Práctica 2.b: Técnicas de Búsqueda basadas en Poblaciones 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

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 && 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
```

3.2 Algoritmos evolutivos basados en poblaciones

3.2.1 Introducción

La práctica 2 consiste en estudiar el funcionamiento de las Técnicas de Búsqueda basadas en Poblaciones para el PAR. A diferencia de la Búsqueda Local, con estos algoritmos trabajamos sobre un conjunto de soluciones, llamado **población**. Cada solución que forma parte de la población decimos que es un **cromosoma**. A su vez, cada cromosoma está formado por **genes**.

Estos conceptos los podemos trasladar al PAR de la siguiente forma:

- **Población**: representa un conjunto de cromosomas, es decir, un conjunto de soluciones.
- **Cromosoma**: representa una solución, que está formada por genes.
- **Gen**: representa un elemento de una solución. Además, cada elemento tiene asociado el cluster al que pertenece.

Estos algoritmos se basan en aplicar diferentes operadores sobre la población para mejorar cada cromosoma. Así, podremos explorar el espacio de búsqueda y obtendremos soluciones diversas, por lo que evitará estancarse en mínimos locales, como ocurría con la Búsqueda Local.

En esta práctica se han implementado dos tipos de algoritmos evolutivos.

- **Algoritmos Genéticos (AGs)**. Se implementarán dos variantes generacionales elitistas (**AGGs**) y otras dos estacionarias (**AGEs**).
- **Algoritmos Meméticos (AMs)**. Se implementarán tres variantes basadas en un AGG.

Ambos algoritmos tienen en común el esquema de representación, la función objetivo, la generación de la solución inicial, el esquema de generación de vecinos y el criterio de aceptación ya comentados anteriormente.

En estos algoritmos, a partir de una población inicial, podemos seleccionar algunos individuos que actúen como padres y obtener a partir de ellos unos hijos. A estos hijos se les aplicarán ciertas mutaciones para garantizar la diversidad y, finalmente, estos formarán parte de la nueva población.

3.2.2 Algoritmos Genéticos (AG)

Variantes implementadas

En esta práctica se han implementado dos variantes generacionales elitistas (**AGGs**) y otras dos estacionarias (**AGEs**), que se distinguen por el esquema de evolución y el operador de cruce empleado.

En cuanto a los valores de los parámetros, en ambos casos, se establece que el tamaño de la población será de 50 cromosomas, la probabilidad de cruce será de 0.7 por cromosoma, la probabilidad de mutación será de $0.1/numero\_de\_genes$ por gen y el criterio de parada será de relizar 100000 evaluaciones de la función objetivo, el tamaño de la población será de 50 cromosomas. Todos los Algoritmos Genéticos los he implementado en la misma función, estableciento al inicio los valores de los parámetros correspondientes en cada caso.

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

Esquema de evolución

El esquema de evolución consta de 4 pasos.

1. **Selección.** Se han implementado las siguientes variantes del operador de selección:
   - **Versión basada en el esquema generacional con elitismo (AGG)**. La población seleccionada tendrá exactamente el mismo tamaño que la población inicial.
   - **Versión basada en el esquema estacionario (AGE)**. De la población actual, solo se seleccionarán dos padres.

   Para realizar esta selección, se ha aplicado un **torneo binario** entre los padres. Esto es, se han elegido aleatoriamente dos padres de la población actual y, entre ellos, el padre que menor función objetivo tenga será quien formará parte de la siguiente población. Repetimos este proceso hasta conseguir el tamaño de la siguiente población deseado.
2. **Cruce.** Una vez obtenida la población formada por unos padres, los agruparemos por parejas y de ellos obtendremos unos hijos. Para ello, tenemos que tener en cuenta que los padres cruzan con una cierta probabilidad. Además, se implementarán dos variantes del operador de cruce:
   - **Operador de cruce uniforme (UN).** Al aplicar este operador a una pareja de padres, el resultado será un hijo con la mitad de genes de un padre y la otra mitad del otro padre. Para ello, se elige aleatoriamente la mitad de la mitad de un padre y el resto, se escoge del otro padre. De esta forma combinamos uniformemente las características de ambos padres.
   - **Operador de cruce por segmento fijo (SF).** En este operador, seleccionamos de forma aleatoria y copiamos un segmento continuo de genes de un padre y el resto de genes, se determinan de igual forma que en el cruce uniforme. Con este operador se obtiene mayor diversidad.

   Sin embargo, en ambos cruces podemos obtener un resultado puede ser incorrecto, pues algún cluster se puede quedar sin instancias asignadas. Para ello, realizaremos una reparación del cromosoma obtenido.
3. **Mutación.** Los hijos obtenidos pueden experimentar, con una probabilidad muy baja, cambios en sus genes. Esto es, tendremos un operador de mutación que, con cierta probabilidad, determinará si algún gen puede mutar, lo que significa que dicha instancia puede pasar a formar parte de otro cluster. En esta práctica haremos uso del operador de mutación uniforme. Sin embargo, obtenemos el mismo problema que al cruzar dos padres, en el resultado puede haber algún cluster sin instancias asignadas. Se soluciona de la misma forma.
4. **Reemplazamiento.** En esta fase, la población obtenida sustituirá a la población actual. En el caso de que el reemplazamiento se haga con elitismo, se conservará el mejor cromosoma de la población actual en la nueva población.

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

Generación de la población inicial

```
1   Input: tamaño de la población actual M, número de clusters k, número de
        genes n
2
3   cromosoma = vector de enteros vacío
4   pActual = matriz de enteros vacía
5   pActualFitness = vector vacío
6
7   Para i ∈ {0,...,M-1}
8     Hacer
9       cromosoma.clear()
10
11      Para i ∈ {0,...,n-1} do
12        gen = número aleatorio entero entre [0, k)
13        cromosoma.añadir(gen)
14      end
15
16      // Comprobar que ningún cluster se ha quedado vacío
17      Inicializar el contador de clusters a 0
18
19      Para i ∈ {0,...,n-1} do
20        Incrementar el contador del cluster al que cromosoma[i] está asociado
21      end
22
23      Para i ∈ {0,...,k-1} do
24        Comprobar si algún contador del cluster i es 0 (vacío)
25      end
26    Mientras (haya clusters sin instancias asignadas)
27
28    pActual.añadir(cromosoma)
29    pActualFitness.añadir(fitness(pActual[i]))
30  end
31
32  return pActual, pActualFitness
```

Operador selección

Este operador es común a ambas variantes generacionales del Algoritmo Genético, la única diferencia es el número de iteraciones realizadas.

