Práctica 3.b: Búsquedas por Trayectorias 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.

```
Input: Conjunto de restricciones R, índices a los clusters {l_0,...,l_{n-1}}

infeasibility = 0

for i ∈ {0,...,n-1} do
  for j ∈ {i+1,...,n-1} do
    if (hay restricción ML pero l_i es distinto de l_j) do
      infeasibility++
    else if (hay restricción CL pero l_i es igual a l_j) do
      infeasibility++
    end
  end
end

return infeasibility
```

2.2.1 Generación de soluciones aleatorias

Para generar una solución aleatoria, primero 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.

```
Input: índices a los clusters {l_0,...,l_{n-1}}

do
  recalcular = false

  for i ∈ {0,...,n-1} do
    Asignar a l_i un número aleatorio entre 0 y k-1
  end

  // Comprobar que ningún cluster se ha quedado vacío
  Inicializar el contador de clusters a 0

  for i ∈ {0,...,n-1} do
    Incrementar el contador del cluster al que l_i está asociado
  end

  for i ∈ {0,...,k-1} do
    Comprobar si algún contador del cluster i es 0 (vacío)
  end
while Haya clusters sin instancias asignadas

Actualizar los centroides
```

2.2.2 Comprobar que el cambio de cluster sea válido

Esto es, si tenemos un par (i, k), donde i es la posición de una instancia y k es el nuevo índice a un cluster. Dada una solución tal que la instancia i tenga asociado un cluster distinto al cluster k,

comprobamos si le asignamos ese cluster, se sigue cumpliendo que no quede ningún cluster vacío. Esto lo hacemos con la función `parValido (par, solucion)`

```
1     Input: vector solución solucion, par formado por la posición y el nuevo
          cluster par(i,k), número de clusters num_clusters
2
3     aux = solucion
4
5     aux[par.first] = par.second
6
7     // Comprobar que ningún cluster se ha quedado vacío
8     contador = vector de enteros de tamaño num_clusters // Inicializar el contador
          de clusters a 0
9
10    for i ∈ {0,...,n-1} do
11      Incrementar el contador del cluster al que aux[i] está asociado
12    end
13
14    for i ∈ {0,...,k-1} do
15      //Comprobar si algún contador del cluster i es 0 (vacío)
16      Si (contador[i] == 0)
17        return false
18      end
19    end
20
21    return true
```

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

3.1 Algoritmo de Búsqueda Local

El método de búsqueda por trayectorias simples que he usado en la práctica 1 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.

3.1.1 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
```

3.1.2 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
```

