Problema de Agrupamiento con Restricciones (PAR)

Búsquedas por trayectorias para el PAR

Metaheurísticas: Práctica 3.b, Grupo MH2 (Jueves de 17:30h a 19:30h)

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

18 de mayo 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 3) 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 va a ser evaluada 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 ← (*constraint_value* == 1 and $S[i] \neq S[j]$)
  incumplimiento_CL ← (*constraint_value* == −1 and $S[i] == S[j]$)
  incumplimiento ← (incumplimiento_ML or incumplimiento_CL)
  return incumplimiento
end

Algorithm 5: *infeasibility*

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

3. Algoritmos

En esta práctica se han usado seis algoritmos. El Greedy (*COPKM*) y el de Búsqueda Local (ya implementados en la práctica 1) ya se habían ejecutado sobre todos los conjuntos de datos en las prácticas anteriores. Los otros 4 son el de enfriamiento simulado (*ES*), búsqueda local multiarranque (*BMB*), búsqueda local reiterada (*ILS*) y la hibridación de *ILS* y *ES* (*ILS-ES*). Estos se han implementado y se han ejecutado sobre todos los conjuntos de datos. La descripción de los dos primeros algoritmos ya se vió en la primera práctica. No obstante, dada la relevancia de la búsqueda local en el resto de algoritmos (es una componente de algunos) y el hecho de que se han realizado pequeñas modificaciones en su implementación daremos una descripción del algoritmo en esta memoria también. El resto de algoritmos (los de la práctica 3) también se describen a continuación mostrando pseudocódigo y comentando los aspectos más importantes.

3.1. Generación de soluciones iniciales aleatorias

Un primer punto en común de todos los algoritmos que vamos a describir (esto es, de todos los algoritmos que compararemos excepto el Greedy) es su necesidad de generar soluciones aleatorias en cierta etapa del algoritmo. En concreto, el algoritmo *BMB* tiene la necesidad de

generar una solución inicial aleatoria cada vez que finaliza una búsqueda local y reinicia para hacer otra. El resto de algoritmos solo tienen esta necesidad en su inicio, en el que generan una solución inicial aleatoria para partir de ella. Describimos a continuación en pseudocódigo el proceso de generación de soluciones aleatorias.

El proceso se divide en dos funciones. La primera de ellas, *generate_initial_sol(X, k)*, genera una partición de forma aleatoria sin tener en cuenta la restricción fuerte de que no haya ningún cluster con 0 instancias asignadas.

Algorithm 6: 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
```

La segunda, *generate_valid_initial_sol(X, k)*, hace uso de esta primera función y se asegura de que se devuelve una solución que cumpla que asigna a cada uno de los k clusters al menos una instancia, o lo que es lo mismo, una solución "válida".

Algorithm 7: generate_valid_initial_sol

```
Data: Conjunto de datos X, número de clusters k
Result: Una partición válida S y su contador de asignaciones assignations_counter
begin
    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
    return S, assignations_counter
end
```

Cabe comentar que no solo se devuelve la solución generada, sino también un diccionario (*assignations_counter*) con la cuenta de asignaciones de cada cluster que se podrá usar para comprobar que no se rompe la restricción fuerte (de no asignar ninguna instancia a algún cluster) cada vez que se modifique la partición.

3.2. Búsqueda local

Tratamos ahora el algoritmo búsqueda local paso por paso. Primero se generará una partición de forma aleatoria como acabamos de describir en la sección anterior.

Una vez tengamos la partición inicial, en cada iteración exploramos el vecindario hasta encontrar una solución mejor y sustituimos la actual por la encontrada. Repetiremos este proceso hasta que exploremos todo el vecindario y no encontremos una solución mejor o hasta que hayamos evaluado la función objetivo *max_evaluations* veces.

Con el objetivo de poder reutilizar las funciones de este algoritmo en algoritmos posteriores (en concreto en el *ILS*), se ha dividido el algoritmo en dos funciones. Una primera (*local_search_from_partition*) en la que se lanza una búsqueda local con un punto de partida (una solución inicial S) dado como argumento. Se describe en pseudocódigo a continuación.

Algorithm 8: local_search_from_partition

Data: Solución inicial S, contador de asignaciones de S *assignations_counter*, conjunto de datos X, lista de restricciones *const_list*, número de clusters k, hiperparámetro λ, número máximo de evaluaciones de la función objetivo *max_evaluations*

Result: Partición solución S