```
1   Input: población actual pActual, fitness de la población actual
        pActualFitness, tamaño de la población actual M, tamaño de la población
        seleccionada m
2
3   pSiguiente = vector de enteros vacío
4
5   Para i ∈ {0, ..., m}
6     c1 = número aleatorio entero en [0, M)
7
8     Hacer
9       c2 = número aleatorio entero en [0, M)
10    Mientras (c1 == c2)
11
12    Si pActualFitness[c1] > pActualFitness[c2]
```

```
13          pSiguiente.añadir(pActual[c2])
14        Si no
15          pSiguiente.añadir(pActual[c1])
16        end
17      end
18      return pSiguiente
```

Operador de reparación

Con este operador, comprobamos si un cromosoma no es válido y, en ese caso, lo reparamos.

```
1       Input: cromosoma C, número de genes n, número de clusters k
2
3       contador = vector de enteros de tamaño k inicializado a 0
4
5       Para i ∈ {0,...,n}
6         Incrementar el contador del cluster al que C_i está asociado
7       end
8
9       Para i ∈ {0,...,contador.size()}
10        Si el contador del cluster i es 0 (vacío)
11          Hacer
12            j = número aleatorio entero en [0, n)
13          Mientras (contador[C[j]] - 1 <= 0)
14
15          C[j] = i
16          Decrementar contador[C[j]]
17          Incrementar contador[i]
18        end
19      end
20
21      return C
```

Operador de cruce

Para realizar un cruce, tenemos que tener en cuenta que la probabilidad con la que se cruzan dos individuos es de 0.7. Para no tener que elegir números aleatorios en cada iteración, calculamos a priori cuántos cruces debe haber, esto es $nCruces = probCruce * m/2.0$, donde $nCruces$ es el número de cruces, $probCruce$ es la probabilidad de cruzar y m es el tamaño de la población seleccionada.

```
1       Input = población siguiente pSiguiente, tamaño de la población siguiente m,
            número de cruces nCruces
2
3       numCruces = nCruces
4
5       Para i ∈ {0,...,m} && nCruces > 0
6         Si i es par
7           pSiguiente.añadir(operadorCruce(pSiguiente[i], pSiguiente[i+1]))
8         Si no
9           pSiguiente.añadir(operadorCruce(pSiguiente[i], pSiguiente[i-1]))
10          Decrementar nCruces
11        end
12      end
13
```

```
14      // Eliminar los padres que han cruzado
15      Para i ∈ {0,...,numCruces}
16        pSiguiente.eliminar(pSiguiente.begin())
17        pSiguiente.eliminar(pSiguiente.begin())
18      end
19
20      return pSiguiente
```

El siguiente esquema es independiente del operador de cruce que usemos, solo tendríamos que sustituir `operadorCruce` por el que queramos usar.

Operador de cruce uniforme (UN)

```
1       Input: primer padre p1, segundo padre p2, número de genes n
2
3       posP1 = vector de enteros vacío   // Almacenar las posiciones de los
           genes de p1
4       C = vector de enteros de tamaño n inicializado a -1
5
6       Para i ∈ {0,...,n/2}
7         Hacer
8           pos = número aleatorio entero en [0, n)
9         Mientras (pos esté en posP1)
10
11        posP1.añadir(pos)
12        C[pos] = p1[pos]
13      end
14
15      Para i ∈ {0,...,n}
16        Si C[i] == -1     // No se le ha asignado gen de p1
17          C[i] = p2[i]
18        end
19      end
20
21      Si queda algún cluster vacío en C
22        Reparar C
23      end
24
25      return C
```

Operador de cruce por segmento fijo (SF)

```
1       Input: primer padre p1, segundo padre p2, número de genes n
2
3       C = vector de enteros de tamaño n
4       posP1 = vector de enteros     // Almacenar las posiciones de los genes de
            p1
5       r = número aleatorio entero entre [0, n)
6       v = número aleatorio entero entre [0, n]
7
8       Para i ∈ {0,...,v}
9         C[r] = p1[r]
```

```
10          posP1.añadir(r)
11          r = (r + 1) % n
12        end
13
14        nIter = n - v     // Iteraciones restantes
15
16        Para i ∈ {0,...,nIter/2}
17          Hacer
18            pos = número aleatorio entero en [0, n)
19          Mientras (pos esté en posP1)
20
21          posP1.añadir(pos)
22          C[pos] = p1[pos]
23        end
24
25        Para i ∈ {0,...,n}
26          Si C[i] == -1     // No se le ha asignado gen de p1
27            C[i] = p2[i]
28          end
29        end
30
31        Si queda algún cluster vacío en C
32          Reparar C
33        end
34
35        return C
```

Operador de mutación

Para realizar una mutación, tenemos que tener en cuenta que la probabilidad con la que se produce una mutación de un individuo es de $0.1/numero\_de\_genes$. En el caso del AGG, para no tener que elegir números aleatorios en cada iteración, calculamos a priori cuántas mutaciones debe haber, esto es $nMutaciones = probMutacion * m * n$, donde $nMutaciones$ es el número de mutaciones, $probMutacion$ es la probabilidad de cruzar, m es el tamaño de la población seleccionada y n es el número de genes. Sin embargo, en el AGE no es tan necesario calcularlos a priori, pues el tamaño de la población es 2, por lo que no habría problema en generar números aleatorios.