3.1.3 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 and no hay mejora and 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 and no hay mejora and 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
```

3.2 Algoritmo de Enfriamiento Simulado (ES)

3.2.1 Introducción

El algoritmo de Enfriamiento Simulado (ES) es un algoritmo de búsqueda por entornos con criterio probabilístico de aceptación de soluciones basado en la Termodinámica.

Es una modificación de la Búsqueda Local para evitar estancarse en óptimos locales. Para ello, permite realizar algunos movimientos que empeoren la solución actual. Esto es posible gracias a una función de probabilidad, la cual hará disminuir la probabilidad conforme avanza la búsqueda. Por ello, necesitaremos un nuevo parámetro, la temperatura. Este algoritmo simulará el enfriamiento de dicha temperatura y gracias a ella, podremos controlar cómo movernos hacia soluciones peores. Cuando la temperatura sea elevada, podremos explorar nuevas soluciones para salir de óptimos locales y conforme avance la búsqueda, la temperatura disminuirá y podremos explotar las soluciones.

3.2.2 Operador de vecino y exploración del entorno para L(T)

Este algoritmo, al igual que la Búsqueda Local, explora el entorno y evalúa si los vecinos ofrecen mejores resultados. Para ello, se elige una posición aleatoria del vector que contiene los índices a los clusters y le asigna otro distinto (aleatoriamente también), comprobando que se siguen cumpliendo todas las restricciones.

3.2.3 Condición de enfriamiento L(T)

En cada iteración para generar un nuevo vecino, tendremos una condición de enfriamiento L(T) que debe cumplirse. La condición de enfriamiento L(T) considerada en este algoritmo es que el número de vecinos generados no sobrepase un cierto límite y que el número de exitos tampoco supere otro límite.

- El máximo número de vecinos a generar será $10 \cdot n$, donde n es el tamaño del caso del problema.
- El máximo número de éxitos se calculará como $0.1 * numeroMaximoVecinos$.

3.2.4 Esquema de enfriamiento

Temperatura inicial Para calcular la temperatura inicial, hay que tener en cuenta ciertos parámetros. Tenemos que considerar el coste de la solución inicial $C(S_0)$ y phi la probabilidad de aceptar una solución un μ por 1 peor que la inicial. Tomaremos $\phi = \mu = 0.3$. Por lo que la temperatura inicial se podrá calcular como sigue

$$T_0 = \frac{\mu \cdot C(S_0)}{-ln(\phi)}$$

Temperatura final La temperatura final se fijará a 10^{-3} y deberemos comprobar que siempre sea menor que la temperatura inicial.

Esquemas En este algoritmo, se ha empleado el esquema de Cauchy modificado.

Con esto, podremos enfriar la temperatura actual de la siguiente forma.

$$T_{k+1} = \frac{T_k}{1 + \beta \cdot T_k} \qquad ; \qquad \beta = \frac{T_0 - T_f}{M \cdot T_0 \cdot T_f}$$

donde $M = 100000/numeroMaximoVecinos$ es el número de enfriamientos a realizar, T_0 es la temperatura inicial y T_f es la temperatura final.

3.2.5 Condición de parada

En el bucle principal tendremos varias condiciones que deben cumplirse.

Una condición es que el número de evaluaciones de la función objetivo no debe superar las 100000 evaluaciones. El número de éxitos no debe ser nulo y la temperatura actual debe ser superior a la temperatura final.

3.2.6 Implementación del algoritmo

```
1   Input: solución actual actualSol, número de clusters k, número máximo de
       evaluaciones de la función objetivo nFitnessMAX, φ PHI, μ MU,
2   temperatura final Tf
3
4   // Inicializar parámetros
5   n = actualSol.size()
6   nVecinosMAX = 10*n
7   nExitosMAX = 0.1*nVecinosMAX
8   nEnfriamientosMAX = nFitnessMAX/nVecinosMAX
9   mejorSol = actualSol
10  nFitness = 0    // número de evaluaciones de la función objetivo
11  nExitos = 1     // número de éxitos
12  fitActual = fitness (actualSol)
13  fitMejor = fitActual
14  T0 = (MU * fitActual)/(-ln(PHI))
15  beta = (T0 - Tf)/(nEnfriamientosMAX * T0 * Tf)
16  T = T0
17
18  Mientras (nFitness < nFitnessMAX and nExitos != 0)
19    nVecinos = 0
20    nExitos = 0
21
22    Mientras (nVecinos < nVecinosMAX and nExitos < nExitosMAX)
23      // Generar un vecino
24      nuevaSol = actualSol
25      pos = número aleatorio entero en [0, n)
26      nuevoCluster = número aleatorio entero en [0, k)
27      c_i = actualSol[pos]
28      par = (c_i, nuevoCluster)   // Par formado por el cluster actual y el nuevo
            cluster
29
30      // Comprobamos que no sean iguales y que el par sea válido (esto es,
31      que no deje ningún cluster vacío al realizar cambiar el cluster c_i por
           nuevoCluster)
32      Si (c_i != nuevoCluster and parValido (par, actualSol))
33        nuevaSol[pos] = nuevoCluster
34        fitNueva = fitness (nuevaSol)
35        Incrementar nFitnes
36        Incrementar nVecinos
37        diferenciaFitness = fitNueva - fitActual
38
```

