Problema de Agrupamiento con Restricciones (PAR)

Técnicas de Búsqueda basadas en Poblaciones para el PAR

Metaheurísticas: Práctica 2.b, Grupo MH2 (Jueves de 17:30h a 19:30h)

Jorge Sánchez González - 75569829V
jorgesg97@correo.ugr.es

26 de abril de 2020

Índice

1. El problema

1.1. Descripción del problema

El **agrupamiento** o **análisis de clusters** clásico (en inglés, *clustering*) es un problema que persigue la clasificación de objetos de acuerdo a posibles similitudes entre ellos. Así, se trata de una técnica de aprendizaje no supervisado que permite clasificar en grupos (desconocidos a priori) objetos de un conjunto de datos que tienen características similares.

El **Problema del Agrupamiento con Restricciones (PAR)** (en inglés, Constrained Clustering, CC) es una generalización del problema del agrupamiento clásico. Permite incorporar al proceso de agrupamiento un nuevo tipo de información: las restricciones. Al incorporar esta información la tarea de aprendizaje deja de ser no supervisada y se convierte en semi-supervisada.

El problema consiste en, dado un conjunto de datos X con n instancias, encontrar una partición C del mismo que minimice la desviación general y cumpla con las restricciones del conjunto de restricciones R. En nuestro caso concreto, solo consideramos restricciones de instancia (Must-Link o Cannot-Link) y todas ellas las interpretaremos como restricciones débiles (Soft); con lo que la partición C del conjunto de datos X debe minimizar el número de restricciones incumplidas pero puede incumplir algunas. Formalmente, buscamos

$$\text{Minimizar } f = \hat{C} + \lambda * infeasibility$$

donde:

- C es una partición (una solución al problema), que consiste en una asignación de cada instancia a un cluster.
- $\hat{C}$ es la desviación general de la partición C, que se define como la media de las desviaciones intra-cluster.
- $infeasibility$ es el número de restricciones que C incumple.
- λ es un hyperparámetro a ajustar (en función de si le queremos dar más importancia a las restricciones o a la desviación general).

1.2. Conjuntos de datos considerados

Trabajaremos con 6 instancias del PAR generadas a partir de los 3 conjuntos de datos siguientes:

1. Iris: Información sobre las características de tres tipos de flor de Iris. Se trata de 150 instancias con 4 características por cada una de ellas. Tiene 3 clases ($k = 3$).
2. Ecoli: Contiene medidas sobre ciertas características de diferentes tipos de células. Se trata de 336 instancias con 7 características. Tiene 8 clases ($k = 8$).
3. Rand: Está formado por tres agrupamientos bien diferenciados generados en base a distribuciones normales ($k = 3$). Se trata de 150 instancias con 2 características.
4. Newthyroid: Contiene medidas tomadas sobre la glándula tiroides de múltiples pacientes. Presenta 3 clases distintas ($k = 3$) y se trata de 215 instancias con 5 características.

Para cada conjunto de datos se trabaja con 2 conjuntos de restricciones, correspondientes al 10 % y 20 % del total de restricciones posibles.

2. Descripción de la aplicación de los algoritmos

En esta sección se describen las consideraciones comunes a los distintos algoritmos. Se incluyen la representación de las soluciones, la función objetivo y los operadores comunes a los distintos algoritmos. Ya que los únicos puntos en común de todos los algoritmos (incluyendo los de la práctica 1 y la 2) son la función objetivo y la representación de las soluciones, no estudiaremos ningún operador común. No se han incluido tampoco los detalles específicos de ninguno de los algoritmos en esta sección.

Una primera consideración general es el hecho de que la distancia usada como medida de cercanía (o similitud) entre las distintas instancias y clusters es la distancia euclídea.

El lenguaje utilizado para la implementación de la práctica ha sido **Python**.

2.1. Representación de la soluciones

Las soluciones a nuestro problema serán las llamadas *particiones*, que asignan a cada instancia del conjunto de datos uno de los k clusters. Aunque existen varias formas de representar una partición, por simplicidad se ha decidido utilizar la misma forma en todos los algoritmos. En concreto, respresentamos una partición de un conjunto de n instancias en k clusters con una lista S de tamaño n cuyos valores son enteros $S_i \in \{0, 1, 2, ..., k-1\}$. Así, el valor de la posición i del vector, S_i, indica el cluster al que la i-ésima instancia, x_i, ha sido asignada.

Por verlo con un ejemplo, la representación será de la siguiente forma:

```
partition = [0,0,0,5,0,1,1,1,1,1,2,3,1,1,1,...]
```

Esta partición indicaría, por ejemplo, que la instancia x_0 se ha asignado al cluster número 0, c_0, y que la instancia x_3 se ha asignado al cluster c_5.

Cabe mencionar también que durante la ejecución de los algoritmos, las particiones en proceso de construcción tendrán sus valores en $\{-1, 0, 1, 2, ..., k-1\}$. Cuando una posición i tenga el valor -1, esto indicará que aún no se le ha asignado ningún cluster a la instancia x_i.

2.2. Función objetivo

Como hemos visto en la seccion anterior (1.1) para calcular la función objetivo se necesita saber tanto la desviación general de la partición, $\hat{C}$, como el número de restricciones que esta viola, $infeasibility$. Se muestra a continuación su expresión y su pseudocódigo.

$$f(C) = \hat{C} + \lambda * infeasibility$$

Algorithm 1: objective_function

Data: Conjunto de datos X, partición en forma de vector S, lista de restricciones *constraints_list*, lista de centroides *centroids*, hiperparámetro *lambda*
Result: function_value
begin
  dev ← general_deviation(X, S, centroids)
  inf ← $infeasibility$(S, constraints_list)
  function_value ← dev + lambda * inf
  return function_value
end

El hyperparámetro λ que se usará por defecto será el mínimo indicado en las diapositivas del Seminario 2, es decir, el cociente entre la distancia máxima existente en el conjunto de datos y el número de restricciones del problema.

Cabe mencionar que la función objetivo como tal será usada por todos los algoritmos excepto por el Greedy, que usará la desviación general y la infactibilidad en diferentes etapas (y por separado). De cualquier manera, la calidad de las soluciones de todos los algoritmos van a ser evaluadas con esta función.

2.2.1. Desviación general de una partición

La desviación general de una partición se define como la media de las distancias medias intra-cluster, esto es,

$$\hat{C} = \frac{1}{k} \sum_{c_i \in C} \hat{c}_i.$$

Donde las distancias medias intra-cluster se definen a su vez como

$$\hat{c_i} = \frac{1}{|c_i|} \sum_{\vec{x_j} \in c_i} distance(\vec{x_j}, \vec{\mu_i}).$$

En pseudocódigo:

Algorithm 2: mean_dist_intra_cluster

```
Data: Conjunto de datos X, partición S, cluster del que queremos calcular su distancia
      intra cluster cluster_id, centroide del cluster centroid
Result: distance_intra_cluster
begin
    Y ← instancias_asignadas_al_cluster(X, S, cluster_id)
    if centroid == None then
        centroid = mean(Y, axis = 0)
    end
    distances ← list([ ])
    foreach y ∈ Y do
        distances.append(distance(y, centroid))
    end
    distance_intra_cluster ← media(distances)
    return distance_intra_cluster
end
```

Algorithm 3: general_deviation

```
Data: Conjunto de datos X, partición S, lista de centroides centroids
Result: deviation
begin
    cluster_ids ← {0,1,...,longitud(centroids)-1}
    intra_cluster_distances ← list([ ])
    foreach c_id ∈ cluster_ids do
        d ← mean_dist_intra_cluster(X, S, c_id, centroids[c_id])
        intra_cluster_distances.append(d)
    end
    deviation ← media(intra_cluster_distances)
    return deviation
end
```

Cabe comentar que la lista de centroides que se pasa como argumento es opcional, y en caso de no pasarse se calculará cada centroide directamente a la hora de calcular su distancia media intra-cluster asociada (como indica el condicional *if* en la función mean_dist_intra_cluster (2)).

2.2.2. Infactibilidad

La infactibilidad de una partición (*infeasibility*) se define como el número de restricciones que incumple. Por ello para calcularla utilizamos una función auxiliar, *V*, que nos va a decir, dada una partición, y un par de instancias con un valor de restricción (-1,0 o 1), si la partición incumple alguna restricción asociada a las dos instancias. Por otro lado, cabe destacar también que usamos la lista de restricciones, y no la matriz de restricciones, para recorrer todas las restricciones para calcular el *infeasibility* de una manera más eficiente.

Algorithm 4: V (si se incumple alguna restricción o no)

Data: Índice i de la instancia x_i, índice j de la instancia x_j, partición S, valor de restricción *constraint_value*)
Result: incumplimiento
begin
  incumplimiento_ML $\leftarrow$ (*constraint_value* == 1 and $S[i] \neq S[j]$)
  incumplimiento_CL $\leftarrow$ (*constraint_value* == -1 and $S[i] == S[j]$)
  incumplimiento $\leftarrow$ (incumplimiento_ML or incumplimiento_CL)
  return incumplimiento
end

Algorithm 5: *infeasibility*

Data: Partición S, lista de restricciones *constraints_list*
Result: infeasibility
begin
  infeasibility $\leftarrow$ 0
  foreach $(i, j, contraint\_value) \in$ *constraints_list* **do**
    infeasibility $\leftarrow$ infeasibility + V(i, j, S, constraint_value)
  end
  return infeasibility
end

3. Algoritmos

En esta práctica se han usado nueve algoritmos. El Greedy (*COPKM*) y el de Búsqueda Local (ya implementados en la práctica 1) se han ejecutado sobre el nuevo conjunto de datos *Newthyroid*. El resto (*AGG-UN*, *AGG-SF*, *AGE-UN*, *AGE-SF*, *AM-(10,1.0)*, *AM-(10,0.1)* y *AM-(10,0.1mej)*) se han implementado y se han ejecutado sobre todos los conjuntos de datos. Para la descripción de los dos primeros algoritmos se hace referencia a la memoria de la práctica anterior. El resto de algoritmos (los de la práctica 2) son técnicas de búsqueda basadas en poblaciones y se describen a continuación mostrando pseudocódigo y comentando los aspectos más importantes.

Un primer aspecto importante en la implementación de todos ellos es que se trabaja con poblaciones con tripletas de la forma

$$(particion, valor\_funcion\_objetivo, contadores\_de\_asignaciones)$$

donde *contadores_de_asignaciones* es un diccionario utilizado para comprobar que no se rompe la restricción fuerte (de no asignar ninguna instancia a algún cluster) cada vez que se crea o modifica una partición y *valor_funcion_objetivo* se emplea para ordenar las particiones en función de su evaluación en la función objetivo.

3.1. Operadores de cruce

En primer lugar vamos a describir los operadores de cruce que se usan en estos algoritmos, ya que son comunes a todos ellos. Los operadores de cruce se usarán para crear nuevas soluciones (o particiones) combinando soluciones que ya teníamos. En concreto en esta práctica se emplean dos operadores de cruce para representación real, explicados en el Seminario 3. Uno de ellos será el operador uniforme mientras que el otro será el de cruce por segmento fijo.

3.1.1. Operador de cruce uniforme

Algorithm 6: uniform_cross_operator

Data: Partición padre 1 *father1*, partición padre 2 *father2*
Result: Partición hija *child*
begin
  n ← longitud(father1)
  RSI ← list([0,1,...,n-1])
  RSI ← RandomShuffle(RSI)
  child ← list(longitud = n, valores = 0)
  foreach $i \in \{0, 1, ..., truncar(n/2)\}$ **do**
    child[RSI[i]] ← father1[RSI[i]]
  end
  foreach $i \in \{truncar(n/2), ..., n-1\}$ **do**
    child[RSI[i]] ← father2[RSI[i]]
  end
  return child
end

Como se observa en el pseudocódigo, este operador consiste en crear una partición hijo como resultado de coger la mitad de los genes escogidos de forma aleatoria de un primer padre y la otra mitad de los genes de un segundo padre (cuando el número de genes es impar se toma un gen más del segundo padre).

3.1.2. Operador de cruce por segmento fijo

Este operador se basa en tomar un "segmento de genes" que empieza en una posición aleatoria y tiene una longitud también aleatoria. Dicho segmento se copia tal cual del primer padre a la partición hija. El resto de genes se obtienen con el operador de cruce (aplicado tan solo a los genes que no estaban en el segmento). Este operador es menos disruptivo, pues la partición hija copia un mayor número de genes del padre 1 y es más probable que se parezca bastante a este. Se describe en pseudocódigo en el bloque siguiente (7).