```
1         Input = probabilidad de mutación probMutacion, población pSiguiente, tamaño
              de la población m, número de genes n
2
3         nMutaciones = 0
4
5         // Calculo cuántas mutaciones se van a realizar
6         Si la evolución es AGG
7           nMutaciones = probMutacion * m * n
8         Si la evolución es AGE
9           Para i ∈ {0,...,m}
10            prob = número aleatorio entre [0.0, 1.0)
11
12            Si (prob < probMutacion * n)
13              Incrementar nMutaciones
14            end
15          end
16        end
```

```
17
18      Para i ∈ {0,...,nMutaciones}
19        pos = número aleatorio entero entre [0, m)
20        pSiguiente[pos] = operadorMutacionUN(pSiguiente[pos])
21      end
22
23      return pSiguiente
```

Operador de mutación uniforme

```
1       Input: cromosoma C, número de genes n, número de clusters k
2
3       Hacer
4         posGen = número aleatorio entero en [0, n)
5         gen = número aleatorio entero en [0, k)
6         C[posGen] = gen
7       Mientras quede algún cluster vacío en C
8
9       return C
```

Esquema de reemplazamiento

```
1       Input: población actual pActual, población siguiente pSiguiente, fitness de
           la población actual pActualFitness,
2       tamaño de la población actual M, tamaño de la población siguiente m, nú
           mero de evaluaciones fitness nFitness
3
4       Si la evolución es AGG
5         posCMejor = posición de pActualFitness que tenga el menor valor
6         CMejor = pActual[posCMejor]
7         fit_min = pActualFitness[posCMejor]
8         pActual = pSiguiente
9
10        Para i ∈ {0,...,M}
11          Evaluar pActualFitness
12          Incrementar nFitness
13        end
14
15        Si (CMejor no está en pActual)
16          Calcular la posición de pActualFitness que tenga el mayor valor
17          Asignar a pActual en esa posición CMejor
18          Asignar a pActualFitness en esa posicion fit_min
19        end
20      Si la evolución es AGE
21        Para i ∈ {0,...,m}
22          posCPeor = posición de pActualFitness que tenga el mayor valor
23          fit = fitness de pSiguiente[i]
24          Incrementar nFitness
25
26          Si la fitness de posCPeor es mayor que fit
27            Asignar a pActual en la posición posCPeor pSiguiente[i]
28            Asignar a pActualFitness la posición posCPeor fit
```

```
29          end
30        end
31      end
32
33      return pActual, pActualFitness
```

Implementación de los Algoritmos Genéticos

Al realizar esta práctica, he considerado una única función para implementar todas las variantes de los Algoritmos Genéticos. Para ello, en los parámetros de entrada podremos especificar las distintas variantes de estos algoritmos. Por lo que el esquema de la implementación general de un algoritmo genético es el siguiente, en el que se hace referencia a las funciones vistas en los apartados anteriores.

```
1       Input: tamaño de la población M, número de genes n, tipo de generación
           evolucion, operador de cruce operadorCruce, probabilidad de cruce probCruce,
            número de clusters k
2
3       // Inicializar variables con respecto a los parámetros de entrada
4       probMutacion = 0.1/n
5
6       // m es el tamaño de la población siguiente
7       Si (evolucion == "G") // generacional
8         m = M
9       Si (evolucion == "E")   // estacionario
10        m = 2
11      end
12
13      Generar la población inicial aleatoriamente y evaluarla
14
15      nFitnessMAX = 100000
16      nFitness = 0    // contador del numero de evaluaciones de la función
           objetivo
17      t = 0       // número de la generación
18
19      Mientras (nFtiness < nFitnessMAX) hacer
20        // SELECCIONAR
21        pSiguiente = operador de selección
22
23        // CRUZAR
24        nCruces = probCruce *m/2.0
25        pSiguiente = operador de cruce (operadorCruce, nCruces)
26
27        // MUTAR
28        pSiguiente = operador de mutación
29
30        // REEMPLAZAR Y EVALUAR
31        pActual,pActualFitness = esquema de reemplazamiento
32
33        Incrementar t
34      end
35
36      // Actualizamos los atributos con la solución
37      posCMejor = posición de pActualFitness que tenga el menor valor
38      clusters = pActual[posCMejor]
```

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

```
39        desvGeneral = desviacionGeneral(pActual[posCMejor])
40        fitness = fitness(pActual[posCMejor])
41        infeasibility = infeasibility(pActual[posCMejor])
42
43        return clusters
```

3.2.3 Algoritmos Meméticos (AM)

Los algoritmos meméticos implementados se basan en un algoritmo genético generacional, pero cada un cierto número de generaciones se le aplica una búsqueda local a un cierto número de cromosomas que conforman la población.

Así, con el algoritmo genético generacional podremos explorar nuevas zonas del espacio y, con la búsqueda local, podremos minimizar la función objetivo de ciertos cromosomas, pues encontraremos mínimos locales.

El algoritmo genético generacional escogido para implementar el algoritmo memético es el que usa el operador de cruce uniforme, pues da mejores resultados que el de por segmento fijo. El esquema de evolución de este algoritmo es similar al algoritmo genético generacional, salvo que después de la mutación y cada 10 generaciones se le aplica la búsqueda local a un cierto número de cromosomas. Esta búsqueda local no es la misma que la búsqueda local implementada en la práctica 1, sino que es más suave. Esto se debe a que la búsqueda local de la práctica 1 hacía que las soluciones convergieran demasiado rápido. De esta forma, la búsqueda local suave solo recorrerá cada cromosoma una sola vez y tendremos en cuenta el número de fallos (cuando la BLS no produce cambios en el cromosoma), pues este no podrá superar un cierto límite.

En cuanto a los valores de los parámetros, se establece el número de fallos máximo permitido es $nFallosMAX = 0.1 * n$, donde n es el número de genes de cada cromosoma de la población. Al igual que en los algoritmos genéticos, el tamaño de la población será de 50 cromosomas, la probabilidad de cruce será de 0.7 por cromosoma, la probabilidad de mutación será de $0.1/numero\_de\_genes$ por gen y el criterio de parada será de relizar 100000 evaluaciones de la función objetivo.

En esta práctica se han implementado los tres siguientes Algoritmos Meméticos.

- **AM-(10, 1.0)**: la búsqueda local suave se aplica sobre todos los cromosomas de la población cada 10 generaciones.
- **AM-(10, 0.1)**: la búsqueda local suave se aplica sobre un subconjunto de cromosomas de la población seleccionado aleatoriamente con una probabilidad de 0.1 para cada cromosoma cada 10 generaciones.
- **AM-(10, 0.1mej)**: la búsqueda local suave se aplica sobre los $0.1 * M$ mejores cromosomas de la población actual (M es el tamaño de esta) cada 10 generaciones.

Esquema de reemplazamiento

He realizado algunos cambios con respecto al esquema de reemplazamiento de los Algoritmos Genéticos.