```
39          Si (diferenciaFitness < 0 or Rand(0,1) <= exp(-diferenciaFitness/T))
40            actualSol = nuevaSol
41            fitActual = fitNueva
42            Incrementar nExitos
43
44            Si (fitActual < fitMejor)
45              mejorSol = actualSol
46              fitMejor = fitActual
47            end
48          end
49        end
50      end
51
52      T = esquemaEnfriamiento (T, beta)
53    end
54
55    return mejorSol
```

3.3 Algoritmo de Búsqueda Multiarranque Básica (BMB)

Este algoritmo se basa en generar un determinado número de soluciones aleatorias y aplicarle a cada una de ellas el algoritmo Búsqueda Local (BL) para optimizarlas. Devolverá la mejor solución que encuentre.

Los parámetros que se han utilizado para ejecutar el algoritmo son los siguientes. En cada ejecución se han realizado 10 iteraciones en total, luego se han generado 10 soluciones aleatorias y se les ha aplicado a cada una de ellas la Búsqueda Local. En cada una de estas aplicaciones, la Búsqueda Local termina cuando se hayan realizado 10000 evaluaciones o cuando no encuentre mejora en todo el entorno.

```
1   Input: número de iteraciones máximas nIterMAX, número de evaluaciones máximas
        nEvaluacionesMAX
2
3   mejorFit = +infinito
4
5   Para i ∈ {0,...,nIterMAX}
6     actualSol = generarSolucionAleatoria ()
7     nuevaSol = busquedaLocal (actualSol, nEvaluacionesMAX)
8     nuevaFit = fitness (nuevaSol)
9
10    Si (nuevaFit < mejorFit)
11      mejorFit = nuevaFit
12      mejorSol = nuevaSol
13    end
14  end
15
16  return mejorSol
```

3.4 Algoritmo de Búsqueda Local Reiterada (ILS)

Este algoritmo consiste en generar una solución aleatoria y aplicarle la Búsqueda Local. Tras esta aplicación, se realizan un determinado número de iteraciones. En cada una se le aplica una mutación

a la mejor solución obtenida hasta el momento (con el operador de mutación de segmento fijo) y después se le aplica a esta nueva solución la Búsqueda Local. Si esta última solución que se obtiene es mejor que la mejor solución actual, actualizamos la mejor solución a esta última y así pasamos a la siguiente iteración.

Con la Búsqueda Local obtenemos un mínimo y, al aplicarle una mutación, conseguimos cambiar bruscamente la solución y abandonar el óptimo local. Así, se cambia el espacio de búsqueda, aunque se mantiene cierta parte de la solución sin mutar.

3.4.1 Operador de mutación de segmento fijo

```
1   Input: índices a los clusters C, número de clusters que permanecen v, número
        de clusters k
2
3   n = C.size()
4   solución = vector de enteros de tamaño n inicializado a -1
5   r = número aleatorio en [0, n)  // Inicio del segmento
6
7   // Copiar los v elementos
8   Para i ∈ {0,...,v}
9     solucion[r] = C[r]
10    r = (r + 1) % n
11  end
12
13  nIter = n - v   // Iteraciones restantes
14
15  // Asignar clusters aleatorios al resto de índices
16  Para i ∈ {0,...,nIter}
17    solucion[r] = número aleatorio en [0, k)
18    r = (r + 1) % n
19  end
20
21  // Comprobar que ningún cluster se ha quedado vacío
22
23  contador = vector de enteros de tamaño k inicializado a 0
24
25  Para i ∈ {0,...,n}
26    Incrementar el contador del cluster al que C_i está asociado
27  end
28
29  Para i ∈ {0,...,contador.size()}
30    Si el contador del cluster i es 0 (vacío)
31      Hacer
32        j = número aleatorio entero en [0, n)
33      Mientras (contador[C[j]] - 1 <= 0)
34
35      C[j] = i
36      Decrementar contador[C[j]]
37      Incrementar contador[i]
38    end
39  end
40
41  return C
```

3.4.2 Implementación del algoritmo ILS

En cada ejecución se han utilizado unos determinados parámetros. El número máximo de evaluaciones de la función objetivo se establece en 10000 evaluaciones. Además, se realizarán 10 iteraciones. También se considera que el número de elementos que permanecerán en la mutación será del 10% de elementos.