3.1.3. Reparación de soluciones

Ambos operadores de cruce podrían dar lugar a particiones que rompieran la restricción fuerte de que cada cluster tenga al menos una instancia asignada, por lo que necesitamos una función que las repare en dicho caso, y que se puede describir en pseudocódigo como se observa en el bloque 8.

Algorithm 7: fixed_segment_operator

Data: Partición padre 1 *father1*, partición padre 2 *father2*
Result: Partición hija *child*
begin
  n ← longitud(father1)
  segment_ini ← entero_aleatorio_entre(0, n-1)
  segment_lenght ← entero_aleatorio_entre(0, n-1)
  segment_indices ← [i mod(n) for $i \in$ {segment_ini,..., segment_ini + segment_lenght}]
  child ← list(longitud = n, valores = -1)
  foreach $i \in$ *segment_indices* **do**
    child[i] ← father1[i]
  end
  -Para el resto de genes que no están en el segmento: Cruce uniforme
  RSI ← indices_con_valor(-1, child)
  RSI ← RandomShuffle(RSI)
  n_2 ← longitud(RSI)
  foreach $i \in \{0, 1, ..., truncar(n_2/2)\}$ **do**
    child[RSI[i]] ← father1[RSI[i]]
  end
  foreach $i \in \{truncar(n_2/2), ..., n_2 - 1\}$ **do**
    child[RSI[i]] ← father2[RSI[i]]
  end
  return child
end

Algorithm 8: repair_partition

Data: Partición a reparar S, diccionario con número de instancias asignadas a cada cluster *assignations_counter*, número de clusters k
Result: Partición reparada S y diccionario de asignaciones actualizado *assignations_counter*
begin
  n ← longitud(S)
  foreach $i \in \{0, 1, ..., k - 1\}$ **do**
    if *assignations_counter[i] == 0* **then**
      gen_idx ← entero_aleatorio_entre(0,n-1)
      while *assignations_counter[S[gen_idx]]* ≤ 1 **do**
        gen_idx ← entero_aleatorio_entre(0,n-1)
      end
      assignations_counter[S[gen_idx]] -= 1
      S[gen_idx] ← i
      assignations_counter[i] ← 1
    end
  end
  return S, assignations_counter
end

Básicamente la reparación de particiones consiste en asignar una instancia escogida de forma aleatoria a los clusters que no tienen ninguna instancia asignada (siempre comprobando que al hacer esta nueva asignación no estamos dejando a otro cluster con 0 instancias asignadas).

3.2. Algoritmos Genéticos

Ahora sí, nos lanzamos a describir los algoritmos propiamente. Empezamos con los algoritmos genéticos, dentro de los cuáles tendremos los generacionales y los estacionarios. Ambos van a generar su población inicial de la misma forma; como se describe a continuación.

Algorithm 9: generate_initial_population

Data: Conjunto de datos X, lista de restricciones *const_list*, número de clusters k, hiperparámetro λ, tamaño de la población *population_size*, y contador de evaluaciones de la función objetivo *counter*
Result: Población inicial generada *current_population* y contador de evaluaciones de la función objetivo actualizado *counter*

```
begin
    current_population ← list([ ])
    foreach i ∈ {0, 1, ..., population_size − 1} do
        -Declaramos diccionarios assignations_counter para vigilar cuantas instancias
         tiene cada cluster (para asegurar que nunca tienen 0) en cada partición
        valid_partition ← False
        while not valid_partition do
            S ← generate_initial_sol(X, k)
            assignations_counter ← cuenta_asignaciones(S)
            if asignaciones_validas(assignations_counter) then
                valid_partition ← True
            end
        end
        func_value ← objective_func(X, S, const_list, λ)
        counter ← counter + 1
        current_population.append([S, func_value, assignations_counter])
    end
    return current_population, counter
end
```

Se trata de generar $M = population\_size$ particiones de forma aleatoria y comprobando que cumplen la restricción fuerte de que no haya ningún cluster con ninguna instancia asignada. Esta función hace uso de la función *generate_initial_sol(X, k)*, que ya implementamos en la práctica 1 y que genera una partición de forma aleatoria como se describe en el siguiente pseudocódigo.

Algorithm 10: generate_initial_sol

Data: Conjunto de datos X, número de clusters k
Result: Una partición S

```
begin
    n ← longitud(X); S ← list([ ])
    foreach i ∈ {0, 1, ..., n − 1} do
        S.append(entero_aleatorio_entre(0,k-1))
    end
    return S
end
```

3.2.1. Algoritmos Genéticos Generacionales

Vemos ahora el pseudocódigo de una parte crucial de los algoritmos genéticos generacionales; la siguiente función es la que se encarga de pasar de una población $P(t)$ a la siguiente generación $P(t+1)$. Dentro de este bloque de pseudocódigo se encuentran descritos el proceso de selección (mediante el torneo binario), de cruce y de mutación. Cabe destacar que el operador de cruce (ya sea el uniforme o el de segmento fijo) es simplemente pasado como argumento a la función.

Algorithm 11: new_generation

Data: Conjunto de datos X, lista de restricciones *const_list*, número de clusters k, hiperparámetro λ, tamaño de la población *population_size*, contador de evaluaciones *counter*, número de cruces esperados *n_cross_expected*, número de instancias n, operador de cruce *cross_operator*, número de mutaciones esperadas *n_mutations_expected*, población actual *current_population*

Result: Nueva población *new_population* y contador de evaluaciones *counter*

begin

new_population ← list([ ])

1-Selección

foreach $i \in \{0, 1, ..., population\_size - 1\}$ **do**

-Torneo binario - Aprovechamos que la población está ordenada

father_idx ← minimo(RandomSample($\{0, 1, ..., population\_size - 1\}$, 2))

new_population.append(current_population[father_idx])

end

2-Aplicamos el operador de cruce a las parejas de forma ordenada

foreach $i \in \{0, 1, ..., n\_cross\_expected - 1\}$ **do**

father1 ← new_population[i*2][0]

father2 ← new_population[i*2 + 1][0]

foreach $j \in \{0, 1\}$ **do**

child_partition ← cross_operator(father1, father2)

assignations_counter ← cuenta_asignaciones(child_partition)

if *not asignaciones_validas(assignations_counter)* **then**

repair_partition(child_partition, assignations_counter, k)

end

-No llamamos aún a la función objetivo porque sería un malgasto si estas nuevas soluciones mutaran...

new_population[i*2 + j] ← [child_partition, -1, assignations_counter]

end

end

3-Mutaciones: **foreach** $i \in \{0, 1, ..., n\_mutations\_expected - 1\}$ **do**

cromo_idx ← entero_aleatorio_entre(0,population_size-1)

gen_idx ← entero_aleatorio_entre(0,n-1)

-Comprobamos que la mutación en el gen elegido no rompe las restricciones (fuertes)

while *new_population[cromo_idx][2][new_population[cromo_idx][0][gen_idx]] ≤ 1* **do**

gen_idx ← entero_aleatorio_entre(0,n-1)

end

new_population[cromo_idx][2][new_population[cromo_idx][0][gen_idx]] -= 1

-Cambiamos de forma aleatoria el cluster asociado a esta instancia

new_population[cromo_idx][0][gen_idx] ← (new_population[cromo_idx][0][gen_idx] + entero_aleatorio_entre(1,k-1)) mod k

new_population[cromo_idx][2][new_population[cromo_idx][0][gen_idx]] += 1

new_population[cromo_idx][1] ← -1

end

4-Llamamos a la función objetivo el número de veces estrictamente necesario (solo para los cromosomas nuevos)

foreach *cromo* ∈ *new_population* **do**

if *cromo[1] == -1* **then**

cromo[1] ← objective_func(X, cromo[0], const_list, λ)

counter += 1

end

end

return new_population, counter

end

Ahora sí, hemos descrito todas las funciones que usaremos dentro de la implementación principal del algoritmo.

Algorithm 12: generational_genetic_algo

Data: Conjunto de datos X, matriz de restricciones *const_matrix*, lista de restricciones *const_list*, número de clusters k, hiperparámetro λ, operador de cruce *cross_operator*, tamaño de la población *population_size*, probabilidad de cruce *cross_prob*, probabilidad de mutación *mutation_prob*

Result: La mejor partición de la población final

```
begin
    n ← longitud(X)
    -Número esperado de cruces (2 hijos por cruce)
    n_cross_expected ← truncar(cross_prob * population_size / 2)
    n_mutations_expected ← truncar(mutation_prob * population_size * n)
    counter ← 0 -Contará el número de evaluaciones de la función objetivo
    1-Generamos la población inicial:
    current_population, counter ← generate_initial_population(X, const_list, k, λ,
     population_size, counter)
    -Ordenamos la población según la calidad de los cromosomas (de mejor a peor)
    ordenar_según_func_objetivo(current_population)
    while counter < 100000 do
        2-Generamos la población siguiente
        new_population, counter ← new_generation(X, const_list, k, λ, population_size,
         counter, n_cross_expected, n, cross_operator, n_mutations_expected,
         current_population)
        ordenar_según_func_objetivo(new_population)
        3-Aplicamos el elitismo
        if current_population[0][1] < new_population[0][1] then
            new_population[population_size-1] ← current_population[0]
            ordenar_según_func_objetivo(new_population)
        end
        current_population ← new_population
    end
    return current_population[0][0]
end
```

Del anterior pseudocódigo caben destacar dos aspectos. El primero de ellos es que el algoritmo sigue generando nuevas poblaciones (a partir de la anterior) hasta que hayamos evaluado la función objetivo 100000 veces. Puesto que la comprobación del contador de evaluaciones solo se hace cada vez que terminamos el reemplazamiento de la población (en el bucle *while*), tendremos que se pueden sobrepasar las 100000 pudiendo llegar a

$$(100000 - 1) + n\_cross\_expected + n\_mutations\_expected$$

evaluaciones (esto pasará también con el resto de algoritmos; en el foro de esta práctica se nos indicó que esto no sería ningún problema). El segundo aspecto que cabe señalar es la aplicación del elitismo, que se produce justo después de obtener la siguiente generación de individuos y que se basa en conservar el mejor elemento de la población anterior si este es mejor que todos los de la nueva población. Con este segundo aspecto se completa la descripción del esquema de evolución y reemplazamiento del algoritmo.

Para utilizar particularmente los algoritmos *AGG-UN* y *AGG-SF* bastará con usar este algoritmo pasándole como argumento los operadores de cruce *uniform_cross_operator* y *fi-*

xed_segment_operator respectivamente (ambos ya han sido presentados). El tamaño de la población y las probabilidades de cruce y de mutación usados en ambas variantes son las que se nos indican en el guión de la práctica, 50, 0,7 y 0,001 respectivamente.

3.2.2. Algoritmos Genéticos Estacionarios

Los algoritmos genéticos estacionarios no se basarán en generar nuevas poblaciones que sustituyan a las anteriores sucesivamente; sino que consistirán en generar tan solo dos nuevos individuos en cada iteración, de forma que estos reemplacen a los dos peores de la población (si son mejores que ellos en términos de la función objetivo). Se muestra el pseudocódigo en la siguiente página (algoritmo 14).

En él se pueden observar claramente los esquemas de evolución y reemplazamiento. Como comentábamos, solo dos individuos son generados en cada iteración; además es importante notar que esta vez solo estos dos individuos son los que pueden mutar (en el algoritmo generacional podían mutar todos los individuos de la población, incluso aquellos que no habían sido producto de un cruce). También cabe señalar que la probabilidad de cruce en este algoritmo es 1 (es por ello que no se recibe como argumento). Finalmente cabe comentar que para tener en cuenta la probabilidad de mutación se ha elegido por simplicidad trabajar con la probabilidad de mutación a nivel de cromosoma

$$cromo\_mutation\_prob = mutation\_prob * n\_genes,$$

generando para cada cromosoma resultante de un cruce un número aleatorio en $[0, 1]$ y si sale menor que *cromo_mutation_prob*, mutando un gen aleatorio de ese cromosoma.

De nuevo, para utilizar particularmente los algoritmos *AGE-UN* y *AGE-SF* bastará con usar este algoritmo pasándole como argumento los operadores de cruce *uniform_cross_operator* y *fixed_segment_operator* respectivamente. Además, el tamaño de la población y la probabilidad de mutación por gen que se han usado son 50 y 0,001, como se indicaba en el guión de la práctica.