```
1       Input: población actual pActual, población siguiente pSiguiente, fitness de
           la población actual pActualFitness,
2       fitness de la población siguiente pSigFitness
3
4       posCMejor = posición de pActualFitness que tenga el menor valor
5       CMejor = pActual[posCMejor]
6       fit_min = pActualFitness[posCMejor]
7       pActual = pSiguiente
8       pActualFitness = pSigFitness
9
10      Si (CMejor no está en pActual)
11        Calcular la posición de pActualFitness que tenga el mayor valor
12        Asignar a pActual en esa posición CMejor
13        Asignar a pActualFitness en esa posicion fit_min
14      end
15
16      return pActual, pActualFitness
```

Esquema de búsqueda

El esquema de búsqueda se sitúa después de la fase de mutación, pero antes de la fase de reemplazamiento.

```
1       Input: población siguiente pSiguiente, fitness de la población siguiente
           pSigFitness, tamaño de la población actual M, número de evaluaciones
           fitness nFitness, número de generación t, hibridación, probabilidad de
           selección probSeleccion
2
3       Si t es múltiplo de 10
4         Si la hibridación no es AM-(10, 0.1mej)
5           Para i ∈ {0,...,M}
6             Si probSeleccion es 1.0 or randfloat(0.0, 1.0) < probSeleccion
7               nFitness += BLS (pSiguiente[i], pSigFitness[i], nFallos)  // dos
                   primeros parámetros pasados por referencia
8             end
9           end
10        Si la hibridación es AM-(10, 0.1mej)
11          m = 0.1*M
12          // Ordenar de menor a mayor
13          Para i ∈ {1,...,M}
14            Para j ∈ {0,...,M-i}
15              Si (pSigFitness[j] > pSigFitness[j+1])
16                Intercambiar los valores de pSigFitness[j] y pSigFitness[j+1]
17                Intercambiar los valores de pSiguiente[j] y pSiguiente[j+1]
18              end
19            end
20          end
21          Para i ∈ {1,...,m}
22            nFitness += BLS (pSiguiente[i], pSigFitness[i], nFallos)  // dos
                 primeros parámetros pasados por referencia
23          end
```

```
24          end
25        end
26
27        return pSiguiente, pSigFitness
```

Búsqueda Local Suave

```
1         Input: cromosoma (solución) S, fitness del cromosoma S fitnessActual, número
              de fallos máximo nFallosMAX, número de clusters k
2
3         n = S.size()    // número de genes
4         nFallos = 0
5         i = 0
6         mejora = true
7         fit_min = fitnessActual
8         indicesDatos = vector de enteros vacío
9
10        Para i ∈ {0,...,n}
11          indicesDatos.añadir(i)
12
13        RSI <- RandomShuffle(indicesDatos)  //Barajar los índices a los datos
14
15        Mientras ((mejora or nFallos < nFallosMAX) and i < n)
16          mejora = false
17
18          Para j ∈ {0,...k}
19            c_i = S[RSI[i]]   // Cluster asociado a la instancia RSI[i]
20            par <- (c_i, j)   // Par formado por el cluster actual y el cluster j
21
22            // Comprobamos que no sean iguales y que el par sea válido (esto es,
                  que no deje ningún cluster vacío al realizar cambiar el cluster c_i
                  por j)
23            Si (c_i != j and parValido(par, S))
24              S[RSI[i]] = j   // Asociamos el cluster j a la instancia RSI[i]
25              fit = fitness(S)
26              Incrementar nEvaluaciones
27
28              Si (fit < fit_min)  // Si mejora la función objetivo, actualizamos
29                fit_min = fit
30                mejora = true
31              Si no       // Si no mejora la función objetivo, asignamos el
                    cluster anterior
32                S[RSI[i]] = c_i
33              end
34            end
35          end
36
37          Si (no hay mejora)
38            Incrementar nFallos
39          end
40
41          Incrementar i
42        end
43
```

```
44      return nEvaluaciones, S, fitnessActual  // S y fitnessActual se devuelven
            por referencia
```

Implementación de los Algoritmos Meméticos

Al realizar esta práctica, he considerado una única función para implementar todas las variantes de los Algoritmos Meméticos. Para ello, en los parámetros de entrada podremos especificar las distintas variantes de estos algoritmos. Por lo que el esquema de la implementación general de un algoritmo memético es el siguiente, en el que se hace referencia a las funciones vistas en los apartados anteriores.