```
1   Input: número de iteraciones máximas nIterMAX, número de evaluaciones máximas
        nEvaluacionesMAX, porcentaje de número
2   de elementos que permanecen en la mutación porcentaje
3
4   S0 = generarSolucionAleatoria ()
5   S = busquedaLocal (S0, nEvaluacionesMAX)
6   mejorSol = S
7   fitnessMejor = fitness (mejorSol)
8   n = mejorSol.size()
9   v = porcentaje*n/100
10
11  Para i ∈ {1,...,nIterMAX}
12    S1 = operadorMutacionSF (mejorSol, v)
13    S2 = busquedaLocal (S1, nEvaluacionesMAX)
14    fitnessS2 = fitness (S2)
15
16    Si (fitnessS2 < fitnessMejor)
17      mejorSol = S2
18      fitnessMejor = fitnessS2
19    end
20  end
21
22  return mejorSol
```

3.5 Algoritmo Híbrido ILS-ES

Este algoritmo es una hibridación del algoritmo ILS y del algoritmo ES. La diferencia con el algoritmo ILS es que, en vez de aplicar la Búsqueda Local, se emplea el algoritmo del Enfriamiento Simulado.

Para implementarlo, en el algoritmo ILS he añadido un parámetro para distinguir entre el algoritmo ILS y el algoritmo ILS-ES.

En cada ejecución se han utilizado unos determinados parámetros. El número máximo de evaluaciones de la función objetivo se establece en 10000 evaluaciones. Además, se realizarán 10 iteraciones. También se considera que el número de elementos que permanecerán en la mutación será del 10% de elementos.

```
1   Input: número de iteraciones máximas nIterMAX, número de evaluaciones máximas
        nEvaluacionesMAX, porcentaje de número
2   de elementos que permanecen en la mutación porcentaje
3
4   S0 = generarSolucionAleatoria ()
5   S = ES (S0, nEvaluacionesMAX, 0.3, 0.3)
6   mejorSol = S
7   fitnessMejor = fitness (mejorSol)
8   n = mejorSol.size()
9   v = porcentaje*n/100
```

```
10
11    Para i ∈ {1,...,nIterMAX}
12      S1 = operadorMutacionSF (mejorSol, v)
13      S2 = ES (S1, nEvaluacionesMAX, 0.3, 0.3)
14      fitnessS2 = fitness (S2)
15
16      Si (fitnessS2 < fitnessMejor)
17        mejorSol = S2
18        fitnessMejor = fitnessS2
19      end
20    end
21
22    return mejorSol
```

4 Pseudocódigo de los algoritmos de comparación

En la práctica 1, he implementado el algoritmo **Greedy COPKM** para compararlo con el otro algoritmo de la práctica 1 (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 `practicaPAR`.

5.2.3 Ejecución

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

```
1    ./BIN/practicaPAR 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/practicaPAR 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.

6.2.1 Algoritmo de Enfriamiento Simulado (ES)

Resultados obtenidos por el algoritmo ES en el PAR con 10% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	13	0.611606	0.710115	0.353	52	0.188284	0.239454	3.538	210	0.128178	0.184456	9.526
22	9	0.631308	0.699506	0.437	68	0.186863	0.253778	3.543	266	0.12915	0.200435	9.676
100	10	0.629655	0.705431	0.542	50	0.191289	0.240491	3.507	184	0.120047	0.169357	9.427
222	10	0.620488	0.696264	0.471	52	0.188891	0.240061	3.523	243	0.133605	0.198726	9.485
273687	9	0.614174	0.682373	0.722	65	0.18711	0.251073	3.498	257	0.133619	0.202492	9.620
Media	10.2	0.6214462	0.6987378	0.505	57.4	0.1884874	0.2449714	3.5218	232	0.1289198	0.1910932	9.5468