```
begin
    n ← longitud(X)
    current_func_value ← objective_function(X, S, const_list, λ)
    counter ← 1 (Número de veces que se evalua la función objetivo)
    -Creamos una lista de parejas [(0,+1),(0,+2),..]. Esta lista representará todas las
     operaciones de movimiento posibles desde una partición dada.
    virtual_neighborhood ← [(index, add) for index in {0,1,...,n-1} for add in {1,2,...,k}]
    found_better_sol ← True
    while counter < max_evaluations and found_better_sol do
        found_better_sol ← False
        virtual_neighborhood ← RandomShuffle(virtual_neighborhood)
        i ← 0
        while counter < max_evaluations and not found_better_sol and i < n do
            operation ← virtual_neighborhood[i]
            -Ejecutamos la operación
            tmp ← S[operation[0]]
            S[operation[0]] ← (S[operation[0]] + operation[1]) mod k
            func_val ← objective_function(X, S, const_list, λ)
            counter ← counter + 1
            -Si llegamos a una mejor partición que sea válida, la elegimos.
            if func_val < current_func_value and assignations_counter[tmp] > 1 then
                assignations_counter[tmp] ← assignations_counter[tmp] - 1
                assignations_counter[partition[operation[0]]] ←
                 assignations_counter[partition[operation[0]]] + 1
                current_func_value ← func_val
                found_better_sol ← True
            else
                -Si no, volvemos a la particición anterior
                S[operation[0]] ← tmp
            end
            i ← i + 1
        end
    end
    return S
end
```

Y una segunda función (*local_search*) en la que se tiene como punto de partida una solución generada de forma aleatoria. Esta segunda función hace uso de la primera, y es la que se corresponde con el algoritmo original de la práctica 1 (el que veremos como *BL* más adelante en los resultados).

Algorithm 9: local_search

Data: Conjunto de datos X, lista de restricciones *const_list*, número de clusters k, hiperparámetro λ, número máximo de evaluaciones de la función objetivo *max_evaluations*
Result: Partición solución S
begin
  S, assignations_counter ← generate_valid_initial_sol(X, k)
  return local_search_from_partition(S, *assignations_counter*, X, *const_list*, k, λ, *max_evaluations*)
end

3.3. Enfriamiento simulado

El Enfriamiento Simulado (*ES*) es un algoritmo de búsqueda por entornos con un criterio probabilístico de aceptación de soluciones basado en termodinámica. Este criterio depende del parámetro temperatura en cada etapa del algoritmo, que a su vez quedará determinado por la temperatura inicial y el esquema de enfriamiento que hayamos elegido. En nuestra implementación, siguiendo las instrucciones del guión se ha decidido tomar la temperatura inicial como

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

siendo $f(S_0)$ la función objetivo evaluada en la solución inicial, y ϕ la probabilidad de aceptar una solución un μ por 1 peor que la inicial (en nuestro caso siempre usamos $\mu = \phi = 0{,}3$).

Por su parte, el esquema de enfriamiento usado es el de Cauchy modificado, el cuál viene descrito por las siguientes expresiones.

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

donde M es el número de enfriamientos a realizar y T_f es la temperatura final. En nuestra implementación, siguiendo las indicaciones del guión usaremos $M = max\_evaluations/max\_neighbours$ y $T_f = 10^{-3}$; discutiremos más adelante el valor de *max_evaluations* y *max_neighbours*.

Algorithm 10: cauchy_scheme

Data: Valor actual del parámetro temperatura *current_temp*, parámetro beta β
Result: Siguiente valor del parámetro temperatura
begin
  return current_temp/$(1 + \beta * current\_temp)$
end

De nuevo con el objetivo de poder reutilizar las funciones de este algoritmo en algoritmos posteriores (en concreto en el *ILS-ES*), se ha dividido el algoritmo en dos funciones. Una primera (*simulated_annealing_from_partition*) en la que se lanza el algortimo con un punto de partida (una solución inicial S) dado como argumento.

Algorithm 11: simulated_annealing_from_partition

Data: Solución inicial *current_partition*, asignaciones de esta *assignations_counter*, conjunto de datos X, lista de restricciones *const_list*, número de clusters k, hiperparámetro λ, máximas evaluaciones de la función objetivo *max_evaluations*
Result: Partición solución S
begin
n ← longitud(X)
best_partition ← current_partition
current_func_value ← objective_function(X, current_partition, const_list, λ)
best_func_value ← current_func_value
initial_temp ← $(0{,}3 * current\_func\_value)/(-\log(0{,}3))$
current_temp ← initial_temp
final_temp ← 10^{-3}
while *final_temp* $\geq$ *initial_temp* **do**
final_temp ← final_temp * 10^{-3}
end
max_neighbours ← 10 * n * max_evaluations/100000 -Escalamos max_neighbours
max_successes ← 0.1 * max_neighbours
m_iterations ← max_evaluations/max_neighbours
beta ← (initial_temp - final_temp)/(m_iterations * initial_temp * final_temp)
annealings ← 0; successes ← 1
while *annealings<m_iterations and successes* > 0 **do**
successes ← 0; generated_neighbours ← 0
while *generated_neighbours* $<$ *max_neighbours and successes* $<$ *max_successes* **do**
-Generamos un vecino comprobando que no rompe la restriccion fuerte
gen_idx ← entero_aleatorio_entre(0,n-1)
while *assignations_counter[current_partition[gen_idx]]* ≤ 1 **do**
gen_idx← entero_aleatorio_entre(0,n-1)
end
previous_gen_value = current_partition[gen_idx]
current_partition[gen_idx] ← (current_partition[gen_idx] + entero_aleatorio_entre(1,k-1)) mod k
candidate_func_value ← objective_func(X, current_partition, const_list,λ)
generated_neighbours += 1
increment_f ← candidate_func_value - current_func_value
if *(increment_f* < 0 *or aleatorio_entre(0,1)* $\leq$ *exp(-increment_f/current_temp))* **then**
successes += 1
current_func_value ← candidate_func_value
assignations_counter[previous_gen_value] -= 1
assignations_counter[current_partition[gen_idx]] += 1
if *current_func_value* $<$ *best_func_value* **then**
best_func_value ← current_func_value
best_partition ← current_partition
end
else
-Si la solución no es aceptada volvemos a la partición anterior
current_partition[gen_idx] ← previous_gen_value
end
end
annealings += 1; current_temp ← cauchy_scheme(current_temp, beta)
end
return best_partition
end

Y una segunda función (*simulated_annealing_algo*) en la que se tiene como punto de partida una solución generada de forma aleatoria. Esta segunda función hace uso de la primera, y es la que se corresponde con el algoritmo original de la práctica 1 (el que veremos como *ES* más adelante en los resultados).

Algorithm 12: simulated_annealing_algo

Data: Conjunto de datos X, lista de restricciones *const_list*, número de clusters k, hiperparámetro λ, número máximo de evaluaciones de la función objetivo *max_evaluations*
Result: Partición solución S
begin
    S, assignations_counter ← generate_valid_initial_sol(X, k)
    return simulated_annealing_from_partition(S, *assignations_counter*, X, *const_list*, k, λ, *max_evaluations*)
end

Descrito el algoritmo en pseudocódigo, comentamos algunos aspectos importantes. El primer aspecto a comentar es la comprobación de que la temperatura final sea menor que la temperatura inicial (ver primer bucle "while"del algoritmo 11). Esta comprobación se nos sugiere hacerla en el guión de la práctica y, en caso de que tengamos $T_0 \leq T_f = 10^{-3}$ lo que hacemos es multiplicar T_f por 10^{-3} para que su valor disminuya hasta que obtener $T_f < T_0$. De cualquier forma, experimentalmente se ha comprobado que en ninguna de nuestras ejecuciones (las que se presentan en la sección de resultados) se da este caso por lo que la comprobación carece de relevancia práctica en nuestra tarea.

En segundo lugar, dada una iteración del algoritmo los valores de las variables *max_neighbours* y *max_successes* nos indican el número máximo de soluciones vecinas a generar y el número máximo de soluciones a aceptar antes de pasar al siguiente enfriamiento. En el guión de la práctica se nos indica que usemos $max\_neighbours = 10 * n$ donde n es el número de instancias del conjunto de datos (X) y $max\_successes = 0{,}1 * max\_neighbours$. Estos valores están adaptados para una ejecución individual del algoritmo y considerando $max\_evaluations = 100000$. Sin embargo, cuando usamos el algoritmo dentro de *ILS-ES*, donde se lanzan varios enfriamientos simulados con $max\_evaluations = 10000$, mantener esos parámetros provoca que se hagan muy pocas iteraciones (pues recordemos que el número de iteraciones es $M = max\_evaluations/max\_neighbours$), dañando la efectividad del algoritmo. Es por ello que se ha decidido escalar el valor *max_neighbours* con respecto al número de evaluaciones *max_evaluations* de la forma que se muestra en el pseudocódigo:

$$max\_neighbours = 10 * n * (max\_evaluations/100000).$$

Al hacer el ajuste en *max_neighbours*, las variables M y *max_successes* se ajustan de forma proporcional (pues como hemos visto se calculan a partir de *max_neighbours*). Así, esta pequeña modificación no supone ningún cambio cuando el algoritmo se ejecuta de forma individual con $max\_evaluations = 100000$ y mejora su comportamiento cuando se ejecuta dentro de *ILS-ES*.

3.4. Búsqueda Multiarranque Básica

El algoritmo de Búsqueda Multiarranque Básica *BMB* consiste simplemente en generar un determinado número de soluciones aleatorias iniciales y optimizar cada una de ellas con el algoritmo de búsqueda local. Finalmente se devolverá la mejor solución encontrada en todo el proceso. Se describe a continuación en pseudocódigo (ver algoritmo 13).

Como vemos, una vez vista la búsqueda local original, este algoritmo es bastante sencillo. Cabe destacar el argumento $restart\_mode = True$ de la búsqueda local, el cual no explicamos

Algorithm 13: random_restart_local_search

Data: Conjunto de datos X, lista de restricciones *const_list*, número de clusters k, hiperparámetro λ, número de búsquedas locales a ejecutar n_ls, número máximo de evaluaciones de la función objetivo en cada búsqueda local *evaluations_per_ls*

Result: Mejor solución encontrada *best_partition*