```
1       Input: tamaño de la población M, número de genes n, tipo de hibridación
           hibridacion, probabilidad de cruce probCruce, número de clusters k
2
3       // Inicializar variables con respecto a los parámetros de entrada
4       probMutacion = 0.1/n
5       nMutaciones = probMutacion * M * n
6       nFallos = 0.1*n
7       prob = 0
8
9       // m es el tamaño de la población a aplicarle BLS
10      Si (hibridacion == "1.0")
11        probSeleccion = 1.0
12        m = M
13        mejor = false
14      Si (hibridacion == "0.1")
15        probSeleccion = 0.1
16        m = probSeleccion * M
17        mejor = false
18      Si (hibridacion == "0.1mej")
19        probSeleccion = 0.1
20        m = probSeleccion * M
21        mejor = true
22      end
23
24      Generar la población inicial aleatoriamente y evaluarla
25
26      nFitnessMAX = 100000
27      nFitness = 0    // contador del numero de evaluaciones de la función
           objetivo
28      t = 1       // número de la generación
29
30      Mientras (nFtiness < nFitnessMAX) hacer
31        // SELECCIONAR
32        pSiguiente = operador de selección
33
34        // CRUZAR
35        nCruces = probCruce *m/2.0
36        pSiguiente = operador de cruce uniforme
37
38        // MUTAR
39        pSiguiente = operador de mutación
40
41        // BLS
42        // Evaluar la nueva población
```