Resultados obtenidos por el algoritmo ES en el PAR con 20% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	56	0.510917	0.73705	0.532	102	0.210275	0.263377	5.121	461	0.13218	0.196463	13.507
22	40	0.61125	0.772774	0.572	33	0.249345	0.266525	4.969	440	0.133129	0.194483	13.643
100	21	0.718862	0.803662	0.413	38	0.250121	0.269904	4.965	403	0.127013	0.183208	13.664
222	40	0.609028	0.770552	0.551	107	0.209993	0.265698	4.974	341	0.119711	0.16726	13.757
273687	31	0.597283	0.722464	0.254	39	0.249156	0.269459	5.007	457	0.125812	0.189537	13.800
Media	37.6	0.609468	0.7613004	0.4644	63.8	0.233778	0.2669926	5.0072	420.4	0.127569	0.1861902	13.6742

En cuanto al **número de restricciones incumplidas**, tenemos que son un poco más elevadas con el 20% de restricciones (el doble en algunos casos, en Glass apenas se nota). Además, en el conjunto de datos de Zoo esta medida es bastante menor en comparación con la tasa que tiene Bupa. Aunque esto último tiene sentido, pues Bupa tiene mayor número de restricciones máximas.

Con respecto a la **desviación general**, las diferencias apenas se notan al cambiar el número de restricciones. Aunque hay que observar que entre los conjuntos de datos, esta medida es mucho menor en el conjunto de datos Bupa que en Zoo, esto quiere decir que los datos están más cerca del centroide en Bupa que en Zoo. El conjunto de datos Glass también tiene una desviación general pequeña, aunque un poco mayor que Bupa.

Con respecto al **agregado**, la función objetivo, también obtenemos resultados muy similares cuando pasamos de considerar 10% de restricciones a 20% de restricciones, aunque en el caso de Bupa disminuye un poco. Si comparamos entre los conjuntos de datos, ocurre lo mismo que con la desviación general, Zoo tiene un agregado mayor que Bupa.

También hay que destacar el **tiempo** (medido en segundos). Conforme aumentamos el número de instancias de nuestro conjunto de datos, el tiempo aumenta notablemente. Esto es, en Zoo con 101 instancias el algoritmo apenas tarda medio segundo en obtener el resultado; pero en Bupa con 345 instancias, el tiempo aumenta considerablemente hasta los 9 segundos y 13 segundos, con el 10% y 20% de restricciones respectivamente.

6.2.2 Algoritmo de Búsqueda Multiarranque Básica (BMB)

Resultados obtenidos por el algoritmo BMB en el PAR con 10% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	7	0.616808	0.669852	0.213	48	0.201214	0.248448	0.531	553	0.186367	0.334564	1.047
22	7	0.616808	0.669852	0.221	40	0.217145	0.256506	0.539	529	0.18465	0.326416	1.046
100	8	0.59832	0.658942	0.231	38	0.222659	0.260052	0.529	522	0.193161	0.333051	1.051
222	10	0.604394	0.68017	0.234	57	0.210585	0.266676	0.536	528	0.185776	0.327274	1.052
273687	8	0.623561	0.684182	0.218	44	0.208861	0.252159	0.529	541	0.193367	0.338349	1.048
Media	8	0.6119782	0.6725996	0.2234	45.4	0.2120928	0.2567682	0.5328	534.6	0.1886642	0.3319308	1.0488

Resultados obtenidos por el algoritmo BMB en el PAR con 20% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	17	0.712827	0.781474	0.181	36	0.249131	0.267873	0.536	1063	0.177612	0.325838	1.035
22	17	0.704317	0.772964	0.179	31	0.248801	0.26494	0.526	952	0.175493	0.308241	1.045
100	40	0.604281	0.765804	0.179	34	0.247595	0.265296	0.525	994	0.185851	0.324455	1.070
222	13	0.711055	0.76355	0.211	43	0.245525	0.267911	0.525	1025	0.180437	0.323364	1.037
273687	12	0.731895	0.780352	0.166	101	0.210628	0.26321	0.522	1060	0.177205	0.325012	1.035
Media	19.8	0.692875	0.7728288	0.1832	49	0.240336	0.265846	0.5268	1018.8	0.1793196	0.321382	1.0444

Analicemos los datos correspondientes al algoritmo BMB.