Algorithm 13: stable_genetic_algo

Data: Conjunto de datos X, lista de restricciones *const_list*, número de clusters k, hiperparámetro λ, operador de cruce *cross_operator*, tamaño de la población *population_size*, probabilidad de mutación *mutation_prob*

Result: La mejor partición de la población final

begin

n ← longitud(X)
cromo_mutation_prob ← mutation_prob * n
counter ← 0
1-Generamos la poblacion inicial
current_population, counter ← generate_initial_population(X, const_list, k, λ, population_size, counter)
ordenar_según_func_objetivo(current_population)
while *counter* < 100000 **do**
2-Seleccionamos dos individuos mediante torneo binario
parents ← list([ ])
foreach $i \in \{0, 1\}$ **do**
-Torneo binario - Aprovechamos que la población está ordenada
father_idx ← minimo(RandomSample($\{0, 1, ..., population\_size\}$), 2))
parents.append(current_population[father_idx])
end
children ← list([ ])
foreach $i \in \{0, 1\}$ **do**
3-Aplicamos el operador de cruce a la pareja de padres
child_partition ← cross_operator(parents[0][0], parents[1][0])
assignations_counter ← cuenta_asignaciones(child_partition)
if *not asignaciones_validas(assignations_counter)* **then**
repair_partition(child_partition, assignations_counter, k)
end
4-Mutación según cierta probabilidad
if *real_aleatorio_entre(0,1)* $\leq$ *cromo_mutation_prob* **then**
gen_idx ← entero_aleatorio_entre(0,n-1)
-Aseguramos que cambiar el gen elegido no rompe la restricción fuerte
while *assignations_counter[child_partition[gen_idx]]* ≤ 1 **do**
gen_idx ← entero_aleatorio_entre(0,n-1)
end
assignations_counter[child_partition[gen_idx]] -= 1
child_partition[gen_idx] ← (child_partition[gen_idx] + entero_aleatorio_entre(1,k-1)) mod k
assignations_counter[child_partition[gen_idx]] += 1
end
func_value ← objective_func(X, child_partition, const_list, λ)
counter += 1
children.append([child_partition, func_value, assignations_counter])
end
-Introducimos los hijos generados tras el cruce y la mutación si mejoran a los peores de la población
children.add_list([current_population[-2], current_population[-1]])
ordenar_según_func_objetivo(children)
current_population[-2] ← children[0]
current_population[-1] ← children[1]
ordenar_según_func_objetivo(current_population)
end
return current_population[0][0]

end

3.3. Algoritmos Meméticos

Nuestro algoritmo memético consistirá en hibridar el algoritmo genético generacional que mejor resultado haya dado (que ya adelantamos que es el que utiliza el operador de cruce por segmento fijo, *AGG-SF*) con un nuevo tipo de búsqueda local que denominamos búsqueda local suave (*BLS*). Puesto que ya hemos descrito los algoritmos genéticos generacionales, pasamos a describir la búsqueda local suave.

Algorithm 14: smooth_local_search

Data: Conjunto de datos X, lista de restricciones *const_list*, partición desde la que empieza la búsqueda S, valor de la función objetivo con S, $current\_func\_value$, diccionario con el número de instancias asignadas por S a cada cluster $assignations\_counter$, número de clusters k, número de fallos $max\_failures$, contador de evaluaciones *counter*, hiperparámetro λ

Result: Mejor partición encontrada S, el valor de la función objetivo con S, $best\_func\_value$, su contador de asignaciones $assignations\_counter$ y el contador de evaluaciones actualizado *counter*

```
begin
    n ← longitud(X)
    RSI ← [0,1,...,n-1]
    RSI ← RandomShuffle(RSI)
    failures ← 0; improvement ← True; best_func_value ← current_func_value; i ← 0
    while (improvement or (failures < max_failures)) and i < n do
        improvement ← False
        -Asignar el mejor cluster posible al gen RSI[i] si cambiar este gen no rompe una
         restricción fuerte.
        if assignations_counter[S[RSI[i]]] > 1 then
            best_cluster ← S[RSI[i]]
            foreach j ∈ {0, 1, ..., k − 1} do
                if j ≠ S[RSI[i]] then
                    S[RSI[i]] ← j
                    func_value ← objective_func(X, S, const_list, λ)
                    counter += 1
                    if func_value < best_func_value then
                        assignations_counter[best_cluster] -=1
                        assignations_counter[j] +=1
                        best_func_value ← func_value
                        best_cluster ← j
                        improvement ← True
                    else
                        -Si no es mejor, volvemos a la asignación anterior
                        S[RSI[i]] ← best_cluster
                    end
                end
            end
        end
        if improvement == False then
            failures += 1
        end
        i += 1
    end
    return partition, best_func_value, assignations_counter, counter
end
```

Presentamos también el pseudocódigo principal del memético, que comentaremos más abajo.

Algorithm 15: memetic_algo

Data: Conjunto de datos X, matriz de restricciones *const_matrix*, lista de restricciones *const_list*, número de clusters k, hiperparámetro λ, operador de cruce *cross_operator*, tamaño de la población *population_size*, probabilidad de cruce *cross_prob*, probabilidad de mutación *mutation_prob*, generaciones antes de ejecutar las BLS *generation_per_ls*, porcentaje de la población sobre el que hacer una BLS *perc_ls*, si coger los mejores de la población en este porcentaje o no *best_population*

Result: La mejor partición de la población final

begin

n ← longitud(X)
n_cross_expected ← truncar(cross_prob * population_size / 2)
n_mutations_expected ← truncar(mutation_prob * population_size * n)
n_solutions_for_local_search ← truncar(population_size * perc_ls)
max_failures ← truncar(0.1 * n)
counter ← 0 -Contará el número de evaluaciones de la función objetivo
1-Generamos la población inicial:
current_population, counter ← generate_initial_population(X, const_list, k, λ, population_size, counter)
ordenar_según_func_objetivo(current_population)
while *counter* < *100000* **do**
  foreach *generation* $\in \{0, 1, ..., generation\_per\_ls - 1\}$ **do**
    2-Generamos la población siguiente
    new_population, counter ← new_generation(X, const_list, k, λ, population_size, counter, n_cross_expected, n, cross_operator, n_mutations_expected, current_population)
    ordenar_según_func_objetivo(new_population)
    3-Aplicamos el elitismo
    if *current_population[0][1]* < *new_population[0][1]* **then**
      new_population[population_size-1] ← current_population[0]
      ordenar_según_func_objetivo(new_population)
    end
    current_population ← new_population
  end
  -Aplicamos la búsqueda local suave a la proporción de la población determinada por perc_ls y best_population.
  if *best_population* **then**
    indices_for_ls ← [0,1,...,n_solutions_for_local_search-1]
  else
    indices_for_ls ← RandomSample({0,1,...,population_size-1}, n_solutions_for_local_search)
  end
  foreach $i \in indices\_for\_ls$ **do**
    S, func_value, assignations_counter, counter ← smooth_local_search(X, const_list, current_population[i][0], current_population[i][1], current_population[i][2], k, max_failures, counter, λ)
    current_population[i] ← [S, func_value, assignations_counter]
  end
end
return current_population[0][0]

end

La búsqueda local suave (algoritmo 14) se basará en ir optimizando los genes de la partición (encontrando el mejor cluster para ellos dadas el resto de asignaciones) en orden aleatorio. Cuando en una iteración no se produzcan cambios en el cromosoma diremos que falla. Permitiremos como mucho $max\_failures$ fallos para que no se desperdicien evaluaciones de la función objetivo en cromosomas de mucha calidad. Para el parámetro $max\_failures$ utilizaremos el valor $0{,}1 * numero\_de\_instancias$ como se indica en el guión.

Vemos finalmente cómo se puede combinar esta búsqueda local suave con el algoritmo genético generacional que ya vimos para dar lugar a un algoritmo memético (algoritmo 15). Comprobamos que el algoritmo es bastante similar al algoritmo generacional, pero incorporando un bucle en el que se producen un determinado número de generaciones $generation\_per\_ls$ (como lo haría el algoritmo generacional) y tras este bucle se pasa a usar la búsqueda local suave que acabamos de explicar. Es de señalar que el diseño del algoritmo es de forma que las 3 variantes que se nos piden en la práctica, *AM-(10,1.0)*, *AM-(10,0.1)* y *AM-(10,0.1mej)* se pueden usar simplemente cambiando los argumentos de la función.

En concreto, el parámetro $generation\_per\_ls$ hace referencia al número de generaciones que se producen antes de hacer las búsquedas locales suaves (en las tres variantes es $generation\_per\_ls = 10$). El parámetro $perc\_ls$ se refiere al porcentaje de la población que se utiliza como punto de partida para las búsquedas locales (cada BLS tomará una partición como punto de partida) y toma los valores 1,0, 0,1 y 0,1 respectivamente en cada variante del algoritmo. Y el parámetro $best\_population$ sirve para indicar si para empezar las búsquedas locales se deben tomar los mejores individuos de la población o tomarlos de forma aleatoria (este parámetro solo es $True$ en la última variante). Las probabilidades de cruce y de mutación son las mismas que usamos en los algoritmos generacionales (0,7 y 0,001), pero el tamaño de la población esta vez es de 10. Finalmente, cabe señalar también que el operador de cruce que se ha usado en las tres variantes es el de cruce por segmento fijo, ya que este es el que mejor ha funcionado (aunque no por mucho) en los algoritmos generacionales.

4. Procedimiento de desarrollo

Todo el código, desde la lectura de datos hasta los algoritmos, se ha implementado en *Python* y se encuentra en la carpeta *Software*. En concreto, los algoritmos se encuentran en el fichero *algoritmos.py*. La función objetivo y estadísticos comunes a ambos algoritmos explicados en la sección 2 se encuentra en el fichero *funciones_auxiliares_y_estadisticos.py*. Por otro lado, las funciones dedicadas a la lectura y carga de los conjuntos de datos se encuentran en el fichero *leer_datos.py*.

Finalmente, se han desarrollado dos Jupyter Notebook (en *Python* también). La primera con el objetivo de, usando las funciones definidas en los mencionados ficheros, hacer las ejecuciones que se nos requieren en el guión de una forma clara y sin distracciones y obtener las tablas que se nos pide (así como exportarlas al formato *Excel*). Esta Notebook se llama *Ejecución_y_resultados.ipynb* y para la **replica de los resultados**, se recomienda ejecutarla directamente (las semillas están fijadas en ella). La segunda Notebook, *Análisis_resultados.ipynb*, se ha usado para analizar los resultados y realizar varias visualizaciones; se adjunta parte de ella en las siguientes páginas de esta memoria.

4.1. Entorno y paquetes

Para el desarrollo del proyecto se ha trabajado con Anaconda en Windows 10; en concreto con Python y Jupyter Notebook. Un ordenador con una versión instalada de Python 3 será requerido. Específicamente, se ha usado la version Python 3.7.3. Para instalarla, puedes ejecutar en Anaconda Prompt:

```
conda install python=3.7.3
```

Los paquetes NumPy (1.16.2) y Matplotlib (3.0.3) son usados, los puedes obtener con la herramienta pip:

```
pip3 install numpy
pip3 install matplotlib
```

Los siguientes paquetes son también requeridos para ejecutar todo el código:

- seaborn (0.9.0) para mejorar las visualizaciones.
- pandas (0.24.2) para tratar con los resultados, crear las tablas y trabajar con ellas.

Los puedes instalar con conda:

```
conda install -c anaconda seaborn
conda install -c anaconda pandas
```

5.- Análisis de los resultados ¶

En esta sección describiremos los experimentos realizados y estudiaremos los resultados obtenidos. (Se adjunta también esta parte en formato HTML, donde las tablas y texto se muestran de una forma más elegante).

En el PAR consideraremos 5 ejecuciones con semillas de generación de números aleatorios diferentes para cada algoritmo en cada conjunto de datos con su conjunto de restricciones. Esto quiere decir que un algoritmo cualquiera, se ejecutará 5 veces por cada conjunto de datos y de restricciones, por lo que supondrá un total de $5 * 4 * 2 = 40$ ejecuciones por algoritmo. Las semillas elegidas para cada una de las 5 ejecuciones de los algoritmos, como se puede comprobar en la Notebook `Ejecucion_y_resultados.ipynb` , son `seeds = [0, 14, 17, 25, 31]` . Estas han sido elegidas de forma arbitraria siempre comprobando que la ejecución del algoritmo Greedy (COPKM) no cicla de forma infinita con la inicialización pseudoaleatoria de los centroides dada por cada semilla.

