunque la desviación general sea mayor.

4.1 Generación de centroides aleatorios

Este algoritmo comienza generando unos centroides aleatorios con valores en el dominio de los datos.

```
1   Input: dimensión del conjunto de datos n, número de clusters k,
        centroides {μ_1, ..., μ_k}
2
3   Borrar el contenido de los centroides
4
5   for i ∈ {0,...,k-1} do
6     Asignar al centroide μ_i un vector de dimensión n con valores
        aleatorios entre 0 y 1
7   end
```

4.2 Pseudocódigo para actualizar los centroides

En este algoritmo es necesario que se actualicen los centroides cada vez que se produzca un cambio en los clusters.

```
1   Input: índices a los clusters {l_1, ..., l_n}
2
3   Comprobar que todos los elementos estén asociados a un cluster
4
5   // Calcular el número de elementos de cada cluster
6   for i ∈ {0,...,n-1] do
7     Incrementar el contador de cluster al que pertenece l_i
8   end
9
10  Inicializar los centroides a 0
11
12  // Promediar las instancias asociadas a cada cluster
13  for i ∈ {0,...,n-1} do
14    Sumar las coordenadas de los datos que pertenecen al cluster
        l_i en su centroide μ_{l_i}
15  end
```

```
16
17      for i ∈ {0,...,k-1} do
18        Asignar al centroide μ_i la división del centroide μ_i entre el
             número de elementos del cluster c_i
19      end
```

4.3 Pseudocódigo del algoritmo Greedy

Una vez inicializados los centroides, barajamos los datos para recorrerlos aleatoriamente y en cada iteración, asignamos cada dato al cluster en el que menos restricciones incumple y al que menor distancia de encuentra.

```
1       Input: conjunto de datos X
2
3       Inicializar centroides aleatoriamente
4
5       //Barajar los índices a los datos
6       RSI <- RandomShuffle({0,...,n-1})
7
8       do
9         for i ∈ RSI do
10          for j ∈ {0,...,k-1} do
11            Calcular el número de restricciones que incumple el dato
                 x_i en el cluster j
12
13            Si (el número de restricciones incumplidas es menor que
                 el actual) do
14              Guardar el número de restricciones incumplidas, el
                   cluster y la distancia actuales por los nuevos
                   valores
15            Si (el número de restricciones incumplidas es igual que
                 el actual) do
16              Si (la distancia del dato x_i al cluster j es menor que
                   la distancia actual) do
17                Guardar el cluster y la distancia actuales por los
                     nuevos valores
18              end
19            end
20          end
21
22          Si (el cluster asignado a x_i ha sido modificado) do
23            Actualizar el cluster asignado a x_i por el nuevo valor
24          end
25        end
26
27        Actualizar los centroides
```

```
28      while Haya cambios en algún cluster
29
30      // Comprobar que no ha quedado algún cluster vacío
31      Si (no hay algún cluster vacío) do
32        Actualizar la desviación general, fitness e infeasibility
33        Devolver la lista de clusters
34      Si no
35        Volver a ejecutar Greedy
36      end
```

5 Procedimiento considerado para desarrollar la práctica

5.1 Implementación a partir del código proporcionado en prácticas o a partir de cualquier otro

Para el desarrollo de mi práctica he necesitado generar números aleatorios, para ello he utilizado el código proporcionado por los profesores que se encuentra en los archivo `random.h` y `random.cpp`.

Además, para medir el tiempo he usado la librería `chrono`.

Para implementar la clase `PAR` he usado la STL de C++ (`vector` y `pair`), la librería `math`, etc.

5.2 Manual de usuario

5.2.1 Estructura de carpetas

La organización de esta práctica se ha dividido en varias carpetas.

- `BIN/`: carpeta que contiene el ejecutable.
- `BIN/DATA/`: carpeta que contiene el conjunto de datos y sus restricciones. Al ejecutar la práctica, creará los archivos con los resultados de cada algoritmo.
- `FUENTES/include/`: carpeta que contiene los archivos de cabecera.
- `FUENTES/src/`: carpeta que contiene los archivos fuente.
- `FUENTES/obj/`: carpeta que contiene los archivos objeto.

5.2.2 Compilación

Para compilar la práctica, he creado un fichero `Makefile`. Por lo que bastará con ejecutar el siguiente comando

```
1    make
```

Nos creará en la carpeta `BIN/` un ejecutable llamado `practica1`.

5.2.3 Ejecución

Para ejecutar la práctica, necesitamos pasarle al ejecutable los siguientes parámetros.

```
1    ./BIN/practica1 ficheroDatos.dat ficheroRestricciones.const
2    númeroDeClusters semilla
```

Por ejemplo, si queremos ejecutar la práctica con los datos de `zoo` con el 10% de restricciones con la semilla 22, tendremos que hacer lo siguiente.

```
1    ./BIN/practica1 zoo_set.dat zoo_set_const_10.const 7 22
```

En la terminal veremos qué algoritmos se están ejecutando y, una vez terminen, mostrarán un mensaje con la ruta donde se encuentra el fichero con los resultados.