Observamos que al pasar del 10% de restricciones al 20% de restricciones, en Zoo aumenta el número de restricciones incumplidas, la desviación general y el agregado; sin embargo, el tiempo disminuye. Este conjunto de datos sigue obtieniendo las peores medidas entre los tres.

Con respecto al Glass, el número de restricciones incumplidas es más similar, al igual que la desviación general y el agregado, que son un poco mayores al considerar el 20% de restricciones. Este conjunto de datos obtiene el menor agregado entre los tres conjuntos de datos considerados.

Con respecto al Bupa, el número de restricciones incumplidas es casi el doble al pasar de considerar el 10% al 20% de restricciones. Sin embargo, la desviación general y el agregado se mantienen similares, incluso llegan a disminuir al considerar el 20% de las restricciones. Además, obtiene la menor desviación general de entre los conjuntos de datos considerados.

6.2.3 Algoritmo de Búsqueda Local Reiterada (ILS)

Resultados obtenidos por el algoritmo ILS en el PAR con 10% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	9	0.608752	0.67695	0.221	27	0.225946	0.252515	0.530	492	0.18332	0.315171	1.040
22	10	0.611369	0.687146	0.226	56	0.187053	0.242159	0.546	519	0.195432	0.334518	1.044
100	9	0.611511	0.67971	0.223	31	0.218086	0.248592	0.528	556	0.183831	0.332833	1.038
222	9	0.609177	0.677376	0.231	48	0.199236	0.24647	0.533	540	0.170344	0.315057	1.040
273687	8	0.612355	0.672976	0.205	41	0.21303	0.253376	0.527	548	0.188945	0.335802	1.075
Media	9	0.6106328	0.6788316	0.2212	40.6	0.2086702	0.2486224	0.5328	531	0.1843744	0.3266762	1.0474

Resultados obtenidos por el algoritmo ILS en el PAR con 20% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	45	0.597152	0.778866	0.184	39	0.250149	0.270453	0.532	930	0.180138	0.309818	1.050
22	45	0.586468	0.768182	0.161	44	0.243045	0.265952	0.526	1022	0.17242	0.314929	1.057
100	34	0.617175	0.75447	0.176	42	0.245606	0.267472	0.530	935	0.188833	0.31921	1.055
222	18	0.707237	0.779922	0.177	35	0.245834	0.264055	0.528	1005	0.186831	0.326969	1.046
273687	11	0.723304	0.767723	0.169	26	0.253457	0.266993	0.529	974	0.18116	0.316975	1.046
Media	30.6	0.6462672	0.7698326	0.1734	37.2	0.2476182	0.266985	0.529	973.2	0.1818764	0.3175802	1.0508

Podemos observar que los resultados, salvo el número de restricciones incumplidas, se mantienen prácticamente iguales al considerar el 10% de restricciones o el 20% de restricciones. Además, sigue ocurriendo lo ya comentado anteriormente. Seguimos obteniendo resultados más bajos en la función objetivo con el conjunto de datos Glass, aunque los resultados de Bupa son también muy similares.

6.2.4 Algoritmo Híbrido ILS-ES

Resultados obtenidos por el algoritmo ILS-ES en el PAR con 10% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	9	0.622371	0.69057	0.747	54	0.188437	0.241575	3.847	435	0.163036	0.279611	11.296
22	9	0.609168	0.677366	0.690	48	0.190464	0.237698	3.791	482	0.153588	0.282759	10.514
100	8	0.62502	0.685641	0.726	52	0.186642	0.237812	3.818	542	0.162413	0.307663	10.385
222	7	0.619515	0.672559	0.772	51	0.187685	0.237871	3.648	502	0.162702	0.297232	10.588
273687	9	0.611205	0.679404	0.734	49	0.188532	0.23675	3.838	500	0.143949	0.277943	10.467
Media	8.4	0.6174558	0.681108	0.7338	50.8	0.188352	0.2383412	3.7884	492.2	0.1571376	0.2890416	10.65