```
begin
    best_partition, best_func_value ← local_search(X, const_matrix, const_list, k, λ,
     max_evaluations = evaluations_per_ls, restart_mode = True)
    foreach i ∈ {0, 1, ..., n_ls − 2} do
        partition, func_value ← local_search(X, const_matrix, const_list, k, λ,
         max_evaluations = evaluations_per_ls, restart_mode = True)
        if func_value < best_func_value then
            best_func_value ← func_value
            best_partition ← partition
        end
    end
    return best_partition
end
```

en el algoritmo original (algoritmo 9) porque carecía de interés para el algoritmo propiamente. Este argumento lo único que hace es indicarle a la función (en este caso a *local_search*) que no solo devuelva la mejor partición solución encontrada, sino también el valor de la función objetivo asociado a esta partición. Esto nos permite no tener que llamar a la función objetivo tras cada búsqueda local para comparar la calidad de las soluciones dadas por las distintas búsquedas locales.

En nuestro caso, el número de búsquedas locales que fijamos es $n\_ls = 10$ (por lo que en la práctica la ventaja del argumento *restart_mode* no tiene mucha relevancia); y el número máximo de evaluaciones de la función objetivo en cada búsqueda local es $evaluations\_per\_ls = 10000$.

3.5. Búsqueda Local Reiterada (ILS) y algoritmo híbrido ILS-ES

Describimos los algortimos ILS e ILS-ES en esta misma sección porque en esencia son el mismo salvo porque uno (el ILS) usará repetidamente la búsqueda local mientras que el otro (ILS-ES) usará repetidamente el algoritmo de enfriamiento simulado. En concreto, el algoritmo consiste en generar una solución inicial aleatoria y aplicar el algoritmo de búsqueda (BL o ES) sobre ella. Una vez obtenida una solución optimizada, se comprobará si es mejor que la mejor solución encontrada hasta el momento y se realizará una mutación sobre la mejor de estas dos, volviendo a aplicar el algoritmo de búsqueda (BL o ES) sobre esta solución mutada. Se repite este proceso varias veces y finalmente se devuelve la mejor solución de todas.

Comenzamos describiendo en pseudocódigo el operador de mutación que usa el algoritmo, esto es, el operador de mutación por segmento (ver algoritmo 14). Vemos que el operador consiste en coger un "segmento de genes"de la solución que empiece en una posición aleatoria y tenga una determinada longitud y cambiar los genes de dicho segmento asignándoles clusters de forma aleatoria. En nuestro caso, usaremos siempre $segment\_length = 0{,}1 * n$ siendo n el número de instancias del conjunto de datos con el que estemos trabajando. Un aspecto de implementación que cabe mencionar es que este operador de mutación no cambia la partición que se le pasa como argumento, sino que crea y devuelve una nueva (esto es importante porque, como veremos más adelante, se le pasará como argumento la mejor solución encontrada hasta el momento).

Además, es claro que este operador podría 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 15. Esta función ya fue usada en la práctica anterior.

Algorithm 14: segment_mutation_operator