```
43          Para i ∈ {0,...M}
44            pSigFitness = fitness(pSiguiente[i])
45            Incrementar nFitness
46          end
47
48          Si (t es múltiplo de 10)
49            pSiguiente, pSigFitness = esquema de búsqueda
50          end
51
52          // REEMPLAZAR Y EVALUAR
53          pActual = esquema de reemplazamiento
54
55          Incrementar t
56        end
57
58        // Actualizamos los atributos con la solución
59        posCMejor = posición de pActualFitness que tenga el menor valor
60        clusters = pActual[posCMejor]
61        desvGeneral = desviacionGeneral(pActual[posCMejor])
62        fitness = fitness(pActual[posCMejor])
63        infeasibility = infeasibility(pActual[posCMejor])
64
65        return clusters
```

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 `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.

6.2.1 Algoritmo Genético Generacional con operador de cruce uniforme (AGG-UN)

Resultados obtenidos por el algoritmo AGG-UN 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	0.616729	0.669773	0.932	94	0.21078	0.30328	3.856	642	0.161176	0.338852	9.771
22	3	0.696395	0.719128	0.906	117	0.291395	0.406528	3.782	623	0.172427	0.339384	9.194
100	7	0.756588	0.809631	0.902	34	0.258035	0.291492	3.745	517	0.162132	0.300682	9.289
222	9	0.618185	0.686383	0.908	45	0.224313	0.268595	3.584	647	0.165722	0.33911	9.172
273687	9	0.610618	0.678817	0.911	58	0.235553	0.292627	3.570	630	0.163313	0.332146	9.174
Media	7	0.659642	0.7633408	0.9118000	69.6	0.2440152	0.3125044	3.7074000	611.8	0.1649540	0.3300348	9.3200000

Resultados obtenidos por el algoritmo AGG-UN en el PAR con 20% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	10	0.733622	0.774003	1.026	94	0.24375	0.292687	4.807	1271	0.170463	0.347692	13.631
22	16	0.800919	0.865528	0.973	99	0.209938	0.280197	4.721	1330	0.166984	0.35244	12.913
100	23	0.712827	0.805703	0.974	126	0.200623	0.26622	4.769	1236	0.157651	0.33	13.572
222	36	0.61315	0.758521	0.971	122	0.261986	0.3255	4.684	1181	0.157295	0.321975	12.918
273687	13	0.710965	0.76346	0.973	114	0.214026	0.273376	4.703	1095	0.159297	0.311985	12.899
Media	19.6	0.7142966	0.7934430	0.9834000	111.0	0.2266004	0.2804316	4.7368000	1222.6	0.1623380	0.3328184	13.1866000

Como podemos observar, en ambos casos obtenemos unos resultados muy similares.

En cuanto al **número de restricciones incumplidas**, tenemos que son un poco más elevadas con el 20% de restricciones (el doble aproximadamente). 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 Glass 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. Sin embargo, Glass consigue obtener menor agregado que Bupa, aunque no hay gran diferencia.

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 1 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 Genético Generacional con operador de cruce de segmento fijo (AGG-SF)

Resultados obtenidos por el algoritmo AGG-SF en el PAR con 10% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	6	0.621439	0.666904	1.001	48	0.207558	0.254792	3.952	664	0.174835	0.352779	9.791
22	25	0.747291	0.936732	0.957	115	0.249864	0.363029	3.927	664	0.162338	0.340282	9.510
100	16	0.604719	0.725961	0.956	43	0.241458	0.283772	3.856	649	0.168756	0.34268	9.435
222	7	0.599621	0.652665	0.959	165	0.300234	0.462601	3.697	614	0.168778	0.333322	9.526
273687	12	0.606473	0.697405	0.961	68	0.258702	0.325617	3.693	674	0.169531	0.350155	9.602
Media	13.2	0.6359086	0.7359334	0.9668000	87.8	0.2515632	0.3379622	3.8250000	653.0	0.1688476	0.3438436	9.6328000

Resultados obtenidos por el algoritmo AGG-SF en el PAR con 20% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	18	0.667604	0.74029	1.083	96	0.24029	0.290268	4.864	1284	0.153488	0.332531	12.948
22	9	0.754158	0.790501	1.028	116	0.20881	0.269201	4.848	1135	0.170115	0.328381	12.186
100	26	0.705901	0.810891	1.022	68	0.236814	0.272215	4.801	1272	0.159274	0.336643	12.583
222	62	0.871068	1.12143	1.019	78	0.256109	0.296716	4.866	1263	0.161611	0.337725	12.114
273687	21	0.693301	0.778101	1.030	108	0.258661	0.314887	4.806	1130	0.163048	0.320616	12.120
Media	27.2	0.7384064	0.8482426	1.0364000	93.2	0.2401368	0.2886574	4.8370000	1216.8	0.1615072	0.3311792	12.3902000

En este caso volvemos a tener las mismas conclusiones que con el operador de cruce uniforme.

Al pasar del 10% de restricciones al 20% de restricciones, en Zoo aumenta el número de restricciones incumplidas, la desviación general, el agregado y el tiempo (aunque en este último no se aprecia gran diferencia). Este conjunto de datos sigue obteniendo 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, aunque estos últimos disminuyen 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. Además, obtiene la menor desviación general de entre los conjuntos de datos considerados.

6.2.3 Algoritmo Genético Estacionario con operador de cruce uniforme (AGE-UN)

Resultados obtenidos por el algoritmo AGE-UN en el PAR con 10% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	6	0.624073	0.669538	1.022	53	0.209767	0.261921	4.035	527	0.151924	0.293154	9.792
22	3	0.679621	0.702354	0.979	46	0.250655	0.295921	4.013	554	0.163209	0.311674	9.617
100	21	0.71289	0.87202	0.979	48	0.258733	0.305967	3.959	604	0.155568	0.317433	9.582
222	19	0.773927	0.917902	0.982	65	0.225725	0.289688	3.786	553	0.142867	0.291064	9.554
273687	20	0.565396	0.716948	0.980	94	0.267731	0.360231	3.780	605	0.168274	0.330407	9.721
Media	13.8	0.671152	0.7799214	0.9884000	61.2	0.2425222	0.3027456	3.9146000	568.6	0.1563684	0.3087464	9.5332000

Resultados obtenidos por el algoritmo AGE-UN en el PAR con 20% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	19	0.730876	0.807599	1.119	62	0.241121	0.273399	4.966	1026	0.154744	0.29781	12.011
22	38	0.61912	0.772567	1.049	28	0.237202	0.251779	5.044	1102	0.166333	0.319997	12.300
100	37	0.591113	0.740522	1.104	196	0.221104	0.323144	4.921	994	0.143613	0.282218	12.526
222	11	0.777527	0.821946	1.054	131	0.195658	0.263858	4.915	1048	0.143829	0.289963	12.251
273687	12	0.718619	0.767076	1.060	82	0.270145	0.312835	4.914	1122	0.148791	0.305244	12.226
Media	23.4	0.6874510	0.7819420	1.0772000	99.8	0.2308080	0.2852010	4.9615000	1058.4	0.1514614	0.299041	12.4628

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 Genético Estacionario con operador de cruce de segmento fijo (AGE-SF)

Resultados obtenidos por el algoritmo AGE-SF en el PAR con 10% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	17	0.851007	0.979827	1.108	31	0.240894	0.2714	4.185	454	0.173861	0.295527	9.534
22	8	0.594655	0.655277	1.053	64	0.204385	0.267364	4.174	596	0.165332	0.325053	9.891
100	5	0.656766	0.694654	1.053	39	0.255808	0.294186	4.158	536	0.163286	0.306928	9.832