Resultados obtenidos por el algoritmo ILS-ES en el PAR con 20% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	43	0.589079	0.762717	0.621	31	0.248801	0.26494	5.527	993	0.149746	0.288211	15.276
22	40	0.61125	0.772774	0.645	35	0.249177	0.267398	5.411	1000	0.152313	0.291754	15.395
100	33	0.624383	0.757639	0.621	34	0.24624	0.263941	5.503	1014	0.150097	0.29149	15.491
222	16	0.710871	0.77548	0.701	105	0.212441	0.267105	5.421	999	0.155042	0.294344	15.601
273687	9	0.731666	0.768009	0.594	105	0.208798	0.263462	5.541	986	0.149608	0.287097	15.580
Media	28.2	0.6534498	0.7673238	0.6364	62	0.2330914	0.2653692	5.4806	998.4	0.1513612	0.2905792	15.4686

Seguimos obteniendo unos resultados mucho más similares considerando el 10% o el 20% de restricciones y tampoco hay cambios entre lo ya comentado.

6.2.5 Resultados globales

Resultados globales en el PAR con 10% de restricciones												
Algoritmo	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
COPKM	7.4	0.9741962	1.0302698	0.0060	4.6	0.3786250	0.3831514	0.0378	41.6	0.2327392	0.2438876	0.3292
BL	15.0	0.5940342	0.7076986	0.5426	34.0	0.2194266	0.2528840	1.1332	117.6	0.1130182	0.1747302	8.5574
ES	10.2	0.6214462	0.6987378	0.5050	57.4	0.1884874	0.2449714	3.5218	232.0	0.1289198	0.1910932	9.5468
BMB	8.0	0.6119782	0.6725996	0.2234	45.4	0.2120928	0.2567682	0.5328	534.6	0.1886642	0.3319308	1.0488
ILS	9.0	0.6106328	0.6788316	0.2212	40.6	0.2086702	0.2486224	0.5328	531.0	0.1843744	0.3266762	1.0474
ILS-ES	8.4	0.6174558	0.6811080	0.7338	50.8	0.1883520	0.2383412	3.7884	492.2	0.1571376	0.2890416	10.6500

Resultados globales en el PAR con 20% de restricciones												
Algoritmo	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
COPKM	1.8	0.9422462	0.9495146	0.0064	1.6	0.3451134	0.3459460	0.0240	6.0	0.2367594	0.2380166	0.2098
BL	21.6	0.7011310	0.7883536	0.3854	102.8	0.2148274	0.2683460	1.1226	215.2	0.1154614	0.1454694	9.2414
ES	37.6	0.6094680	0.7613004	0.4644	63.8	0.2337780	0.2669926	5.0072	420.4	0.1275690	0.1861902	13.6742
BMB	19.8	0.6928750	0.7728288	0.1832	49.0	0.2403360	0.2658460	0.5268	1018.8	0.1793196	0.3213820	1.0444
ILS	30.6	0.6462672	0.7698326	0.1734	37.2	0.2476182	0.2669850	0.5290	973.2	0.1818764	0.3175802	1.0508
ILS-ES	28.2	0.6534498	0.7673238	0.6364	62.0	0.2330914	0.2653692	5.4806	998.4	0.1513612	0.2905792	15.4686

6.3 Análisis de resultados

En estas tablas podemos comparar los resultados obtenidos por todos los algoritmos implementados hasta el momento.