```
Data: Partición a mutar S, longitud del segmento segment_length, número de clusters
      k
Result: Mutación de la partición dada, mut_partition
begin
    n ← longitud(S)
    segment_ini ← entero_aleatorio_entre(0, n-1)
    segment_indices ← [i mod(n) for i ∈ {segment_ini,..., segment_ini + segment_lenght}]
    mut_partition = copy(S)
    foreach i ∈ segment_indices do
        mut_partition[i] ← entero_aleatorio_entre(0, k-1)
    end
    return mut_partition
end
```

Algorithm 15: 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 ∈ {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).

Ahora sí, mostramos el pseudocódigo del algoritmo (ver algoritmo 16). Es importante notar que será el argumento *search_algo* el que determine si estamos ante la búsqueda local reiterada original (*ILS*) o ante su variante hibridada (*ILS-ES*). En concreto tendremos la *ILS* cuando pasemos como argumento $search\_algo = local\_search\_from\_partition$ y la *ILS-ES* cuando pasemos $search\_algo = simulated\_annealing\_from\_partition$. Como ya se ha comentado anteriormente, estas dos funciones ($local\_search\_from\_partition$ y $simulated\_annealing\_from\_partition$) lanzan los algoritmos *BL* y *ES* con una solución que se pasa como argumento como punto de partida. Cabe mencionar que de nuevo se utiliza el argumento *restart_mode* para que ambas funciones devuelvan tanto la partición optimizada como el valor de la función objetivo asociado a esta.

Finalmente, respecto a los parámetros elegidos en nuestras ejecuciones, el número de búsquedas locales (o enfriamientos simulados) que fijamos es $n\_ls = 10$; y el número máximo de evaluaciones de la función objetivo en cada una es $evaluations\_per\_ls = 10000$.

Algorithm 16: iterated_local_search

Data: Conjunto de datos X, lista de restricciones *const_list*, número de clusters k, hiperparámetro λ, número de búsquedas locales (o enfriamientos simulados) a ejecutar n_ls, número máximo de evaluaciones de la función objetivo en cada búsqueda local (o enfriamiento simulado) *evaluations_per_ls*, algoritmo de búsqueda a usar *search_algo* (*BL* o *ES*)

Result: Mejor solución encontrada *best_partition*

```
begin
    n ← longitud(X)
    best_partition, assignations_counter ← generate_valid_initial_sol(X, k)
    best_partition, best_func_value ← search_algo(best_partition, assignations_counter,
     X, const_list, k, λ, max_evaluations = evaluations_per_ls, restart_mode = True)
    foreach i ∈ {0, 1, ..., n_ls − 2} do
        candidate_partition ← segment_mutation_operator(best_partition,
         segment_length = truncar(n * 0.1), k=k)
        assignations_counter ← cuenta_asignaciones(candidate_partition)
        if not asignaciones_validas(assignations_counter) then
            repair_partition(candidate_partition, assignations_counter, k)
        end
        candidate_partition, func_value ← search_algo(candidate_partition,
         assignations_counter, X, const_list, k, λ, max_evaluations = evaluations_per_ls,
         restart_mode = True)
        if func_value < best_func_value then
            -Criterio de aceptación el mejor.
            best_func_value ← func_value
            best_partition ← candidate_partition
        end
    end
    return best_partition
end
```

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 todos los algoritmos explicados en la sección 2 se encuentran 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_algorithmsP3_new.npy',allow_pickle='TRUE').item()
global_results_dfs = np.load(results_folder + 'dataframes_global_comparisonP3_new.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 van a ser comparados con los algoritmos implementados en esta práctica 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 algoritmo de enfriamiento simulado (*ES*) en cada conjunto de datos dado el 10% del total de restricciones:

```
In [7]: dataframes[("ES", 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	24.388364	83.601271	610.000000	99.967294	568.602636	0.373341	0.0	0.373341	15.648497	25.135528	597.000	46.968574	133.888700
Ejecución 2	1.488290e-01	0.0	1.488290e-01	20.386946	85.184817	540.000000	99.672773	516.126057	0.373341	0.0	0.373341	13.962316	39.769372	175.000	46.169344	47.727468
Ejecución 3	1.488290e-01	0.0	1.488290e-01	18.639687	85.080250	520.000000	99.031615	312.123613	0.373341	0.0	0.373341	10.683704	29.457297	510.000	48.108643	90.431475
Ejecución 4	1.488290e-01	0.0	1.488290e-01	29.714125	83.449833	596.000000	99.440243	560.427117	0.373341	0.0	0.373341	12.860730	29.722344	501.000	48.044549	47.193736
Ejecución 5	1.488290e-01	0.0	1.488290e-01	23.612877	85.099526	513.000000	98.863083	642.589063	0.373341	0.0	0.373341	10.621776	39.769372	175.000	46.169344	58.698207
Media	1.488290e-01	0.0	1.488290e-01	23.348400	84.483139	555.800000	99.395002	519.973697	0.373341	0.0	0.373341	12.755405	32.770783	391.600	47.092090	75.587917
Desviación típica	1.241267e-17	0.0	1.241267e-17	3.810935	0.784123	39.791456	0.405391	111.581578	0.000000	0.0	0.000000	1.932978	5.941881	180.004	0.855422	33.127055

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

```
In [8]: dataframes[("ES", 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	43.557818	81.237957	1495.000000	101.293043	700.238895	0.373341	0.0	0.373341	48.718909	28.943602	990.000000	47.042444	74.611647
Ejecución 2	1.488290e-01	0.0	1.488290e-01	50.172229	80.227425	1570.000000	101.288619	1016.689056	0.373341	0.0	0.373341	22.124907	25.598305	1071.000000	45.177961	71.778901
Ejecución 3	1.488290e-01	0.0	1.488290e-01	46.659206	82.082115	1507.000000	102.298178	698.143733	0.373341	0.0	0.373341	35.171341	25.598305	1071.000000	45.177961	71.779810
Ejecución 4	1.488290e-01	0.0	1.488290e-01	30.094212	81.602267	1496.000000	101.670768	857.531322	0.373341	0.0	0.373341	26.845383	29.061726	1002.000000	47.379948	71.138959
Ejecución 5	1.488290e-01	0.0	1.488290e-01	55.373661	83.574147	1265.000000	100.543835	778.606655	0.373341	0.0	0.373341	32.306933	26.913748	1001.000000	45.213688	387.785290
Media	1.488290e-01	0.0	1.488290e-01	45.171425	81.744782	1466.600000	101.418888	810.241932	0.373341	0.0	0.373341	33.033495	27.223137	1027.000000	45.998400	135.418921
Desviación típica	1.241267e-17	0.0	1.241267e-17	8.499803	1.099084	104.534396	0.571860	118.763771	0.000000	0.0	0.000000	9.037926	1.530770	36.171812	0.996064	126.188915

Resultados obtenidos por el algoritmo de Búsqueda Multiarranque Básica (*BMB*) en cada conjunto de datos dado el 10% del total de restricciones:

```
In [9]: dataframes[("BMB", 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	0.148829	0.0	0.148829	35.973920	82.261003	958.000000	107.963709	1199.788595	0.373341	0.0	0.373341	33.161354	27.737002	524.000000	46.900345	144.904103
Ejecución 2	0.148829	0.0	0.148829	42.360612	87.164230	687.000000	105.596129	1238.318286	0.373341	0.0	0.373341	33.902830	25.598305	561.000000	46.114785	157.582788
Ejecución 3	0.148829	0.0	0.148829	38.299747	86.821685	1057.000000	115.180515	1236.363383	0.421364	9.0	0.486178	32.950595	27.720459	530.000000	47.103230	138.466757
Ejecución 4	0.148829	0.0	0.148829	37.644658	82.299854	950.000000	107.787923	1240.492002	0.373341	0.0	0.373341	35.016354	37.723856	245.000000	46.683816	160.337470
Ejecución 5	0.153919	5.0	0.185634	35.018186	90.157633	702.000000	108.991975	1190.254087	0.373341	0.0	0.373341	33.753927	27.407462	531.000000	46.826804	162.037326
Media	0.149847	1.0	0.156190	37.859425	85.740881	870.800000	109.104050	1221.043271	0.382946	1.8	0.395909	33.757012	29.237417	478.200000	46.725796	152.665688
Desviación típica	0.002036	2.0	0.014722	2.534895	3.054570	148.877668	3.233391	21.499396	0.019209	3.6	0.045135	0.722818	4.316571	117.308823	0.334133	9.302833

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

```
In [10]: dataframes[("BMB", 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	65.496991	85.079274	1666.000000	107.428287	2389.854106	0.373341	0.0	0.373341	59.260537	25.598305	1071.000000	45.177961	299.439605
Ejecución 2	1.488290e-01	0.0	1.488290e-01	64.656136	84.241395	1730.000000	107.448952	2461.710289	0.373341	0.0	0.373341	58.242410	25.598305	1071.000000	45.177961	274.448983
Ejecución 3	1.488290e-01	0.0	1.488290e-01	62.112224	83.478556	1606.000000	105.022682	2368.743005	0.373341	0.0	0.373341	64.347739	25.598305	1071.000000	45.177961	244.491781
Ejecución 4	1.488290e-01	0.0	1.488290e-01	69.498532	83.482585	1996.000000	110.258473	2469.102916	0.373341	0.0	0.373341	58.544781	29.235128	951.000000	46.620985	282.046933
Ejecución 5	1.488290e-01	0.0	1.488290e-01	58.961240	96.383800	1467.000000	116.063272	2271.723416	0.373341	0.0	0.373341	65.013118	32.527740	674.000000	44.849578	258.800060
Media	1.488290e-01	0.0	1.488290e-01	64.145025	86.533122	1693.000000	109.244333	2392.226747	0.373341	0.0	0.373341	61.081717	27.711556	967.600000	45.400889	271.845472
Desviación típica	1.755417e-17	0.0	1.755417e-17	3.514342	4.960599	174.672265	3.791310	71.855517	0.000000	0.0	0.000000	2.964362	2.789781	153.981298	0.623164	18.924542

Resultados obtenidos por la búsqueda local reiterada (*ILS*) en cada conjunto de datos dado el 10% del total de restricciones:

```
In [11]: dataframes[("ILS", 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	14.089368	79.928862	872.00000	103.324227	788.760320	0.373341	0.0	0.373341	13.986202	37.903458	237.000000	46.570848	63.097812
Ejecución 2	0.148829	0.0	0.148829	14.472382	84.797558	669.00000	102.746524	652.890976	0.373341	0.0	0.373341	15.574867	26.367764	530.000000	45.750534	70.026819
Ejecución 3	0.148829	0.0	0.148829	15.178891	84.682601	781.00000	105.636477	812.215212	0.373341	0.0	0.373341	14.336048	37.903458	237.000000	46.570848	54.364605
Ejecución 4	0.148829	0.0	0.148829	12.791483	87.174681	571.00000	102.494353	674.356658	0.373341	0.0	0.373341	15.918365	37.903458	237.000000	46.570848	58.635910
Ejecución 5	0.148829	0.0	0.148829	12.921953	77.734082	991.00000	104.322162	770.865840	0.373341	0.0	0.373341	16.353993	39.769372	175.000000	46.169344	51.939596
Media	0.148829	0.0	0.148829	13.890815	82.863557	776.80000	103.704749	739.817801	0.373341	0.0	0.373341	15.233895	35.969502	283.200000	46.326485	59.612948
Desviación típica	0.000000	0.0	0.000000	0.914769	3.480202	147.60542	1.152460	63.940864	0.000000	0.0	0.000000	0.916760	4.854955	125.714597	0.327277	6.449388

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

```
In [12]: dataframes[("ILS", 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	25.475598	84.392186	1551.000000	105.198500	1484.750780	0.373341	0.0	0.373341	27.530818	26.399888	1058.000000	45.741883	143.694905
Ejecución 2	1.488290e-01	0.0	1.488290e-01	27.758618	82.132683	1804.000000	106.332934	1230.047589	0.373341	0.0	0.373341	28.165047	25.598305	1071.000000	45.177961	125.638666
Ejecución 3	1.488290e-01	0.0	1.488290e-01	27.259212	81.397032	1671.000000	103.813118	1388.543345	0.373341	0.0	0.373341	24.086512	25.598305	1071.000000	45.177961	114.866793
Ejecución 4	1.488290e-01	0.0	1.488290e-01	28.197152	85.220991	1408.000000	104.108992	1567.492814	0.373341	0.0	0.373341	24.867744	39.039312	412.000000	46.571355	123.314924
Ejecución 5	1.488290e-01	0.0	1.488290e-01	27.425746	84.166812	1651.000000	106.314603	1365.416083	0.373341	0.0	0.373341	27.285214	31.868410	705.000000	44.756979	165.571916
Media	1.488290e-01	0.0	1.488290e-01	27.223265	83.461941	1617.000000	105.153629	1407.250122	0.373341	0.0	0.373341	26.387067	29.700844	863.400000	45.485228	134.617441
Desviación típica	1.241267e-17	0.0	1.241267e-17	0.930773	1.448256	131.968178	1.061025	114.179543	0.000000	0.0	0.000000	1.604802	5.224238	265.675441	0.626852	18.105431

Resultados obtenidos por la variante de la búsqueda local reiterada hibridada con el algoritmo de enfriamiento simulado (*ILS-ES*) en cada conjunto de datos dado el 10% del total de restricciones:

```
In [13]: dataframes[("ILS-ES", 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	31.481773	83.669590	654.00000	101.216114	982.533511	0.373341	0.0	0.373341	24.226811	39.769372	175.000000	46.169344	103.600461
Ejecución 2	1.488290e-01	0.0	1.488290e-01	31.929571	80.711979	839.00000	103.221969	889.766312	0.373341	0.0	0.373341	28.743328	26.367764	530.000000	45.750534	106.869834
Ejecución 3	1.488290e-01	0.0	1.488290e-01	32.470384	84.019489	639.00000	101.163570	884.927794	0.373341	0.0	0.373341	25.037412	26.475769	530.000000	45.858540	100.861901
Ejecución 4	1.488290e-01	0.0	1.488290e-01	29.785579	83.050520	764.00000	103.548294	923.543959	0.373341	0.0	0.373341	26.397490	27.089132	528.000000	46.398761	110.278951
Ejecución 5	1.488290e-01	0.0	1.488290e-01	27.309467	85.248522	633.00000	102.231625	936.358762	0.373341	0.0	0.373341	25.889336	28.428961	508.000000	47.007164	123.030217
Media	1.488290e-01	0.0	1.488290e-01	30.595355	83.340020	705.80000	102.276314	923.426068	0.373341	0.0	0.373341	26.058875	29.626200	454.200000	46.236869	108.928273
Desviación típica	1.241267e-17	0.0	1.241267e-17	1.872666	1.496869	81.94486	0.987564	35.428956	0.000000	0.0	0.000000	1.532941	5.124386	139.846201	0.448057	7.725002