El resultado final de las medidas de validación será calculado como la media de los 5 valores obtenidos para cada conjunto de datos y restricciones. Como se indica en el guión, para facilitar la comparación de algoritmos en las prácticas del PAR se considerarán cuatro estadísticos distintos denominados Tasa_C (la desviación general de la partición solución en las distintas ejecuciones), Tasa_inf (el $infeasibility$), Agregado (la evaluación de la función objetivo) y el tiempo de ejecución T.

Recordamos que la función objetivo venía dada por:

$f = \hat{C} + \lambda * infeasibility$

Así, es importante mencionar que en los siguientes resultados el hyperparámetro λ utilizado ha sido el cociente entre la distancia máxima existente en el conjunto de datos, $D = \max_{x_i, x_j \in X}\{distance(x_i - x_j)\}$, y el número de restricciones del problema $|R|$:

$\lambda = \frac{D}{|R|}$.

In [1]:
```
import os
import pandas as pd
from leer_datos import *
from funciones_auxiliares_y_estadisticos import *
from algoritmos import *
```

In [2]:
```
#Cargamos las tablas
results_folder = os.pardir + "/Results/"
dataframes = np.load(results_folder + 'dataframes_all_algorithms.npy',allow_pickle='TRUE').item()
global_results_dfs = np.load(results_folder + 'dataframes_global_comparison.npy',allow_pickle='TRUE').item()
```

5.1-Resultados obtenidos con cada algoritmo

En primer lugar vamos a mostrar las tablas de ejecución de cada algoritmo y vamos a comentar las diferencias generales que se observan en los resultados sobre los distintos conjuntos de datos y de restricciones (sin entrar aún a comparar los resultados de los distintos algoritmos, lo cual abordaremos en el siguiente apartado). Recordamos que los resultados del *COPKM* y de la Búsqueda Local fueron ya explicados y analizados en la memoria de la práctica 1. No obstante en esta práctica 2 se han ejecutado en el nuevo dataset *Newthyroid* y es por ello que se muestran también aquí.

Resultados obtenidos por el *COPKM* en cada conjunto de datos dado el 10% del total de restricciones:

In [3]:
```
dataframes[("COPKM", 10)]
```

Out[3]:

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**
Ejecución 1	0.173456	35.000000	0.395463	0.180694	201.296981	124.00000	204.623845	3.581947	0.426085	0.0	0.426085	0.167952	60.153157	24.000000	61.030867	0.490785
Ejecución 2	0.148829	0.000000	0.148829	0.231419	222.752358	126.00000	226.132881	10.429566	0.426085	0.0	0.426085	0.178700	69.892593	168.000000	76.036566	0.332393
Ejecución 3	0.243392	83.000000	0.769865	0.167850	232.058982	115.00000	235.144379	14.684767	0.593006	31.0	0.816521	0.167218	71.069135	66.000000	73.482838	0.333951
Ejecución 4	0.148829	0.000000	0.148829	0.167587	240.091561	301.00000	248.167254	8.954754	0.426085	0.0	0.426085	0.167369	62.460336	3.000000	62.570050	0.478212
Ejecución 5	0.148829	0.000000	0.148829	0.163595	234.233805	365.00000	244.026590	9.654126	0.426085	0.0	0.426085	0.166170	62.300166	20.000000	63.031591	0.454946
Media	0.172667	23.600000	0.322363	0.182229	226.086737	206.20000	231.618990	9.461032	0.459469	6.2	0.504172	0.169482	65.175077	56.200000	67.230382	0.418057
Desviación típica	0.036626	32.647205	0.243288	0.025260	13.592848	105.55643	15.485846	3.551977	0.066768	12.4	0.156175	0.004645	4.423788	59.620131	6.235778	0.070258

Resultados obtenidos por el *COPKM* en cada conjunto de datos dado el 20% del total de restricciones:

In [4]:
```
dataframes[("COPKM", 20)]
```

Out[4]:

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**
Ejecución 1	0.144022	21.000000	0.210594	0.174871	240.088037	90.000000	241.295366	3.699347	0.426085	0.0	0.426085	0.172562	62.659821	0.000000	62.659821	0.340398
Ejecución 2	0.148829	0.000000	0.148829	0.189957	235.489955	250.000000	238.843648	9.898984	0.426085	0.0	0.426085	0.123466	68.470538	352.000000	74.905682	0.452830
Ejecución 3	0.186499	60.000000	0.376706	0.297462	177.463791	59.000000	178.255263	2.910767	0.420902	18.0	0.485765	0.116006	62.659821	0.000000	62.659821	0.340583
Ejecución 4	0.148829	0.000000	0.148829	0.175524	249.287102	422.000000	254.948136	15.851435	0.426085	0.0	0.426085	0.172000	62.100914	91.000000	63.764545	0.342575
Ejecución 5	0.148829	0.000000	0.148829	0.171347	197.459443	78.000000	198.505795	3.559698	0.426085	0.0	0.426085	0.172146	62.659821	0.000000	62.659821	0.342801
Media	0.155402	16.200000	0.206757	0.201832	219.957665	179.800000	222.369642	7.184046	0.425048	3.6	0.438021	0.151236	63.710183	88.600000	65.329938	0.363837
Desviación típica	0.015660	23.361507	0.088277	0.048238	27.635481	139.002734	29.005158	5.020647	0.002073	7.2	0.023872	0.025828	2.390000	136.334295	4.806951	0.044507

Resultados obtenidos por la Búsqueda Local en cada conjunto de datos dado el 10% del total de restricciones:

In [5]:
```
dataframes[("BL", 10)]
```

Out[5]:

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**
Ejecución 1	0.173064	11.000000	0.242838	3.549502	90.488749	1076.000000	119.357342	164.824874	0.567493	17.000000	0.690066	5.421015	37.825051	263.000000	47.443294	15.291635
Ejecución 2	0.147961	10.000000	0.211392	5.472892	88.731481	870.000000	112.073186	246.646611	0.471133	9.000000	0.536025	4.206658	28.058665	533.000000	47.551150	18.193446
Ejecución 3	0.164955	14.000000	0.253758	4.246338	82.376373	882.000000	106.040033	185.752572	0.488334	5.000000	0.524385	4.489495	39.027391	257.000000	48.426206	15.041174
Ejecución 4	0.148829	0.000000	0.148829	4.815099	88.213647	653.000000	105.733341	194.677624	0.426085	0.000000	0.426085	4.098372	36.547584	299.000000	47.482392	20.048154
Ejecución 5	0.148829	0.000000	0.148829	6.975041	73.927553	1154.000000	104.888850	231.527741	0.481247	7.000000	0.531718	4.364523	38.199159	273.000000	48.183115	18.933448
Media	0.156728	7.000000	0.201129	5.011774	84.747561	927.000000	109.618550	204.685884	0.486858	7.600000	0.541656	4.516013	35.931570	325.000000	47.817231	17.501571
Desviación típica	0.010356	5.865151	0.044912	1.168748	6.058741	175.544866	5.498000	30.095593	0.045774	5.571355	0.084630	0.471764	4.016801	104.987618	0.406809	1.997570

Resultados obtenidos por la Búsqueda Local en cada conjunto de datos dado el 20% del total de restricciones:

```
In [6]: dataframes[("BL", 20)]
```

Out[6]:

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Ejecución 1	0.164327	13.000000	0.205538	7.089431	85.653598	1799.000000	109.786775	493.439849	0.476127	44.0	0.634680	10.438419	29.580781	978.000000	47.460243	39.662784
Ejecución 2	0.164935	19.000000	0.225167	7.804216	93.670943	1459.000000	113.243097	344.433477	0.546218	34.0	0.668736	7.594651	25.598305	1071.000000	45.177961	28.515394
Ejecución 3	0.172664	58.000000	0.356531	6.142935	87.998345	1733.000000	111.246147	328.455965	0.426085	0.0	0.426085	8.187196	25.598305	1071.000000	45.177961	30.195772
Ejecución 4	0.148829	0.000000	0.148829	7.712220	81.396033	2436.000000	114.074420	352.515751	0.536014	45.0	0.698171	6.599174	37.757858	509.000000	47.063222	41.513329
Ejecución 5	0.181723	75.000000	0.419481	7.238607	93.999634	1456.000000	113.531544	467.763391	0.619710	54.0	0.814298	7.056082	32.527740	674.000000	44.849578	50.074861
Media	0.166496	33.000000	0.271109	7.197482	88.543710	1776.600000	112.376397	397.321687	0.520831	35.4	0.648394	7.975105	30.212598	860.600000	45.945793	37.992428
Desviación típica	0.010855	28.544702	0.100672	0.593046	4.812382	358.012625	1.609498	68.917279	0.065768	18.8	0.126511	1.341300	4.588539	228.368649	1.088396	7.896506

Resultados obtenidos por el *AGG-UN* en cada conjunto de datos dado el 10% del total de restricciones:

```
In [7]: dataframes[("AGG-UN", 10)]
```

Out[7]:

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Ejecución 1	1.488290e-01	0.0	1.488290e-01	368.221720	95.681475	738.000000	115.481680	1920.986287	0.373341	0.0	0.373341	239.734939	25.598305	561.000000	46.114785	389.665242
Ejecución 2	1.488290e-01	0.0	1.488290e-01	191.557114	79.600594	979.000000	105.866720	899.044149	0.373341	0.0	0.373341	187.600484	37.903458	237.000000	46.570848	359.716703
Ejecución 3	1.488290e-01	0.0	1.488290e-01	187.904033	93.882430	887.000000	117.680238	904.361158	0.373341	0.0	0.373341	192.004951	41.831299	259.000000	51.303256	360.866222
Ejecución 4	1.488290e-01	0.0	1.488290e-01	188.437708	88.505484	1399.000000	126.040020	902.877389	0.373341	0.0	0.373341	188.832887	32.172118	370.000000	45.703486	360.789678
Ejecución 5	1.488290e-01	0.0	1.488290e-01	194.121990	107.149917	1122.000000	137.252668	894.495535	0.373341	0.0	0.373341	191.100880	29.166502	512.000000	47.890990	363.095474
Media	1.488290e-01	0.0	1.488290e-01	226.048513	92.963980	1025.000000	120.464265	1104.352903	0.373341	0.0	0.373341	199.854828	33.334336	387.800000	47.516673	366.826664
Desviación típica	2.482534e-17	0.0	2.482534e-17	71.122084	9.032468	224.861735	10.569814	408.330984	0.000000	0.0	0.000000	20.001638	5.859062	130.438338	2.031051	11.472106

Resultados obtenidos por el *AGG-UN* en cada conjunto de datos dado el 20% del total de restricciones:

```
In [8]: dataframes[("AGG-UN", 20)]
```

Out[8]:

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Ejecución 1	1.488290e-01	0.0	1.488290e-01	690.476907	90.242402	1621.000000	111.987750	2770.253531	0.373341	0.0	0.373341	377.651813	27.470930	899.000000	43.906141	721.756554
Ejecución 2	1.488290e-01	0.0	1.488290e-01	350.986329	83.493980	2079.000000	111.383294	1709.001997	0.373341	0.0	0.373341	350.750173	27.872538	960.000000	45.422930	688.600414
Ejecución 3	1.488290e-01	0.0	1.488290e-01	347.371616	86.582618	3121.000000	128.450126	1731.908041	0.373341	0.0	0.373341	349.399563	29.035613	1004.000000	47.390399	691.928941
Ejecución 4	1.488290e-01	0.0	1.488290e-01	349.254743	94.762281	1871.000000	119.861322	1718.220066	0.373341	0.0	0.373341	349.814923	25.598305	1071.000000	45.177961	685.812867
Ejecución 5	1.488290e-01	0.0	1.488290e-01	354.787134	83.943960	1562.000000	104.897836	1721.569770	0.373341	0.0	0.373341	359.357170	26.399888	1058.000000	45.741883	684.117175
Media	1.488290e-01	0.0	1.488290e-01	418.575346	87.805048	2050.800000	115.316065	1930.190681	0.373341	0.0	0.373341	357.394728	27.275455	998.400000	45.527863	694.443190
Desviación típica	2.149938e-17	0.0	2.149938e-17	135.972782	4.226853	566.057382	8.102429	420.095288	0.000000	0.0	0.000000	10.767793	1.189313	63.575467	1.120559	13.911063

Resultados obtenidos por el *AGG-SF* en cada conjunto de datos dado el 10% del total de restricciones:

```
In [9]: dataframes[("AGG-SF", 10)]
```

Out[9]:

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Ejecución 1	1.488290e-01	0.0	1.488290e-01	188.964792	83.927834	904.000000	108.181744	899.774861	0.373341	0.0	0.373341	190.240936	29.522653	510.000000	48.173999	365.477478
Ejecución 2	1.488290e-01	0.0	1.488290e-01	189.837864	81.801735	1153.000000	112.736202	893.823796	0.373341	0.0	0.373341	186.040029	23.838359	632.000000	46.951399	356.893641
Ejecución 3	1.488290e-01	0.0	1.488290e-01	188.391563	102.602950	935.000000	127.688577	894.215680	0.373341	0.0	0.373341	190.045812	39.769372	175.000000	46.169344	362.864788
Ejecución 4	1.488290e-01	0.0	1.488290e-01	186.218229	83.301533	1132.000000	113.672580	900.127414	0.373341	0.0	0.373341	187.737956	23.838359	632.000000	46.951399	358.707797
Ejecución 5	1.488290e-01	0.0	1.488290e-01	188.391137	84.968744	1117.000000	114.937348	909.743726	0.373341	0.0	0.373341	186.731671	37.903458	237.000000	46.570848	355.909649
Media	1.488290e-01	0.0	1.488290e-01	188.360717	87.320559	1048.200000	115.443290	899.537095	0.373341	0.0	0.373341	188.159281	30.974440	437.200000	46.963398	359.970671
Desviación típica	1.241267e-17	0.0	1.241267e-17	1.194906	7.709774	106.157242	6.532944	5.753948	0.000000	0.0	0.000000	1.708748	6.772254	194.947583	0.670889	3.640211

Resultados obtenidos por el *AGG-SF* en cada conjunto de datos dado el 20% del total de restricciones:

```
In [10]: dataframes[("AGG-SF", 20)]
```

Out[10]:

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Ejecución 1	1.488290e-01	0.0	1.488290e-01	351.988621	92.081937	1782.000000	115.987063	1740.557315	0.373341	0.0	0.373341	358.167816	25.598305	1071.0	45.177961	703.518446
Ejecución 2	1.488290e-01	0.0	1.488290e-01	350.538014	88.528944	1802.000000	112.702366	1711.305833	0.373341	0.0	0.373341	348.670845	25.598305	1071.0	45.177961	681.334973
Ejecución 3	1.488290e-01	0.0	1.488290e-01	346.851720	92.116829	2578.000000	126.700115	1716.384746	0.373341	0.0	0.373341	356.009712	27.470930	899.0	43.906141	691.721778
Ejecución 4	1.488290e-01	0.0	1.488290e-01	346.509919	78.227647	3354.000000	123.220797	1712.519955	0.373341	0.0	0.373341	349.537122	25.598305	1071.0	45.177961	692.487798
Ejecución 5	1.488290e-01	0.0	1.488290e-01	347.318666	95.446842	1548.000000	116.212911	1727.815382	0.373341	0.0	0.373341	345.928407	25.598305	1071.0	45.177961	685.454579
Media	1.488290e-01	0.0	1.488290e-01	348.641388	89.280440	2212.800000	118.964650	1721.716646	0.373341	0.0	0.373341	351.662780	25.972830	1036.6	44.923597	690.903515
Desviación típica	2.149938e-17	0.0	2.149938e-17	2.204397	5.943853	668.133639	5.169265	11.077533	0.000000	0.0	0.000000	4.638199	0.749050	68.8	0.508728	7.530789

Resultados obtenidos por el *AGE-UN* en cada conjunto de datos dado el 10% del total de restricciones:

```
In [11]: dataframes[("AGE-UN", 10)]
```

Out[11]:

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Ejecución 1	0.148829	0.0	0.148829	194.253230	85.494059	549.000000	100.223480	921.175905	0.373341	0.0	0.373341	193.031662	29.200013	511.000000	47.887930	363.747326
Ejecución 2	0.148829	0.0	0.148829	194.153367	81.735917	769.000000	102.367839	924.076518	0.373341	0.0	0.373341	191.581295	29.720269	510.000000	48.371615	366.158458
Ejecución 3	0.148829	0.0	0.148829	194.030646	82.749982	734.000000	102.442869	918.247310	0.373341	0.0	0.373341	195.801747	23.838359	632.000000	46.951399	369.007670
Ejecución 4	0.148829	0.0	0.148829	197.349090	83.116331	701.000000	101.923844	926.766694	0.373341	0.0	0.373341	193.039356	26.403805	586.000000	47.834566	366.643372
Ejecución 5	0.148829	0.0	0.148829	191.659702	85.181838	579.000000	100.716146	989.051303	0.373341	0.0	0.373341	191.884557	25.598305	561.000000	46.114785	372.026588
Media	0.148829	0.0	0.148829	194.289207	83.655625	666.400000	101.534835	935.863546	0.373341	0.0	0.373341	193.067723	26.952150	560.000000	47.432059	367.516683
Desviación típica	0.000000	0.0	0.000000	1.809076	1.449515	86.850676	0.901072	26.745777	0.000000	0.0	0.000000	1.489050	2.215610	46.393965	0.802352	2.806342

Resultados obtenidos por el *AGE-UN* en cada conjunto de datos dado el 20% del total de restricciones:

```
In [12]: dataframes[("AGE-UN", 20)]
```

Out[12]:

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Ejecución 1	1.488290e-01	0.0	1.488290e-01	354.971527	84.081494	1301.000000	101.534115	1792.666803	0.373341	0.0	0.373341	351.279433	39.039312	412.000000	46.571355	701.460048
Ejecución 2	1.488290e-01	0.0	1.488290e-01	353.728319	81.171184	1572.000000	102.259207	1734.233617	0.373341	0.0	0.373341	355.650770	29.033792	960.000000	46.584184	703.031201
Ejecución 3	1.488290e-01	0.0	1.488290e-01	354.980730	81.730512	1485.000000	101.651450	1740.463700	0.373341	0.0	0.373341	355.295596	25.598305	1071.000000	45.177961	692.432237
Ejecución 4	1.488290e-01	0.0	1.488290e-01	356.810838	81.880396	1548.000000	102.646465	1748.958522	0.373341	0.0	0.373341	353.483880	28.908501	969.000000	46.623428	694.566981
Ejecución 5	1.488290e-01	0.0	1.488290e-01	362.056962	83.387782	1396.000000	102.114806	1747.592271	0.373341	0.0	0.373341	361.350157	28.962662	994.000000	47.134630	715.519107
Media	1.488290e-01	0.0	1.488290e-01	356.509675	82.450274	1460.400000	102.041208	1752.782983	0.373341	0.0	0.373341	355.411967	30.308514	881.200000	46.418312	701.401915
Desviación típica	1.241267e-17	0.0	1.241267e-17	2.942876	1.097150	100.288783	0.407021	20.630719	0.000000	0.0	0.000000	3.350463	4.556516	237.829687	0.654918	8.113255

Resultados obtenidos por el *AGE-SF* en cada conjunto de datos dado el 10% del total de restricciones:

```
In [13]: dataframes[("AGE-SF", 10)]
```

Out[13]:

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Ejecución 1	1.488290e-01	0.0	1.488290e-01	191.509849	88.024779	535.000000	102.378586	924.379448	0.373341	0.0	0.373341	191.838236	23.838359	632.000000	46.951399	363.130526
Ejecución 2	1.488290e-01	0.0	1.488290e-01	190.945700	83.809584	675.000000	101.919528	913.625603	0.373341	0.0	0.373341	193.242640	28.206019	533.000000	47.698504	369.415917
Ejecución 3	1.488290e-01	0.0	1.488290e-01	190.802045	83.783921	732.000000	103.423150	905.593943	0.373341	0.0	0.373341	193.569905	37.903458	237.000000	46.570848	366.609698
Ejecución 4	1.488290e-01	0.0	1.488290e-01	190.723107	82.721566	670.000000	100.697363	907.922811	0.373341	0.0	0.373341	189.996464	29.374847	514.000000	48.172477	363.352646
Ejecución 5	1.488290e-01	0.0	1.488290e-01	195.028250	81.354784	757.000000	101.664751	923.484046	0.373341	0.0	0.373341	199.022630	35.990254	421.000000	51.386756	373.326101
Media	1.488290e-01	0.0	1.488290e-01	191.801790	83.938927	673.800000	102.016675	915.001170	0.373341	0.0	0.373341	193.533975	31.062587	467.400000	48.155997	367.166978
Desviación típica	1.241267e-17	0.0	1.241267e-17	1.636534	2.231354	76.929578	0.892527	7.751162	0.000000	0.0	0.000000	3.019967	5.182226	133.270552	1.709452	3.850848

Resultados obtenidos por el *AGE-SF* en cada conjunto de datos dado el 20% del total de restricciones:

```
In [14]: dataframes[("AGE-SF", 20)]
```

Out[14]:

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Ejecución 1	0.148829	0.0	0.148829	352.846310	80.736284	1463.000000	100.362097	1732.549407	0.373341	0.0	0.373341	351.899203	29.052022	998.000000	47.297117	689.593273
Ejecución 2	0.148829	0.0	0.148829	353.425468	84.601977	1291.000000	101.920449	1753.995510	0.373341	0.0	0.373341	362.738422	29.033792	960.000000	46.584184	701.470954
Ejecución 3	0.148829	0.0	0.148829	355.133017	80.631542	1587.000000	101.920787	1741.269800	0.373341	0.0	0.373341	358.252931	29.037118	959.000000	46.569229	688.095022
Ejecución 4	0.148829	0.0	0.148829	351.685254	83.719938	1540.000000	104.378689	1746.592412	0.373341	0.0	0.373341	352.725382	29.061652	1001.000000	47.361592	703.595870
Ejecución 5	0.148829	0.0	0.148829	352.343213	82.305563	1395.000000	101.019172	1727.486413	0.373341	0.0	0.373341	355.162740	29.277023	986.000000	47.302739	694.392668
Media	0.148829	0.0	0.148829	353.086653	82.399061	1455.200000	101.920239	1740.378708	0.373341	0.0	0.373341	356.155736	29.092321	980.800000	47.022972	695.429557
Desviación típica	0.000000	0.0	0.000000	1.172756	1.580838	105.079779	1.362623	9.511369	0.000000	0.0	0.000000	3.964882	0.092901	18.104143	0.365104	6.198708

Resultados obtenidos por el *AM-(10,1.0)* en cada conjunto de datos dado el 10% del total de restricciones:

```
In [15]: dataframes[("AM-(10,1.0)", 10)]
```

Out[15]:

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Ejecución 1	1.488290e-01	0.0	1.488290e-01	173.694771	90.093844	964.000000	115.957527	850.545040	0.373341	0.0	0.373341	176.770859	27.262694	590.000000	48.839741	330.925500
Ejecución 2	1.488290e-01	0.0	1.488290e-01	177.211009	98.172030	966.000000	124.089373	824.227974	0.373341	0.0	0.373341	174.721794	26.367764	530.000000	45.750534	330.804403
Ejecución 3	1.488290e-01	0.0	1.488290e-01	176.502720	76.383706	1145.000000	107.103537	832.523861	0.373341	0.0	0.373341	172.306982	39.769372	175.000000	46.169344	329.374828
Ejecución 4	1.488290e-01	0.0	1.488290e-01	176.978649	94.509949	869.000000	117.824825	827.407357	0.373341	0.0	0.373341	176.461468	25.135528	597.000000	46.968574	332.186067
Ejecución 5	1.488290e-01	0.0	1.488290e-01	177.749722	90.885633	669.000000	108.834599	833.118399	0.373341	0.0	0.373341	173.574085	27.726573	572.000000	48.645337	333.274353
Media	1.488290e-01	0.0	1.488290e-01	176.427374	90.009033	922.600000	114.761972	833.564526	0.373341	0.0	0.373341	174.767038	29.252386	492.800000	47.274706	331.313030
Desviación típica	1.241267e-17	0.0	1.241267e-17	1.424036	7.393483	155.077529	6.190254	9.107160	0.000000	0.0	0.000000	1.694901	5.332233	160.598132	1.262268	1.324726

Resultados obtenidos por el *AM-(10,1.0)* en cada conjunto de datos dado el 20% del total de restricciones:

```
In [16]: dataframes[("AM-(10,1.0)", 20)]
```

Out[16]:

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Ejecución 1	1.488290e-01	0.0	1.488290e-01	327.926709	90.341831	1586.000000	111.617662	1627.026761	0.373341	0.0	0.373341	329.923150	25.598305	1071.000000	45.177961	642.226627
Ejecución 2	1.488290e-01	0.0	1.488290e-01	328.160105	77.694843	2566.000000	112.117151	1596.257371	0.373341	0.0	0.373341	338.734571	29.041046	956.000000	46.518312	637.327470
Ejecución 3	1.488290e-01	0.0	1.488290e-01	331.262856	92.139573	1337.000000	110.075125	1601.717585	0.373341	0.0	0.373341	338.939124	25.598305	1071.000000	45.177961	647.606889
Ejecución 4	1.488290e-01	0.0	1.488290e-01	335.263660	80.837186	1532.000000	101.388619	1575.123439	0.373341	0.0	0.373341	332.091858	25.598305	1071.000000	45.177961	634.531177
Ejecución 5	1.488290e-01	0.0	1.488290e-01	329.345548	89.600236	1502.000000	109.749225	1621.438581	0.373341	0.0	0.373341	332.093503	29.041046	956.000000	46.518312	637.931284
Media	1.488290e-01	0.0	1.488290e-01	330.391775	86.122734	1704.600000	108.989556	1604.312747	0.373341	0.0	0.373341	334.356441	26.975401	1025.000000	45.714101	639.924689
Desviación típica	2.149938e-17	0.0	2.149938e-17	2.707232	5.745651	438.644093	3.904320	18.616446	0.000000	0.0	0.000000	3.743589	1.686592	56.338264	0.656635	4.563426

Resultados obtenidos por el *AM-(10,0.1)* en cada conjunto de datos dado el 10% del total de restricciones:

```
In [17]: dataframes[("AM-(10,0.1)", 10)]
```

Out[17]:

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Ejecución 1	1.488290e-01	0.0	1.488290e-01	182.521583	84.157523	554.000000	99.021092	805.625519	0.373341	0.0	0.373341	180.452538	29.522653	510.000000	48.173999	335.870472
Ejecución 2	1.488290e-01	0.0	1.488290e-01	180.654484	80.067505	822.000000	102.121392	833.675609	0.373341	0.0	0.373341	183.840248	26.367764	530.000000	45.750534	343.027596
Ejecución 3	1.488290e-01	0.0	1.488290e-01	184.107554	84.585163	583.000000	100.226789	818.292506	0.373341	0.0	0.373341	181.554672	28.908501	503.000000	47.303848	350.697486
Ejecución 4	1.488290e-01	0.0	1.488290e-01	184.116049	84.926768	517.000000	98.797643	809.177880	0.373341	0.0	0.373341	184.127473	28.206019	533.000000	47.698504	342.900891
Ejecución 5	1.488290e-01	0.0	1.488290e-01	182.323766	82.315860	667.000000	100.211168	813.208892	0.373341	0.0	0.373341	184.607825	25.598305	561.000000	46.114785	346.045417
Media	1.488290e-01	0.0	1.488290e-01	182.744687	83.210564	628.600000	100.075617	815.996081	0.373341	0.0	0.373341	182.916551	27.720648	527.400000	47.008334	343.708373
Desviación típica	1.755417e-17	0.0	1.755417e-17	1.291042	1.811680	108.606814	1.180819	9.794326	0.000000	0.0	0.000000	1.619014	1.498562	20.323386	0.927677	4.834411

Resultados obtenidos por el *AM-(10,0.1)* en cada conjunto de datos dado el 20% del total de restricciones:

In [18]:
```
dataframes[("AM-(10,0.1)", 20)]
```

Out[18]:

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Ejecución 1	1.488290e-01	0.0	1.488290e-01	337.812359	82.261027	1537.000000	102.879534	1565.331516	0.373341	0.0	0.373341	336.508628	29.041046	956.000000	46.518312	658.146917
Ejecución 2	1.488290e-01	0.0	1.488290e-01	344.722016	82.176692	1522.000000	102.593977	1543.905137	0.373341	0.0	0.373341	343.299946	29.054179	999.000000	47.317556	665.390290
Ejecución 3	1.488290e-01	0.0	1.488290e-01	342.693966	82.991255	1320.000000	100.698755	1584.555410	0.373341	0.0	0.373341	343.065687	29.041046	956.000000	46.518312	656.435438
Ejecución 4	1.488290e-01	0.0	1.488290e-01	338.074493	82.008226	1351.000000	100.131584	1557.714017	0.373341	0.0	0.373341	344.641308	27.470930	899.000000	43.906141	663.841217
Ejecución 5	1.488290e-01	0.0	1.488290e-01	334.169992	81.579681	1585.000000	102.842097	1574.882398	0.373341	0.0	0.373341	335.293095	28.908501	969.000000	46.623428	655.750338
Media	1.488290e-01	0.0	1.488290e-01	339.494565	82.203376	1463.000000	101.829190	1565.277695	0.373341	0.0	0.373341	340.561733	28.703140	955.800000	46.176750	659.912840
Desviación típica	1.755417e-17	0.0	1.755417e-17	3.761912	0.458665	106.615196	1.172506	13.984842	0.000000	0.0	0.000000	3.862557	0.618402	32.455508	1.173878	3.948971

Resultados obtenidos por el *AM-(10,0.1mej)* en cada conjunto de datos dado el 10% del total de restricciones:

In [19]:
```
dataframes[("AM-(10,0.1mej)", 10)]
```

Out[19]:

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Ejecución 1	0.148829	0.0	0.148829	182.033957	74.970955	1185.000000	106.763968	810.322885	0.373341	0.0	0.373341	178.992598	30.186487	463.000000	47.118983	345.908607
Ejecución 2	0.148829	0.0	0.148829	183.808253	88.457620	660.000000	106.165121	798.876569	0.373341	0.0	0.373341	183.642561	29.699268	511.000000	48.387185	336.178916
Ejecución 3	0.148829	0.0	0.148829	184.912770	85.288162	536.000000	99.668799	809.332591	0.373341	0.0	0.373341	179.724969	41.631712	198.000000	48.872823	342.873996
Ejecución 4	0.148829	0.0	0.148829	179.674490	84.315363	541.000000	98.830148	803.695038	0.373341	0.0	0.373341	181.432644	30.511394	538.000000	50.186735	347.917431
Ejecución 5	0.148829	0.0	0.148829	179.176405	83.675590	570.000000	98.968431	820.513749	0.373341	0.0	0.373341	178.500293	30.186487	463.000000	47.118983	341.971061
Media	0.148829	0.0	0.148829	181.921175	83.341538	698.400000	102.079293	808.548167	0.373341	0.0	0.373341	180.458613	32.443070	434.600000	48.336942	342.970002
Desviación típica	0.000000	0.0	0.000000	2.240749	4.496448	247.341545	3.596806	7.266812	0.000000	0.0	0.000000	1.876631	4.601637	121.754836	1.155706	4.005956

Resultados obtenidos por el *AM-(10,0.1mej)* en cada conjunto de datos dado el 20% del total de restricciones:

In [20]:
```
dataframes[("AM-(10,0.1mej)", 20)]
```

Out[20]:

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Ejecución 1	1.488290e-01	0.0	1.488290e-01	337.957959	83.782326	1379.000000	102.281298	1567.703778	0.373341	0.0	0.373341	342.329321	28.995555	1015.000000	47.551438	641.537130
Ejecución 2	1.488290e-01	0.0	1.488290e-01	339.296181	82.994613	1290.000000	100.299670	1543.024345	0.373341	0.0	0.373341	333.146268	39.039312	412.000000	46.571355	643.524877
Ejecución 3	1.488290e-01	0.0	1.488290e-01	350.665536	80.826536	1602.000000	102.317003	1564.718781	0.373341	0.0	0.373341	336.132345	29.052791	996.000000	47.261323	655.575169
Ejecución 4	1.488290e-01	0.0	1.488290e-01	338.905004	82.993932	1301.000000	100.446552	1551.473278	0.373341	0.0	0.373341	338.053059	25.598305	1071.000000	45.177961	657.818083
Ejecución 5	1.488290e-01	0.0	1.488290e-01	343.810929	81.208215	1449.000000	100.646222	1545.088347	0.373341	0.0	0.373341	338.058976	25.598305	1071.000000	45.177961	650.519835
Media	1.488290e-01	0.0	1.488290e-01	342.127122	82.361124	1404.200000	101.198149	1554.401706	0.373341	0.0	0.373341	337.543994	29.656853	913.000000	46.348008	649.795019
Desviación típica	1.241267e-17	0.0	1.241267e-17	4.722667	1.140678	114.419229	0.905741	10.081129	0.000000	0.0	0.000000	2.991871	4.935106	252.278418	1.007005	6.415800

Análisis

En primer lugar, observamos (en las tablas de todos los algoritmos) que los peores resultados medios en los 4 estadísticos se obtienen con el conjunto de datos `Ecoli`. Para explicar este fenómeno es importante recordar los tamaños de cada conjunto de datos:

- Ecoli: 336 instancias con 7 características. Tiene 8 clases ($k = 8$).
- Iris: 150 instancias con 4 características. Tiene 3 clases ($k = 3$).
- Rand: 150 instancias con 2 características. Tiene 3 clases ($k = 3$).
- Newthyroid: 215 instancias con 5 características. Tiene 3 clases ($k = 3$).

El mayor tamaño del dataset Ecoli, unido con el mayor número de clusters ($k = 8$), parece ser una explicación clara para que sus tiempos de ejecución sean bastante mayores y sus desviaciones generales (Tasa_C) peores. Además, un mayor número de instancias implicará que el 10% (o 20%) del total de restricciones en el dataset Ecoli son muchas más que el 10% (o 20%) del total de restricciones de los otros datasets (pues estos tienen un menor número de instancias y por tanto el total de restricciones será menor). Este hecho parece ser la razón por la que la $infeasibility$ (Tasa_inf) es bastante mayor con el dataset `Ecoli`. Además, vemos que los segundos peores resultados los encontramos en el nuevo dataset `Newthyroid`, lo cual refuerza esta explicación, pues se trata del segundo conjunto de datos más grande; los 4 estadísticos resultan ser peores en este dataset que en los dos más pequeños. Los datasets `Iris y Rand` parecen ser más parecidos en tamaño y vemos que sus resultados son más similares.

Finalmente, estas tablas (dos por algoritmo) nos permiten ver también la diferencia de los resultados cuando usamos el 10% de las restricciones y cuando usamos el 20% de estas. Vemos que con todos los datasets las diferencias en los resultados no difieren demasiado (sobre todo en los conjuntos de datos pequeños, donde los algoritmos funcionan bien con ambos conjuntos). No obstante, en los datasets *Ecoli* y *Newthyroid* si parece haber una pequeña mejora cuando usamos el 20% de las restricciones en los estadísticos (excepto en el tiempo). Cuando le damos más restricciones a nuestro algoritmo lo que estamos haciendo es darle parte de la solución, una información que puede ser clave para formar los clusters correctos. Esto podría explicar la mejora en la Tasa_C y la Tasa_inf (y por ende en la función objetivo (Agr.)).

5.2- Comparación de todos los algoritmos

Resultados obtenidos en cada conjunto de datos dado el 10% del total de restricciones:

In [21]:
```
global_results_dfs[10]
```

Out[21]:

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
COPKM	0.172667	23.6	0.322363	0.182229	226.086737	206.2	231.618990	9.461032	0.459469	6.2	0.504172	0.169482	65.175077	56.2	67.230382	0.418057
BL	0.156728	7.0	0.201129	5.011774	84.747561	927.0	109.618550	204.685884	0.486858	7.6	0.541656	4.516013	35.931570	325.0	47.817231	17.501571
AGG-UN	0.148829	0.0	0.148829	226.048513	92.963980	1025.0	120.464265	1104.352903	0.373341	0.0	0.373341	199.854828	33.334336	387.8	47.516673	366.826664
AGG-SF	0.148829	0.0	0.148829	188.360717	87.320559	1048.2	115.443290	899.537095	0.373341	0.0	0.373341	188.159281	30.974440	437.2	46.963398	359.970671
AGE-UN	0.148829	0.0	0.148829	194.289207	83.655625	666.4	101.534835	935.863546	0.373341	0.0	0.373341	193.067723	26.952150	560.0	47.432059	367.516683
AGE-SF	0.148829	0.0	0.148829	191.801790	83.938927	673.8	102.016675	915.001170	0.373341	0.0	0.373341	193.533975	31.062587	467.4	48.155997	367.166978
AM-(10,1.0)	0.148829	0.0	0.148829	176.427374	90.009033	922.6	114.761972	833.564526	0.373341	0.0	0.373341	174.767038	29.252386	492.8	47.274706	331.313030
AM-(10,0.1)	0.148829	0.0	0.148829	182.744687	83.210564	628.6	100.075617	815.996081	0.373341	0.0	0.373341	182.916551	27.720648	527.4	47.008334	343.708373
AM-(10,0.1mej)	0.148829	0.0	0.148829	181.921175	83.341538	698.4	102.079293	808.548167	0.373341	0.0	0.373341	180.458613	32.443070	434.6	48.336942	342.970002