6 Experimentos y análisis de resultados

6.1 Casos del problema empleados y valores de los parámetros considerados en las ejecuciones de cada algoritmo

6.1.1 Casos del problema empleados

Para realizar esta práctica, he considerado tres conjuntos de datos:

1. **Zoo:** contiene los datos de un conjunto de animales, cada uno con 16 atributos sobre sus características. El objetivo es clasificar 101 instancias de animales en 7 clases según sus atributos.
2. **Glass:** contiene los datos de un conjunto de vidrios, cada uno con 5 atributos sobre sus componentes químicos. El objetivo es clasificar 214 instancias de vidrios en 7 clases según sus atributos.
3. **Bupa:** contiene los datos de un conjunto de personas, cada una con 5 atributos sobre sus hábitos de consumo de alcohol. El objetivo es clasificar 345 instancias de personas en 16 clases según sus atributos.

Cada conjunto de datos tiene asociados dos conjuntos de restricciones, correspondientes al 10% y al 20% del total de restricciones posibles. Estas restricciones serán muy importantes a la hora de determinar una solución, pues indican qué conjuntos de datos son los que deben ir en la misma clase, aunque parezcan muy distintos, y los que deban ir a clases distintas, aunque parezcan ser similares. Los algoritmos tendrán en cuenta estas restricciones para agrupar las instancias.

En total, el PAR trabajará con 6 instancias generadas a partir de los datos anteriores.

6.1.2 Valores de los parámetros considerados

Para determinar si los algoritmos funcionan correctamente, necesitamos ejecutarlos de formas diferentes. Para ello, inicializamos una semilla y a partir de esta, los algoritmos tomarán secuencias distintas aleatorias y, por ello, resultados diferentes. Para realizar las ejecuciones, he elegido las siguientes cinco semillas de forma aleatoria: 7, 22, 100, 222, 273687. Por lo que para cada semilla, conjunto de datos y conjunto de restricciones, he realizado una ejecución. Así, he realizado 30 ejecuciones en total por algoritmo.

6.2 Resultados obtenidos

A continuación se muestran los resultados obtenidos con los distintos algoritmos y las medias.

Resultados obtenidos por el algoritmo Greedy en el PAR con 10% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	ErrorDist	Agr.	T	Infeasable	ErrorDist	Agr.	T	Infeasable	ErrorDist	Agr.	T
7	17	0.0541104	1.08773	0.006	9	0.00240336	0.37555	0.039	55	0.0165283	0.251691	
22	9	0.135563	1.10856	0.004	1	0.0164101	0.381684	0.034	37	0.0149684	0.245308	
100	0	0.0110949	0.915895	0.010	4	0.0548859	0.423112	0.029	89	0.0115209	0.255796	
222	1	0.00188341	0.910494	0.006	6	0.0128024	0.382997	0.057	16	0.00465167	0.229363	
273687	10	0.148098	1.12867	0.004	3	0.0148285	0.352414	0.030	11	0.0139084	0.23728	
Media	7.4	0.06815286	1.0302698	0.006	4.6	0.020266052	0.3831514	0.0378	41.6	0.012315534	0.2438876	

Resultados obtenidos por el algoritmo BL en el PAR con 10% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	ErrorDist	Agr.	T	Infeasable	ErrorDist	Agr.	T	Infeasable	ErrorDist	Agr.	T
7	14	0.293276	0.71761	0.389	18	0.121449	0.260554	1.094	140	0.114836	0.143106	8.943
22	18	0.376795	0.664402	0.522	39	0.154189	0.248479	1.061	81	0.0998916	0.142239	8.271
100	15	0.294719	0.723745	0.592	43	0.153585	0.253019	1.338	127	0.108606	0.145852	10.05
222	19	0.296847	0.751928	0.707	49	0.170574	0.241935	0.978	133	0.105082	0.150984	5.437
273687	9	0.29219	0.680808	0.503	21	0.124522	0.260433	1.195	107	0.108612	0.140486	10.07
Media	15	0.3107654	0.7076986	0.5426	34	0.1448638	0.252884	1.1332	117.6	0.10740552	0.1747302	8.557

Resultados globales en el PAR con 10% de restricciones												
Algoritmo	Zoo				Glass				Bupa			
	Infeasable	ErrorDist	Agr.	T	Infeasable	ErrorDist	Agr.	T	Infeasable	ErrorDist	Agr.	T
COPKM	7.4	0.06815286	1.0302698	0.006	4.6	0.0202660	0.3831514	0.0378	41.6	0.0123155	0.2438876	0
BL	15	0.3107654	0.7076986	0.5426	34	0.1448638	0.252884	1.1332	117.6	0.1074055	0.1747302	8