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

In [14]:
```
dataframes[("ILS-ES", 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	1.488290e-01	0.0	1.488290e-01	59.544902	80.788306	1573.000000	101.889744	1786.099009	0.373341	0.0	0.373341	52.304144	39.039312	412.000000	46.571355	172.986608
Ejecución 2	1.488290e-01	0.0	1.488290e-01	59.077231	81.585384	1576.000000	102.727066	1698.232064	0.373341	0.0	0.373341	46.926283	39.039312	412.000000	46.571355	191.684510
Ejecución 3	1.488290e-01	0.0	1.488290e-01	57.503053	84.524849	1278.000000	101.668930	1910.551740	0.373341	0.0	0.373341	54.390607	25.598305	1071.000000	45.177961	218.044906
Ejecución 4	1.488290e-01	0.0	1.488290e-01	53.487030	85.648924	1242.000000	102.310072	1754.837751	0.373341	0.0	0.373341	54.855763	25.598305	1071.000000	45.177961	210.938027
Ejecución 5	1.488290e-01	0.0	1.488290e-01	55.732992	80.518558	1669.000000	102.907815	1741.673109	0.373341	0.0	0.373341	49.060732	25.598305	1071.000000	45.177961	205.504287
Media	1.488290e-01	0.0	1.488290e-01	57.069042	82.613204	1467.600000	102.300725	1778.278735	0.373341	0.0	0.373341	51.507506	30.974708	807.400000	45.735319	199.831668
Desviación típica	1.755417e-17	0.0	1.755417e-17	2.234588	2.080586	173.358126	0.472966	71.908589	0.000000	0.0	0.000000	3.070245	6.584722	322.842748	0.682621	15.964243

Análisis

Nota: Gran parte de los fenómenos que se comentan en esta parte del análisis ya fueron observados y comentados en la práctica 2; no obstante se han modificado ciertas partes de forma que el análisis coincide con lo que se observa exactamente en los resultados de esta práctica.

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`. Este fenómeno no es nuevo y ya lo hemos observado en las dos prácticas anteriores; para explicarlo 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 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. No obstante, en algunos de los datasets (en concreto *Newthyroid* y *Rand*) si parece haber una pequeña mejora general en los estadísticos cuando usamos el 20% de las restricciones (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 [15]:
```
global_results_dfs[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
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
ES	0.148829	0.0	0.148829	23.348400	84.483139	555.8	99.395002	519.973697	0.373341	0.0	0.373341	12.755405	32.770783	391.6	47.092090	75.587917
BMB	0.149847	1.0	0.156190	37.859425	85.740881	870.8	109.104050	1221.043271	0.382946	1.8	0.395909	33.757012	29.237417	478.2	46.725796	152.665688
ILS	0.148829	0.0	0.148829	13.890815	82.863557	776.8	103.704749	739.817801	0.373341	0.0	0.373341	15.233895	35.969502	283.2	46.326485	59.612948
ILS-ES	0.148829	0.0	0.148829	30.595355	83.340020	705.8	102.276314	923.426068	0.373341	0.0	0.373341	26.058875	29.626200	454.2	46.236869	108.928273