En cuanto al **número de restricciones incumplidas**, observamos que con el 10% de las restricciones en el conjunto de datos Zoo se mantiene muy parecida, de hecho los mejores resultados los obtenemos con COPKM, BMB e ILS. Con el 20% de restricciones los mejores resultados se obtienen con el algoritmo Greedy, y le siguen BMB y BL. En el conjunto de datos Glass con el 10% de restricciones, observamos que esta medida varía algo más entre los distintos algoritmos empleados. El mejor resultado lo obtiene COPKM, seguido de BL, ILS y BMB . El peor resultado lo obtiene ES. En cuanto al 20% de las restricciones, el mejor resultado lo obtenemos con COPKM, seguido de ILS y de BMB. Sin embargo, el peor resultado lo obtenemos con BL. Con respecto al conjunto de datos Bupa, con el 10% de las restricciones obtenemos mejores resultados con COPKM, BL y ES. El peor resultado lo obtiene BMB. Si consideramos el 20% de las restricciones, el mejor resultado lo obtiene COPKM, seguido de BL y seguido de ES. Sin embargo, el peor resultado lo obtiene BMB. En resumen, en los tres conjuntos de datos, considerando el 10% y el 20% de las restricciones, COPKM obtiene menor número de restricciones incumplidas. Aunque el algoritmo ES también obtiene pocas restricciones incumplidas en comparación con el resto de los algoritmos empleados.

En cuanto a la **desviación general**, observamos que Greedy obtiene peores resultados en todos los conjuntos de datos considerados. Los mejores resultados los obtienen Bupa y los algoritmos ES e ILS-ES. Estos dos últimos algoritmos obtienen una desviación general muy similar a BL, a veces es un poco mayor y otras veces incluso consiguen obtener resultados más bajos, aunque siguen manteniéndose muy similares. Sin embargo, los algoritmos de esta práctica obtienen unos resultados muy similares al aplicarlos entre los mismos conjuntos de datos, apenas hay diferencia.

Cabe destacar el **tiempo** (medido en segundos) empleado por cada algoritmo en los distintos conjuntos de datos. En el conjunto de datos Zoo apenas se nota el cambio, pues apenas tarda medio segundo. En Glass notamos un pequeño aumento, pues cada algoritmo tarda en ejecutarse entre medio segundo y casi los 4 segundos. Sin embargo, en Bupa observamos que, dependiendo del algoritmo empleado, podemos obtener tiempos de ejecución que varían entre medio segundo y los 15 segundos. Resaltar que los algoritmos ES e ILS-ES tardan bastante más, pues generan

un vecindario. De hecho, en Bupa observamos mayor diferencia pues este conjunto de datos tiene mayor número de instancias.

Con respecto al **agregado**, es decir, la función objetivo que queremos minimizar, también obtenemos unos resultados bastante coherentes. Esto es, en el conjunto de datos Zoo obtenemos que Greedy obtiene el peor resultado. Los algoritmos de esta práctica obtienen buenos resultados, incluso mejores que la BL. En este conjunto de datos, BMB consigue el menor agregado. En cuanto al conjunto de datos Glass, con el 10% de las restricciones, los algoritmos de esta práctica consiguen minimizar más el agregado que la BL. Lo mismo ocurre si consideramos el 20% de las restricciones, aunque en este caso son casi iguales. En cuanto al conjunto de datos Bupa, obtenemos mejores resultados con BL aunque el algoritmo ES obtiene resultados muy similares. El resto de algoritmos de esta práctica obtienen resultados peores que la BL, al haber tantas instancias y tantas posibilidades de agrupamientos, en este conjunto de datos los algoritmos funcionan peor. Hay que destacar también que BMB e ILS obtienen unos resultados bastante altos, pues al tener tantas instancias y al considerar toda la población para aplicarle la BL, se llega al máximo del número de las evaluaciones de la función objetivo mucho antes, por lo que no le da tiempo a minimizar esta función.

En resumen, los algoritmos de esta práctica obtienen unos resultados un poco mejores (salvo BMB, ILS e ILS-ES en Bupa) a los obtenidos por la búsqueda local. Esto se debe a que la población con la que trabajamos tiene mucha mayor diversidad y ayuda a que no se estanquen en mínimos locales.

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/