Resultados obtenidos en cada conjunto de datos dado el 20% del total de restricciones:

In [22]:
```
global_results_dfs[20]
```

Out[22]:

	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
COPKM	0.155402	16.2	0.206757	0.201832	219.957665	179.8	222.369642	7.184046	0.425048	3.6	0.438021	0.151236	63.710183	88.6	65.329938	0.363837
BL	0.166496	33.0	0.271109	7.197482	88.543710	1776.6	112.376397	397.321687	0.520831	35.4	0.648394	7.975105	30.212598	860.6	45.945793	37.992428
AGG-UN	0.148829	0.0	0.148829	418.575346	87.805048	2050.8	115.316065	1930.190681	0.373341	0.0	0.373341	357.394728	27.275455	998.4	45.527863	694.443190
AGG-SF	0.148829	0.0	0.148829	348.641388	89.280440	2212.8	118.964650	1721.716646	0.373341	0.0	0.373341	351.662780	25.972830	1036.6	44.923597	690.903515
AGE-UN	0.148829	0.0	0.148829	356.509675	82.450274	1460.4	102.041208	1752.782983	0.373341	0.0	0.373341	355.411967	30.308514	881.2	46.418312	701.401915
AGE-SF	0.148829	0.0	0.148829	353.086653	82.399061	1455.2	101.920239	1740.378708	0.373341	0.0	0.373341	356.155736	29.092321	980.8	47.022972	695.429557
AM-(10,1.0)	0.148829	0.0	0.148829	330.391775	86.122734	1704.6	108.989556	1604.312747	0.373341	0.0	0.373341	334.356441	26.975401	1025.0	45.714101	639.924689
AM-(10,0.1)	0.148829	0.0	0.148829	339.494565	82.203376	1463.0	101.829190	1565.277695	0.373341	0.0	0.373341	340.561733	28.703140	955.8	46.176750	659.912840
AM-(10,0.1mej)	0.148829	0.0	0.148829	342.127122	82.361124	1404.2	101.198149	1554.401706	0.373341	0.0	0.373341	337.543994	29.656853	913.0	46.348008	649.795019

Análisis

En primer lugar cabe recordar que todos los algoritmos excepto el Greedy (*COPKM*) se basan en minimizar la función objetivo $f = \hat{C} + \lambda * infeasibility$ (el Greedy también trabaja con la desviación general y la infactibilidad, pero lo hace en etapas separadas del algoritmo y no trabaja explicitamente con esta función). Así, el hiperparámetro λ no solo juega su papel a la hora de calcular el Agregado (Agr.) que vemos en las tablas, sino que también juega un papel crucial durante la ejecución de los algoritmos. Un λ alto dará mayor importancia a minimizar la $infeasibility$, mientras que un λ bajo hará que el algoritmo se centre en minimizar la desviación general (Tasa_C). Como se comentó al principio de esta sección, en estos resultado se ha usado $\lambda = \frac{D}{|R|}$. Esto es importante notarlo a la hora de comparar los resutados obtenidos por el algoritmo Greedy con los resultados obtenidos por el resto de algoritmos.

Vemos que en los conjuntos Ecoli y Newthyroid el *COPKM* es el algoritmo que consigue con diferencias notables la mejor Tasa_inf y sin embargo es también el algoritmo que consigue notablemente los peores Agregados (Agr. más altos). Así, se podría decir que los algoritmos heurísticos (el resto de algoritmos) cumplen lo que cabría esperar de ellos y minimizan la función objetivo (el Agr.) mejor que el Greedy. Además, esto no significa necesariamente que el *COPKM* sea el mejor algoritmo para minimizar la Tasa_inf, pues si quisieramos que los algoritmos heurísticos le dieran más importancia a esta tasa, bastaría con darle un valor más alto al hiperparámetro λ.

A continuación vemos los valores específicos de λ para cada dataset (se despejan facilmente de cualquier fila de las tablas, pues este hiperparámetro no cambia en las distintas ejecuciones):

- Ecoli con 10%: $\lambda = 2.682 * 10^{-2}$
- Ecoli con 20%: $\lambda = 1.341 * 10^{-2}$
- Iris con 10%:$\lambda = 6.34 * 10^{-3}$
- Iris con 20%:$\lambda = 3.17 * 10^{-3}$
- Rand con 10%:$\lambda = 7.21 * 10^{-3}$
- Rand con 20%:$\lambda = 3.60 * 10^{-3}$
- Newthyroid con 10%:$\lambda = 3.657 * 10^{-2}$
- Newthyroid con 20%:$\lambda = 1.828 * 10^{-2}$

(Comprobamos que tiene sentido, pues cuando aumentamos las restricciones del 10% al 20%, estamos multiplicando el número de restricciones por 2, y por tanto dividiendo el parámetro λ entre 2).

Mirando ahora a los dos datasets más pequeños (Iris y Rand) comprobamos que en estos los algoritmos heurísticos salen victoriosos tanto en Tasa_C como en Tasa_inf (y por consiguiente en Agr.). De hecho, comprobamos que para los algoritmos heurísticos (ignorando la *BL* por un momento) en estos datasets, muchos resultados se repiten en las distintas ejecuciones y en los distintos algoritmos. Esto, teniendo en cuenta la aletoriedad de las distintas ejecuciones y algoritmos, parece indicarnos que los algoritmos son capaces de llegar a óptimos locales de bastante buena calidad (o incluso a óptimos globales).

Dentro de los algoritmos heurísticos vemos que la búsqueda local (de la práctica 1) parece ser el peor de ellos en términos de la función objetivo (Agr.), lo que puede deberse a que el espacio que explora este algoritmo es más restringido que aquel que exploran el resto de algoritmos (los basados en poblaciones). Además, el algoritmo *BL* termina una vez encuentra la mejor solución de un entorno, pudiendo no haber "gastado" las 100000 evaluaciones de la función objetivo (con las sobrantes quizás se podría plantear un nuevo arranque, pero en este caso no es así); mientras que por ejemplo los algoritmos meméticos incluso ejecutan múltiples búsquedas locales suaves siempre llegando a hacer 100000 evaluaciones de la función objetivo (y por tanto evaluando 100000 soluciones). Este hecho también podría influir en los tiempos de ejecución, ya que en este caso la *BL* es la que mejores tiempos de ejecución tiene dentro de los algoritmos heurísticos.

Profundizando ahora en los tiempos de ejecución, las tablas reflejan claramente que estos son mucho mayores en los algortimos basados en poblaciones en comparación con la *BL* y el *COPKM*. Además, el *COPKM* es el que tiene (siempre que no cicle de forma infinita) mejores tiempos de ejecución. Si bien parece normal que los algoritmos basados en poblaciones tarden más, es importante señalar que el aumento tan drástico en los tiempos de ejecución (el cual es del todo apreciable) se debe también a una implementación de estos algoritmos más bien poco eficiente; y es que se han implementado en Python y trabajando con listas simples, lo cual hace que el código sea bastante entendible pero también bastante poco eficiente.

Nos centramos finalmente en comparar los resultados de los algoritmos basados en poblaciones. Comparando los algoritmos genéticos generacionales con los estacionarios vemos que no existe un claro vencedor, pues en términos del agregado ambos empatan en los datasets *Iris* y *Rand* (donde parecen funcionar muy bien); los generacionales tienen mejores resultados en el *Newthyroid*, y los estacionarios tienen mejores resultados en el *Ecoli*. Tanto en los algoritmos genéticos generacionales como en los estacionarios vemos que los resultados con el operador de cruce uniforme y el de cruce por segmento fijo son bastante similares. Aun así, comprobamos que *AGG-SF* obtiene mejores agregados qur *AGG-UN* en los conjuntos con un 10% de restricciones (primera tabla) mientras que en la segunda tabla hay un empate, pues vemos que el *AGG-SF* es mejor que el *AGG-UN* en el *Newthyroid* pero no es así en el *Ecoli*. Esto unido al hecho de que el operador de cruce por segmento fijo es menos disruptivo justifican la elección de este operador para su uso en los algortimos meméticos.

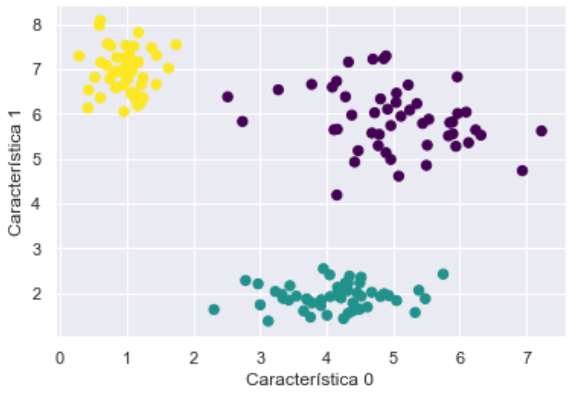
Comparando (en términos del agregado) los algoritmos meméticos entre sí y con los genéticos comprobamos de nuevo que no existe un claro vencedor, pues la mayoría de ellos empatan en los datasets pequeños (*Iris* y *Rand*) y ninguno logra imponerse en los datasets grandes (*Ecoli* y *Newthyroid*, donde en cada ocasión se impone un algoritmo distinto). Para terminar, notamos también que los algoritmos meméticos tienen un tiempo de ejecución menor (en todos los datasets y con las distintas restricciones) que los algoritmos genéticos (aunque como ya hemos comentado anteriormente estos tiempos de ejecución parecen estar muy sujetos a la implementación).

Representación gráfica de las soluciones

Para tener una idea gráfica de los clusters que están construyendo nuestros algoritmos, se proporcionan dos visualizaciones a continuación de las instancias y los centroides representados según cada par de características. Los puntos respresentarán las instancias del conjunto de datos; su color, el cluster al que han sido asignadas. Los puntos de mayor grosor y de color gris representarán los centroides. Por desgracia, estas representaciones no nos proporcionan una visualización de las restricciones.

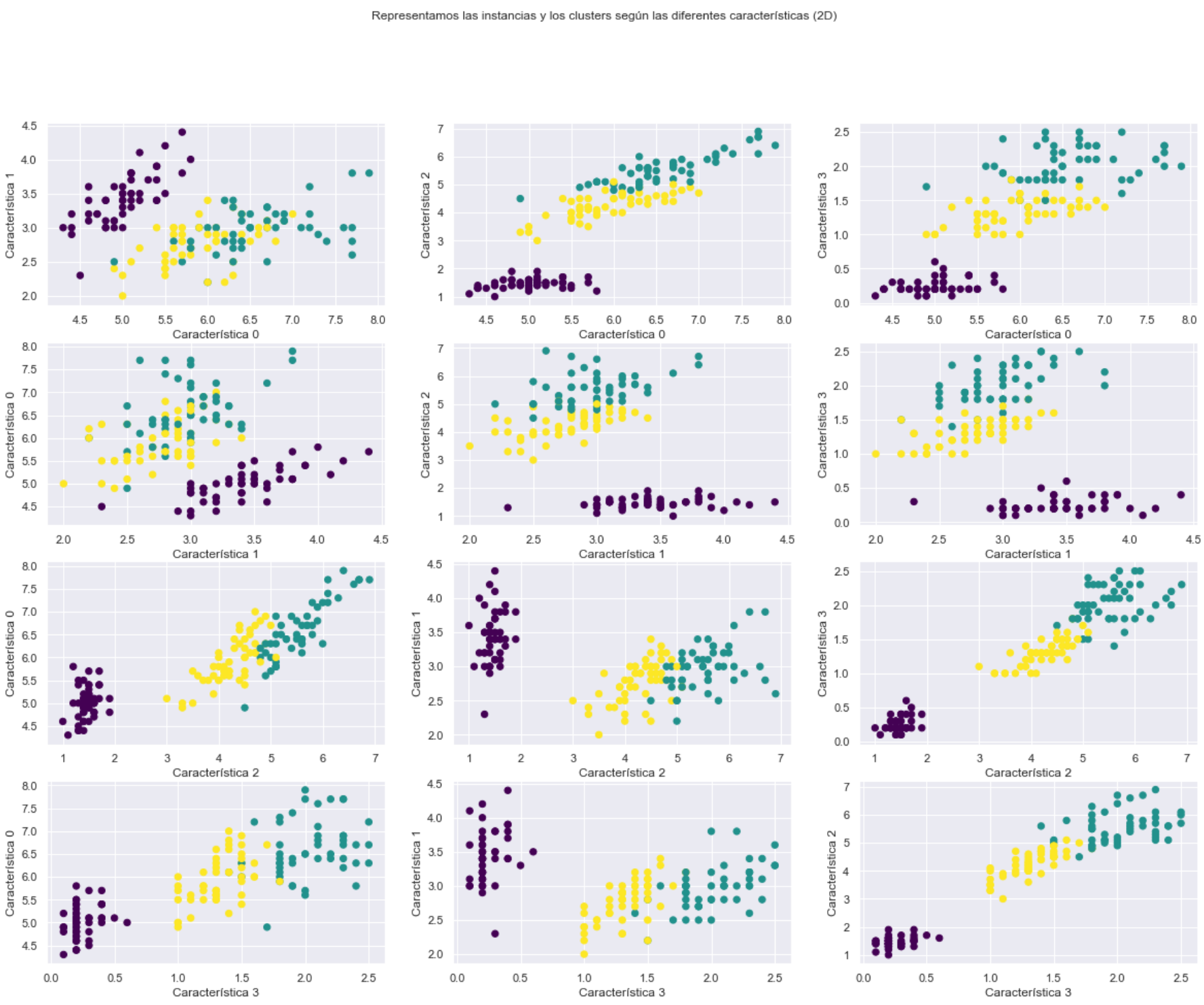
Representación de la partición solución obtenida por el algoritmo de **AGG-SF** en el dataset **Rand** con el 10% de las restricciones:

In [31]:
```
#Leemos los datos
X = read_dat("./data/rand_set.dat")
const_matrix = read_constraints_matrix("./data/rand_set_const_10.const")
const_list = constraints_matrix_to_list(const_matrix)
#Fijamos semilla
np.random.seed(0)
#Ejecutamos el algoritmo:
partition_sol = generational_genetic_algo_SF(X, const_matrix, const_list, k=3)
# Visualizamos la solución obtenida
visualise_rand_clusters(X, partition_sol)
```



Representación de la partición solución obtenida por el algoritmo **AM-(10,1.0)** en el dataset **Iris** con el 10% de las restricciones:

In [32]:
```
#Leemos los datos
X = read_dat("./data/iris_set.dat")
const_matrix = read_constraints_matrix("./data/iris_set_const_10.const")
const_list = constraints_matrix_to_list(const_matrix)
#Fijamos semilla
np.random.seed(0)
#Ejecutamos el algoritmo:
partition_sol = memetic_algo_v1(X, const_matrix, const_list, k=3)
# Visualizamos la solución obtenida
visualise_iris_clusters(X, partition_sol)
```



6.- Extra: Algoritmo memético con búsqueda local normal (no suave)

Se hacen unas pequeñas modificaciones en los argumentos y las salidas del algoritmo de búsqueda local original (el de la práctica 1) para que pueda sustituir al algoritmo de búsqueda local suave y crear así un nuevo algoritmo memético.

Además se añaden algunas variables en el algoritmo memético para monitorizar el número de búsquedas locales (suaves o no suaves) que se realizan (con 100000 evaluaciones de la función objetivo) y el número medio de evaluaciones que consumen cada una de las búsquedas locales (de nuevo, la original *BL* y la suave *BLS*).

Finalmente, compararemos estas variables ejecutando ambos algoritmos meméticos sobre el mismo conjunto de datos grande (*Ecoli* con 20% de restricciones).

In [2]:
```
def local_search_memetic(X, const_list, partition, current_func_value,
                         assignations_counter, k, max_failures, counter, lambda_):
    #Esta función no hace uso de max_failures (se recibe como argumento para minimizar los cambios en la implementación)
    n_instances = X.shape[0]
    if lambda_ == None:
        lambda_ = max_dist(X) / len(const_list)
    #Creamos una lista de parejas [(0,+1),(0,+2),..,(1,+1),(1,+2),...,(n_instances, +1),...] Esta lista representará todas las
    #operaciones de movimiento posibles desde una particion dada. Esto será lo que barajemos.
    virtual_neighborhood = [(index, to_add) for index in range(n_instances) for to_add in range(1, k)]
    found_better_sol = True
    while (counter <= 100000 and found_better_sol): #Condición-Parada: Encontrar mejor solución de un entorno o 100000 eval.
        found_better_sol = False
        np.random.shuffle(virtual_neighborhood)
        i = 0
        while (counter <= 100000 and (not found_better_sol) and i<n_instances):
            operation = virtual_neighborhood[i]
            tmp = partition[operation[0]]
            #Ejecutamos la operación
            partition[operation[0]] = (partition[operation[0]] + operation[1]) % k
            func_val = objective_func(X, partition, const_list, centroids = None, lambda_ = lambda_)
            counter += 1
            #Si la operación nos lleva a una mejor partición que sea válida, nos quedamos con ella.
            if (func_val < current_func_value and assignations_counter[tmp] > 1):
                assignations_counter[tmp] -= 1
                assignations_counter[partition[operation[0]]] += 1
                current_func_value = func_val
                found_better_sol = True
            else: #Si no, volvemos a la particición anterior y probamos con la siguiente operación
                partition[operation[0]] = tmp
            i += 1
    return partition, current_func_value, assignations_counter, counter
```

In [3]:
```
def memetic_algo_ls(X, const_matrix, const_list, k, lambda_ = None, cross_operator = fixed_segment_operator,
                    population_size = 10, cross_prob = 0.7, mutation_prob = 0.001,
                    generation_per_ls = 10, perc_ls = 1.0, best_population = False,
                    local_search_algo = smooth_local_search):
    n_instances = X.shape[0]
    n_cross_expected = int(cross_prob * population_size / 2) #Número esperado de cruces (2 hijos por cruce)
    n_mutations_expected = int(mutation_prob * population_size * n_instances) #Número esperado de mutaciones
    n_solutions_for_local_search = int(population_size * perc_ls)
    max_failures = int(0.1 * n_instances)
    n_local_searches = 0
    counters_per_ls = []
    if lambda_ == None:
        lambda_ = max_dist(X) / len(const_list)
    counter = 0 #Contamos el número de veces que se evalua la función objetivo
    #Generamos la población inicial:
    current_population, counter = generate_initial_population(X, const_list, k, lambda_, population_size, counter)
    current_population.sort(key=lambda x:x[1]) #Ordenamos la población según la calidad de los cromosomas (de mejor a peor)
    while (counter < 100000):
        for generation in range(generation_per_ls):
            new_population, counter = new_generation(X, const_list, k, lambda_, population_size, counter, n_cross_expected,
                                                     n_instances, cross_operator, n_mutations_expected, current_population)
            #Ordenamos la nueva población
            new_population.sort(key=lambda x:x[1]) #Ordenamos la población según la calidad de los cromosomas (de mejor a peor)
            #Aplicamos el elitismo
            if current_population[0][1] < new_population[0][1]:
                new_population[population_size-1] = current_population[0]
                new_population.sort(key=lambda x:x[1])
            current_population = new_population
        #Aplicamos la búsqueda local suave a la proporción de la población determinada por perc_ls y best_population.
        if best_population:
            indices_for_ls = list(range(n_solutions_for_local_search))
        else:
            indices_for_ls = random.sample(range(population_size), n_solutions_for_local_search)
        for i in indices_for_ls:
            partition, func_value, assignations_counter, counter1 = local_search_algo(X, const_list,
                                                                                      current_population[i][0],
                                                                                      current_population[i][1],
                                                                                      current_population[i][2], k,
                                                                                      max_failures,counter=0,lambda_=lambda_)
            #MODIFICACIÓN:
            n_local_searches += 1
            counters_per_ls.append(counter1)
            counter += counter1

            current_population[i] = [partition, func_value, assignations_counter]

        print("Número de evaluaciones total: " + str(counter), end="\r", flush=True)
    return current_population[0][0], n_local_searches, counters_per_ls
```

Cargamos el conjunto de datos *Ecoli* con 20% de restricciones.

In [4]:
```
fname_data = "./data/ecoli_set.dat"
X = read_dat(fname_data)
const_file = "./data/ecoli_set_const_20.const"
const_matrix = read_constraints_matrix(const_file)
const_list = constraints_matrix_to_list(const_matrix)
```

Ejecutamos el algoritmo memético que usa la búsqueda local suave (*BLS*).

In [5]:
```
np.random.seed(0)
partition_sol1, n_local_searches1, counters_per_ls1 = memetic_algo_ls(X, const_matrix, const_list, k=3,
                                                                      local_search_algo = smooth_local_search)
```

```
Número de evaluaciones total: 100006
```

Ejecutamos el algoritmo memético que usa la búsqueda local (*BL*).

In [6]:
```
np.random.seed(0)
partition_sol2, n_local_searches2, counters_per_ls2 = memetic_algo_ls(X, const_matrix, const_list, k=3,
                                                                      local_search_algo = local_search_memetic)
```

```
Número de evaluaciones total: 102656
```

Comparamos el número de búsquedas locales (suaves o no) que se ejecutan en cada caso, así como el número medio de evaluaciones de la función objetivo que consume cada tipo de búsqueda.

In [7]:
```
print("Número de búsquedas locales SUAVES realizadas: ", n_local_searches1)
print("Número medio de evaluaciones de la función objetivo por búsqueda local suave: ", np.mean(counters_per_ls1))
print("Número de búsquedas locales (no suaves) realizadas: ", n_local_searches2)
print("Número medio de evaluaciones de la función objetivo por búsqueda local (no suave): ", np.mean(counters_per_ls2))
```

```
Número de búsquedas locales SUAVES realizadas:  1270
Número medio de evaluaciones de la función objetivo por búsqueda local suave:  71.69921259842519
Número de búsquedas locales (no suaves) realizadas:  40
Número medio de evaluaciones de la función objetivo por búsqueda local (no suave):  2559.125
```

Comprobamos (como era esperable) que el número de búsquedas locales suaves que se pueden hacer con 100000 evaluaciones de la función objetivo es mucho mayor que el número de búsquedas locales (no suaves) que podemos ejecutar (1270 vs 40), pues cada *BL* consume de media muchas más evaluaciones que cada *BLS* (2559.125 vs 71.7). Este hecho justifica el uso de la *BLS* en lugar de la *BL* original para otorgarle más peso a la parte del algoritmo memético que se corresponde con el algoritmo genético generacional (si las búsquedas locales consumen menos evaluaciones, se producirán más generaciones de la población antes de llegar a las 100000 evaluaciones).

En concreto, el algoritmo memético que usa la *BLS* ha consumido $100006 - (1270 * 71.7) = 100006 - 91059 = 8947$ evaluaciones en la parte generacional del algoritmo (la asociada al algoritmo genético generacional), mientras que el algoritmo de búsqueda local ha consumido tan solo $102656 - (40 * 2559.125) = 102656 - 102365 = 291$ evaluaciones en esta parte generacional. (Recordamos que el hecho de que se sobrepasen un poco las 100000 evaluaciones se debe a que la comprobación tan solo se hace en el bucle *while* principal, lo cual se hace por simplicidad y se indicó en el foro que no sería un problema).

Finalmente mostramos también la calidad de las soluciones que se han obtenido por ambos algoritmos meméticos en terminos de desviación general, infactibilidad y función objetivo (aunque esta comparación no será muy determinante para ver qué algoritmo funciona mejor, ya que se basa en una sola ejecución en un solo conjunto de datos).

In [8]:
```
print("Desviación general usando BLS: ", general_deviation(X, partition_sol1))
print("Desviación general usando BL (no suave): ", general_deviation(X, partition_sol2))
```

```
Desviación general usando BLS:  173.60871058686703
Desviación general usando BL (no suave):  135.4291309721036
```

In [9]:
```
print("Infactibilidad usando BLS: ", infeasibility(partition_sol1, const_list))
print("Infactibilidad usando BL (no suave): ", infeasibility(partition_sol2, const_list))
```

```
Infactibilidad usando BLS:  2750
Infactibilidad usando BL (no suave):  5487
```

In [10]:
```
print("Agregado usando BLS: ", objective_func(X, partition_sol1, const_list))
print("Agregado usando BL (no suave): ", objective_func(X, partition_sol2, const_list))
```

```
Agregado usando BLS:  210.4993372695155
Agregado usando BL (no suave):  209.03599228399167
```

Vemos que el nuevo algoritmo memético que hace uso de la búsqueda local original ha sido capaz de obtener una mejor solución (en términos de la función objetivo) que el algoritmo memético que usa la búsqueda local suave. Aunque la diferencia no es muy grande (y como hemos comentado esta comparación no es del todo determinante), esto nos sugiere que darle más peso a la parte generacional del algoritmo memético no tiene por qué ser necesariamente beneficioso.

In [ ]:
```
-
```