222	3	0.698947	0.721679	1.051	55	0.221483	0.275605	4.000	621	0.14771	0.31413	9.978
273687	7	0.611213	0.664256	1.058	96	0.281656	0.376123	3.928	546	0.154073	0.300394	9.778
Media	8.0	0.6825176	0.7431386	1.0646000	57.0	0.2408452	0.2969356	4.0890000	550.6	0.1608524	0.3084064	9.8626000

Resultados obtenidos por el algoritmo AGE-SF en el PAR con 20% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	23	0.713615	0.806491	1.196	58	0.247927	0.278122	5.076	1082	0.157517	0.308392	12.364
22	31	0.586849	0.712029	1.117	33	0.250079	0.267259	5.107	1113	0.15694	0.312137	12.428
100	11	0.716085	0.760504	1.117	119	0.234565	0.296518	5.040	1063	0.153892	0.302118	12.814
222	20	0.720629	0.801391	1.125	48	0.241654	0.266643	5.030	1148	0.143141	0.303219	11.398
273687	11	0.761687	0.806106	1.122	64	0.246541	0.27986	5.044	992	0.151156	0.289482	12.462
Media	19.2	0.6997730	0.7773042	1.1354000	64.4	0.2441532	0.2776804	5.0594000	1079.6	0.1525292	0.3030696	12.69320

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 Algoritmo Memético (AM(10, 1.0))

Resultados obtenidos por el algoritmo AM(10, 1.0) en el PAR con 10% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	21	0.735185	0.894316	0.890	153	0.237252	0.38781	3.745	768	0.217242	0.423057	14.162
22	22	0.730314	0.897022	0.915	140	0.2593	0.397066	3.709	725	0.214201	0.408492	14.994
100	8	0.607626	0.668247	0.913	76	0.283497	0.358284	3.721	732	0.216566	0.412733	14.791
222	10	0.61995	0.695727	0.935	32	0.228566	0.260056	3.840	632	0.217278	0.386646	14.797
273687	7	0.609378	0.662422	0.883	223	0.267884	0.487325	3.889	737	0.217196	0.414703	14.184
Media	13.6	0.6604906	0.7635468	0.9072000	124.8	0.2552998	0.3781082	3.7808000	718.8	0.2164966	0.4091262	14.7856000

Resultados obtenidos por el algoritmo AM(10, 1.0) en el PAR con 20% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	16	0.704467	0.769077	1.045	108	0.198405	0.254631	5.064	1332	0.194935	0.380671	22.165
22	12	0.754505	0.802962	1.086	77	0.242111	0.282198	4.927	1482	0.209528	0.41618	21.351
100	12	0.704538	0.752995	1.099	117	0.228453	0.289365	4.902	1522	0.213668	0.425897	22.295
222	12	0.71877	0.767226	1.090	85	0.239904	0.284156	4.886	1316	0.20384	0.387344	22.417
273687	17	0.712827	0.781474	1.073	101	0.219622	0.272203	5.022	1176	0.190346	0.354329	21.547
Media	12.8	0.7215220	0.7687846	1.0786000	97.6	0.225696	0.27651	4.9602	1365.6	0.2024634	0.3928842	21.95500

En cuanto al numero de restricciones incumplidas, observamos que en Zoo y en Glass disminuyen al considerar el 20% de las restricciones en comparación con el 10% de las restricciones. Sin embargo, la desviación general y el agregado en Zoo aumentan un poco, pero en Glass disminuyen. En cuanto al tiempo empleado en estos dos conjuntos de datos, apenas hay diferencias. Con respecto al conjunto de datos Bupa, el número de restricciones incumplidas sigue aumentando considerablemente al considerar el 10% y 20% de las restricciones. Sin embargo, la desviación general y el agregado se mantienen similares, incluso llegan a disminuir un poco al considerar el 20% de las restricciones. Aunque la diferencia entre los tiempos es bastante grande.

Seguimos obteniendo que Glass obtiene menores resultados en la función objetivo.

6.2.6 Algoritmo Memético (AM(10, 0.1))

Resultados obtenidos por el algoritmo AM(10, 0.1) en el PAR con 10% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	20	0.80683	0.958383	0.953	56	0.186887	0.241993	3.774	240	0.149366	0.213683	9.329
22	6	0.605297	0.650763	0.971	27	0.233863	0.260432	3.867	249	0.140293	0.207022	9.173
100	2	0.70367	0.718825	0.983	47	0.195301	0.241551	3.851	242	0.129072	0.193925	9.256
222	15	0.717589	0.831254	0.977	14	0.189513	0.241667	3.829	157	0.145561	0.187635	9.041
273687	17	0.829324	0.958144	0.976	56	0.189351	0.244457	3.855	244	0.132694	0.198083	9.287
Media	12.0	0.7325420	0.8234738	0.9720000	42.8	0.1989830	0.2460200	3.8352000	226.4	0.1393972	0.2000696	9.2172000

Resultados obtenidos por el algoritmo AM(10, 0.1) en el PAR con 20% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	15	0.69809	0.758661	1.121	42	0.250108	0.271974	5.426	423	0.143491	0.192897	12.540
22	14	0.709731	0.758188	1.155	29	0.24928	0.264377	5.449	449	0.138578	0.194369	12.448
100	16	0.71792	0.78253	1.129	94	0.194802	0.260399	5.488	270	0.130711	0.16836	12.594
222	16	0.724873	0.789483	1.150	101	0.206209	0.25879	5.407	451	0.134871	0.197758	12.570
273687	19	0.703637	0.780361	1.135	121	0.201137	0.26413	5.426	378	0.138892	0.191601	12.022
Media	16.0	0.701556	0.7678202	1.1380000	83.8	0.2203072	0.2639340	5.4392000	394.2	0.1373086	0.1902760	12.4348000

Seguimos obteniendo resultados muy parecidos con el 10% y el 20% de restricciones, aunque hay que destacar que la función objetivo es algo menor en el 20% de restricciones en Zoo y Bupa, aunque en Glass aumenta muy poco. Además, en este algoritmo no obtenemos tanta diferencia entre los tiempos de ejecución en el conjunto de datos Bupa. También cabe resaltar que el número de restricciones inclumplidas en Bupa, no aumenta tan drásticamente como ocurría antes.

6.2.7 Algoritmo Memético (AM(10, 0.1mej))

Resultados obtenidos por el algoritmo AM(10, 0.1mej) en el PAR con 10% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	8	0.773446	0.834067	0.957	12	0.248656	0.260464	3.847	126	0.123256	0.157023	9.082
22	9	0.584812	0.653011	0.975	39	0.210392	0.250738	3.858	179	0.126753	0.174723	9.127
100	10	0.600063	0.67584	0.968	37	0.1962	0.23261	3.840	249	0.122729	0.189458	9.083
222	11	0.624686	0.70804	0.990	48	0.185721	0.232955	3.869	212	0.126298	0.183111	9.024
273687	6	0.602718	0.648184	0.942	52	0.187526	0.238696	3.859	210	0.126964	0.183241	9.098
Media	8.8	0.6371450	0.7038284	0.9664000	37.6	0.2116120	0.2447504	3.8546000	195.2	0.1252000	0.1775112	9.0828000

Resultados obtenidos por el algoritmo AM(10, 0.1mej) en el PAR con 20% de restricciones												
Semilla	Zoo				Glass				Bupa			
	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T	Infeasable	DesvGen	Agr.	T
7	36	0.611821	0.804567	1.126	111	0.205401	0.263189	5.423	472	0.139079	0.204895	10.594
22	14	0.733343	0.841299	1.135	39	0.244055	0.264359	5.451	236	0.138753	0.171661	10.622
100	21	0.695684	0.766212	1.139	34	0.241753	0.259454	5.439	392	0.116866	0.171526	10.685
222	10	0.70747	0.761229	1.135	31	0.242694	0.258833	5.428	245	0.123429	0.157592	10.878
273687	19	0.712457	0.760553	1.147	36	0.246296	0.265038	5.464	225	0.121497	0.152871	10.782
Media	20.0	0.69206	0.7867720	1.1364000	50.2	0.2346206	0.2603150	5.4410000	314.0	0.1276882	0.1761858	10.7122000