Resultados obtenidos por el algoritmo Greedy en el PAR con 20% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	ErrorDist	Agr.	T	Infeasable	ErrorDist	Agr.	T	Infeasable	ErrorDist	Agr.	T
7	0	2.22045e-16	0.9048	0.006	0	0.0148355	0.349455	0.025	2	0.0178596	0.238562	0.1
22	1	0.0295632	0.938401	0.006	0	0.052908	0.311382	0.020	2	0.0088525	0.229555	0.1
100	4	0.0355002	0.956452	0.008	1	0.00854759	0.356263	0.020	16	0.0176109	0.240266	0.2
222	3	0.0772552	0.994169	0.006	6	0.0199431	0.347471	0.035	2	0.0253277	0.24603	0.3
273687	1	0.0449131	0.953751	0.006	1	0.00035010	0.365161	0.020	8	0.0120281	0.233567	0.1
Media	1.8	0.0374463	0.9495146	0.0064	1.6	0.01931685	0.345946	0.024	6	0.01633576	0.2380166	0.20

Resultados obtenidos por el algoritmo BL en el PAR con 20% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	ErrorDist	Agr.	T	Infeasable	ErrorDist	Agr.	T	Infeasable	ErrorDist	Agr.	T
7	10	0.171019	0.774162	0.345	119	0.160275	0.265967	1.015	225	0.11033	0.141468	8.555
22	14	0.171457	0.789876	0.372	29	0.113331	0.266057	1.023	241	0.110878	0.143151	10.16
100	33	0.310457	0.727599	0.486	119	0.155487	0.270756	1.345	274	0.10985	0.148781	10.10
222	21	0.171234	0.818365	0.341	117	0.155015	0.270187	1.182	162	0.0996144	0.143399	9.958
273687	30	0.194177	0.831766	0.383	130	0.163206	0.268763	1.048	174	0.0941389	0.150548	7.427
Media	21.6	0.2036688	0.7883536	0.3854	102.8	0.1494628	0.268346	1.1226	215.2	0.10496226	0.1454694	9.241

Resultados globales en el PAR con 20% de restricciones												
Algoritmo	Zoo				Glass				Bupa			
	Infeasable	ErrorDist	Agr.	T	Infeasable	ErrorDist	Agr.	T	Infeasable	ErrorDist	Agr.	T
COPKM	1.8	0.0374463	0.9495146	0.0064	1.6	0.0193168	0.345946	0.024	6	0.01633576	0.2380166	0.2
BL	21.6	0.2036688	0.7883536	0.3854	102.8	0.1494628	0.268346	1.1226	215.2	0.10496226	0.1454694	9.2

6.3 Análisis de resultados

En los resultados obtenidos por el algoritmo Greedy, observamos que se incumplen menos **restricciones** que con el algoritmo BL. Lo mismo ocurre con la **desviación general** de ambos algoritmos. Esto se debe a que el algoritmo Greedy intenta minimizar estas dos medidas, en especial el número de restricciones incumplidas. Hay que destacar que el número de restricciones incumplidas por el algoritmo Greedy con el 10% de restricciones es mayor que con el 20% de restricciones, mientras que con el algoritmo BL se incrementa el número de restricciones incumplidas con el 20% de restricciones, con respecto al 10% de restricciones. En cuanto a la desviación general, este valor se mantiene parecido o incluso menor cuando pasamos de considerar 10% de restricciones a 20% de restricciones.

Sin embargo, el algoritmo BL muestra mejor resultados en el **agregado**, pues el objetivo de este algoritmo es optimizar la función que calcula este valor, teniendo en cuenta la distancia y el número de restricciones incumplidas, dándole a esta última medida la suficiente relevancia.

También hay que tener en cuenta que el algoritmo Greedy emplea mucho menos **tiempo** (medido en segundos) que el algoritmo BL, pues este último algoritmo tiene que barajar el entorno y evaluar la función objetivo repetidas veces. Con el conjunto de datos Zoo y Glass no se nota demasiado la diferencia, pero con Bupa si es considerable la cantidad de tiempo. Esto puede deberse a que este último conjunto de datos se agrupa en 16 clases, por lo que el algoritmo BL tendrá que generar más vecinos y comprobar si optimizan la función objetivo.

En resumen, tanto con el 10% y 20% de restricciones, que el algoritmo BL nos dé un valor de agrupamiento menor al del algoritmo Greedy nos indica que BL se comporta mejor en cuanto a la búsqueda de la solución. Sin embargo, también tenemos que tener en cuenta que BL obtiene agrupamientos menos respetuosos con las restricciones y mayor desviación hay con los datos, lo que genera una peor calidad. Con respecto al tiempo, en Zoo y Glass apenas se aprecia la diferencia, pero en Bupa sí es considerablemente peor por lo que la calidad de BL tiende a ser mucho peor que Greedy. En conclusión, el algoritmo Greedy es mejor que la búsqueda local.

7 Referencias bibliográficas u otro tipo de material consultado

- Material proporcionado por los profesores sobre la asignatura.
  https://sci2s.ugr.es/node/124
- Material consultado para medir tiempos.
  https://www.geeksforgeeks.org/measure-execution-time-function-cpp/