En este caso aumenta un poco la desviación general en Zoo y Glass al pasar del 10% de restricciones al 20% de restricciones, aunque en Bupa se mantiene similar. Sin embargo, la función objetivo aumenta en Zoo y en Glass al pasar del 10% al 20% de restricciones, pero Bupa disminuye mínimamente. Seguimos obteniendo menor diferencia entre el número de restricciones incumplidas, en especial en el conjunto de datos de Bupa.

6.2.8 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
AGG-UN	7.0	0.6596420	0.7633408	0.9118	69.6	0.2440152	0.3125044	3.7074	611.8	0.1649540	0.3300348	9.3200
AGG-SF	13.2	0.6359086	0.7359334	0.9668	87.8	0.2515632	0.3379622	3.8250	653.0	0.1688476	0.3438436	9.6328
AGE-UN	13.8	0.6711520	0.7799214	0.9884	61.2	0.2425222	0.3027456	3.9146	568.6	0.1563684	0.3087464	9.5332
AGE-SF	8.0	0.6825176	0.7431386	1.0646	57.0	0.2408452	0.2969356	4.0890	550.6	0.1608524	0.3084064	9.8626
AM(10, 1.0)	13.6	0.6604906	0.7635468	0.9072	124.8	0.2552998	0.3781082	3.7808	718.8	0.2164966	0.4091262	14.7856
AM(10, 0.1)	12.0	0.7325420	0.8234738	0.9720	42.8	0.1989830	0.2460200	3.8352	226.4	0.1393972	0.2000696	9.2172
AM(10, 0.1mej)	8.8	0.6371450	0.7038284	0.9664	37.6	0.2116120	0.2447504	3.8546	195.2	0.1252000	0.1775112	9.0828

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	0.024	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
AGG-UN	19.6	0.7142966	0.7934430	0.9834	111.0	0.2266004	0.2804316	4.7368	1222.6	0.1623380	0.3328184	13.1866
AGG-SF	27.2	0.7384064	0.8482426	1.0364	93.2	0.2401368	0.2886574	4.8370	1216.8	0.1615072	0.3311792	12.3902
AGE-UN	23.4	0.6874510	0.7819420	1.0772	99.8	0.2308080	0.2852010	4.9615	1058.4	0.1514614	0.2990410	12.4628
AGE-SF	19.2	0.6997730	0.7773042	1.1354	64.4	0.2441532	0.2776804	5.0594	1079.6	0.1525292	0.3030696	12.6932
AM(10, 1.0)	12.8	0.7215220	0.7687846	1.0786	97.6	0.2256960	0.2765100	4.9602	1365.6	0.2024634	0.3928842	21.9550
AM(10, 0.1)	16.0	0.7015560	0.7678202	1.1380	83.8	0.2203072	0.2639340	5.4392	394.2	0.1373086	0.1902760	12.4348
AM(10, 0.1mej)	20.0	0.6920600	0.7867720	1.1364	50.2	0.2346206	0.2603150	5.4410	314.0	0.1276882	0.1761858	10.7122

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, AGG-UN, AGE-SF y AM(10, 0.1mej). Con el 20% de restricciones este número se mantiene muy similar, salvo en el algoritmo Greedy que es mucho menor. También

obtenemos mejores resultados de esta medida en COPKM, AGG-UN, AGE-SF, AM(10,1.0) y AM(10, 0.1). 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, AM(10, 0.1mej) y AM(10, 0.1). El peor resultado lo obtiene AM(10, 1.0) con diferencia. En cuanto al 20% de las restricciones, el mejor resultado lo obtenemos con COPKM, seguido de AM(10, 0.1mej), seguido de AGE-SF y seguido de AM(10, 0.1). Sin embargo, el peor resultado lo obtenemos con AGG-UN. Con respecto al conjunto de datos Bupa, con el 10% de las restricciones obtenemos mejores resultados con COPKM, BL, AM(10, 0.1mej) y AM(10, 0.1). El peor resultado lo obtiene AM(10, 1.0). Si consideramos el 20% de las restricciones, el mejor resultado lo obtiene COPKM, seguido de BL, seguido de AM(10, 0.1mej) y seguido de AM(10, 0.1). Sin embargo, el peor resultado lo obtiene AM(10, 1.0). 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 los algoritmos meméticos AM(10, 0.1) y AM(10, 0.1mej) también obtienen 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 meméticos AM(10, 0.1) y AM(10, 0.1mej). 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 genéticos 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 esta práctica se ha notado cómo ha aumentado esta medida al implementar los algoritmos genéticos y meméticos. Hemos pasado de apenas tardar de ejecutar cada algoritmo de apenas medio segundo, a tardar casi 22 segundos en el peor de los casos. En el conjunto de datos Zoo apenas se nota el cambio, pues apenas tarda 1 segundo. En Glass notamos un pequeño aumento, pues cada algoritmo tarda en ejecutarse unos 4 segundos. Sin embargo, en Bupa observamos que, dependiendo del algoritmo empleado, podemos obtener tiempos de ejecución que varían entre los 8 segundos y los 21 segundos. Esto tiene sentido, pues en Greedy y en BL solo trabajabamos con un vector solución, en esta práctica hemos trabajado con un conjunto de vectores solución, en concreto 50, por lo que el número de operaciones que realizamos aumenta considerablemente. 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 genéticos obtienen buenos resultados, aunque en el caso del 10% de las restricciones, son un poco peores que los obtenidos por BL; si consideramos el 20% de las restricciones, los resultados son más similares incluso con AGE obtenemos mejores resultados que con BL, aunque apenas se nota la diferencia. Si comparamos la BL con los algoritmos meméticos, en el 10% de las restricciones AM(10, 1.0) y AM(10, 0.1) obtienen unos resultados mayores que los de la BL, pero AM(10, 0.1mej) consigue obtener minimizar la función objetivo, obteniendo el mejor resultado. En cuanto al 20% de las restricciones, los algoritmos meméticos obtienen resultados mucho mejores que la BL, pues la función objetivo resultante es menor. En cuanto al conjunto de datos Glass, con el 10% de las restricciones, BL consigue obtener mejor resultado que los algoritmos genéticos, pero los algoritmos meméticos AM(10, 0.1) y AM(10, 0.1mej) consiguen minimizar más todavía esta función. Lo mismo ocurre si consideramos el 20% de las restricciones. En cuanto al conjunto de datos Bupa, obtenemos mejores resultados con BL aunque

los algoritmos meméticos AM(10, 0.1) y AM(10, 0.1mej) obtienen resultados muy similares. Los algoritmos genéticos siguen obtienendo resultados peores. Hay que destacar también que AM(10, 1.0) obtiene unos resultados bastante altos, pues al tener tantas instancias y al considerar toda la población para aplicarle la BLS, 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 genéticos obtienen unos resultados un poco peores a los obtenidos por la búsqueda local. Sin embargo, los algoritmos meméticos, salvo el AM(10, 1.0), obtienen unos resultados similares e incluso en algunos casos mejores que 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. El algoritmo AM(10, 1.0) no obtiene tan buenos resultados como los otros meméticos, pues aplica la BLS a todos los cromosomas de la población, esto hace que converja mucho más rápido, sobre todo cuando el conjunto de datos tiene mayor número de instancias. También encontramos una diferencia entre los algoritmos genéticos considerados, pues los generacionales obtienen, en general, peores resultados que los estacionarios. Esto se debe a que la exploración de los algoritmos estacionarios es mucho mayor y encuentra soluciones con más diversidad.

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/