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

In [16]:
```
global_results_dfs[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
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
ES	0.148829	0.0	0.148829	45.171425	81.744782	1466.6	101.418888	810.241932	0.373341	0.0	0.373341	33.033495	27.223137	1027.0	45.998400	135.418921
BMB	0.148829	0.0	0.148829	64.145025	86.533122	1693.0	109.244333	2392.226747	0.373341	0.0	0.373341	61.081717	27.711556	967.6	45.400889	271.845472
ILS	0.148829	0.0	0.148829	27.223265	83.461941	1617.0	105.153629	1407.250122	0.373341	0.0	0.373341	26.387067	29.700844	863.4	45.485228	134.617441
ILS-ES	0.148829	0.0	0.148829	57.069042	82.613204	1467.6	102.300725	1778.278735	0.373341	0.0	0.373341	51.507506	30.974708	807.4	45.735319	199.831668

Análisis: Greedy vs Algoritmos heurísticos

Nota: Gran parte de los fenómenos que se comentan en esta parte del análisis ya fueron observados y comentados en la práctica 2; no obstante se han modificado ciertas partes de forma que el análisis coincide con lo que se observa exactamente en los resultados de esta práctica.

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 resultados 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.

Como ya vimos en el análisis de la práctica anterior, 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, le podríamos dar 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).

Análisis: Comparación entre los algoritmos heurísiticos de esta práctica (BL, ES, BMB, ILS e ILS-ES)

Es importante recordar que en esta práctica hemos usado dos tipos de algoritmos distintos. Por un lado tenemos a los algoritmos de búsqueda basados en trayectorias simples (que son *BL* y *ES*) y por otro a los algoritmos basados en trayectorias múltiples (*BMB*, *ILS* e *ILS-ES*). Estos últimos ejecutan de forma repetida alguno de los algoritmos basados en trayectorias simples (*BL* o *ES*); por lo que podría parecer obvio que estos algoritmos deberían dar mejores resultados que los primeros. Sin embargo, debemos tener en cuenta que los algoritmos basados en trayectorias múltiples ejecutan los basados en trayectorias simples con un límite máximo de evaluaciones de la función objetivo mucho menor que el límite con el que se ejecutan ellos de forma individual. En concreto cuando *BL* o *ES* se ejecutan de forma individual lo hacen con un límite de 100000 evaluaciones, mientras que cuando es un algoritmo basado en trayectorias múltiples el que los ejecuta, lo hace con un límite de 10000 evaluaciones (10 veces menor). Este hecho abre claramente la posibilidad de que los algoritmos basados en trayectorias simples puedan tener mejores resultados que los de los basados en trayectorias múltiples. Veamos a continuación qué pasa en nuestro caso.

En primer lugar, dentro de los algoritmos heurísticos vemos que la búsqueda local (*BL*) es el peor de ellos en términos de la función objetivo (Agr.) en la gran mayoría de los casos. Esto puede deberse a que el espacio que explora la *BL* es más restringido que aquel que exploran el resto de algoritmos, ya que por un lado el algoritmo *ES* realiza una gran exploración en sus inicios (gracias a una temperatura alta) y los algoritmos basados en trayectorias múltiples realizan cierta exploración cada vez que se reinician (ya sea empezando desde una nueva solución aleatoria como la *BMB*, o desde una solución fuertemente mutada como en *ILS* o *ILS-ES*).

El segundo algoritmo que parece ser peor en términos del agregado es la búsqueda multiarranque básica (*BMB*). Concretamente esta obtiene peores agregados cuando trabajamos con el 10% de restricciones y vemos que suele obtener de forma notable peor Tasa_inf que los otros tres algoritmos (*ES*, *ILS* e *ILS-ES*). Su efectividad parece cambiar cuando trabajamos con el 20% de las restricciones, y es que si miramos la tabla del 20% de restricciones, no parece que haya un segundo algoritmo claramente peor (vemos que la *BMB* incluso obtiene el mejor agregado con el dataset *Newthyroid*, aunque no por mucha diferencia). Como ya hemos mencionado anteriormente, cuando le damos más restricciones a un algoritmo lo que estamos haciendo es darle parte de la solución; esta información parece tener un efecto bastante positivo para la *BMB* en concreto.

Respecto a los algoritmos *ES*, *ILS* e *ILS-ES*, vemos que no hay un claro ganador. En los datasets pequeños (*Iris* y *Rand*) sus resultados son igual de buenos en términos del agregado. En los dos datasets más grandes las diferencias tampoco son muy grandes; podemos ver que en *Ecoli*, el algoritmo *ES* obtiene el mejor agregado en ambas tablas (con 10% y 20% de restricciones), mientras que en *Newthyroid* es *ILS-ES* el ganador cuando se trabaja con el 10% de restricciones e *ILS* cuando se trabaja con el 20%. Así, vemos que un algoritmo basado en trayectorias simples como es el *ES* puede tener una efectividad similar a la de algoritmos basados en trayectorias múltiples como *ILS-ES* (que ejecuta varias veces *ES* en su interior) cuando ambos algoritmos tienen el mismo límite de llamadas a la función objetivo (100000 en nuestro caso).

Centrándonos en comparar los algoritmos *BL*, *BMB* y *ILS* (algoritmos que usan la búsqueda local), esta vez es claro que los algoritmos basados en trayectorias múltiples (*BMB* e *ILS*) han superado al algoritmo basado en trayectorias simples que utilizan (*BL*). Además, *ILS* obtiene mejores agregados en general que *BMB*, lo que podría deberse a que sus reinicios están en cierta forma mejor dirigidos que los reinicios de *BMB*, pues *ILS* parte de una mutación de la mejor solución encontrada hasta el momento en cada reinicio, mientras que *BMB* genera una nueva solución aleatoria en cada reinicio.

Análisis: Tiempos de ejecución

Profundizando finalmente en los tiempos de ejecución, las tablas reflejan claramente que estos son mucho menores en los algortimos *COPKM* y *BL* (que son también los que peores agregados obtienen). Además, el *COPKM* es el que tiene (siempre que no cicle de forma infinita) mejores tiempos de ejecución.

El resto de algoritmos tienen tiempos de ejecución que son bastante altos y del todo apreciables. Cabe señalar que esto se podría deber en parte 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. Aun así, se puede ver que el algoritmo que tiene peores tiempos de ejecución en general es el *BMB*. Esto se podría deber de nuevo a su punto de partida en cada reinicio. Empezar desde una nueva solución aleatoria en cada búsqueda local interna podría causar que tarde en converger y gaste cerca de las 10000 evaluaciones cada vez, mientras que por ejemplo *ILS*, al comenzar cada búsqueda local desde una solución de calidad mutada, podría converger más rápido y ahorrar tiempo en cada búsqueda local.

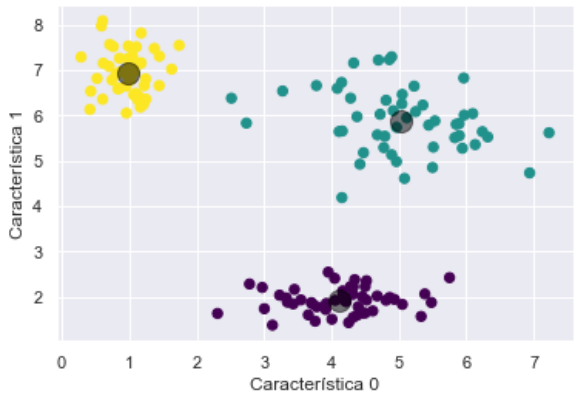
Comparando los tiempos de ejecución de *ILS* e *ILS-ES*, es claro que los tiempos de *ILS* son menores en general. Esto se podría deber a que cada *ES* ejecutado dentro de *ILS-ES* va a tener una fase de exploración de la que carece cada *BL* ejecutada dentro de *ILS*, por lo que cada *BL* podría converger de forma más rápida. Finalmente, los tiempos del algoritmo *ES* (de forma individual) parecen ser los terceros mejores (tras *COPKM* y *BL*), sobre todo si miramos los tiempos de ejecución en el dataset de mayor tamaño (*Ecoli*).

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 **ES** en el dataset **Rand** con el 10% de las restricciones:

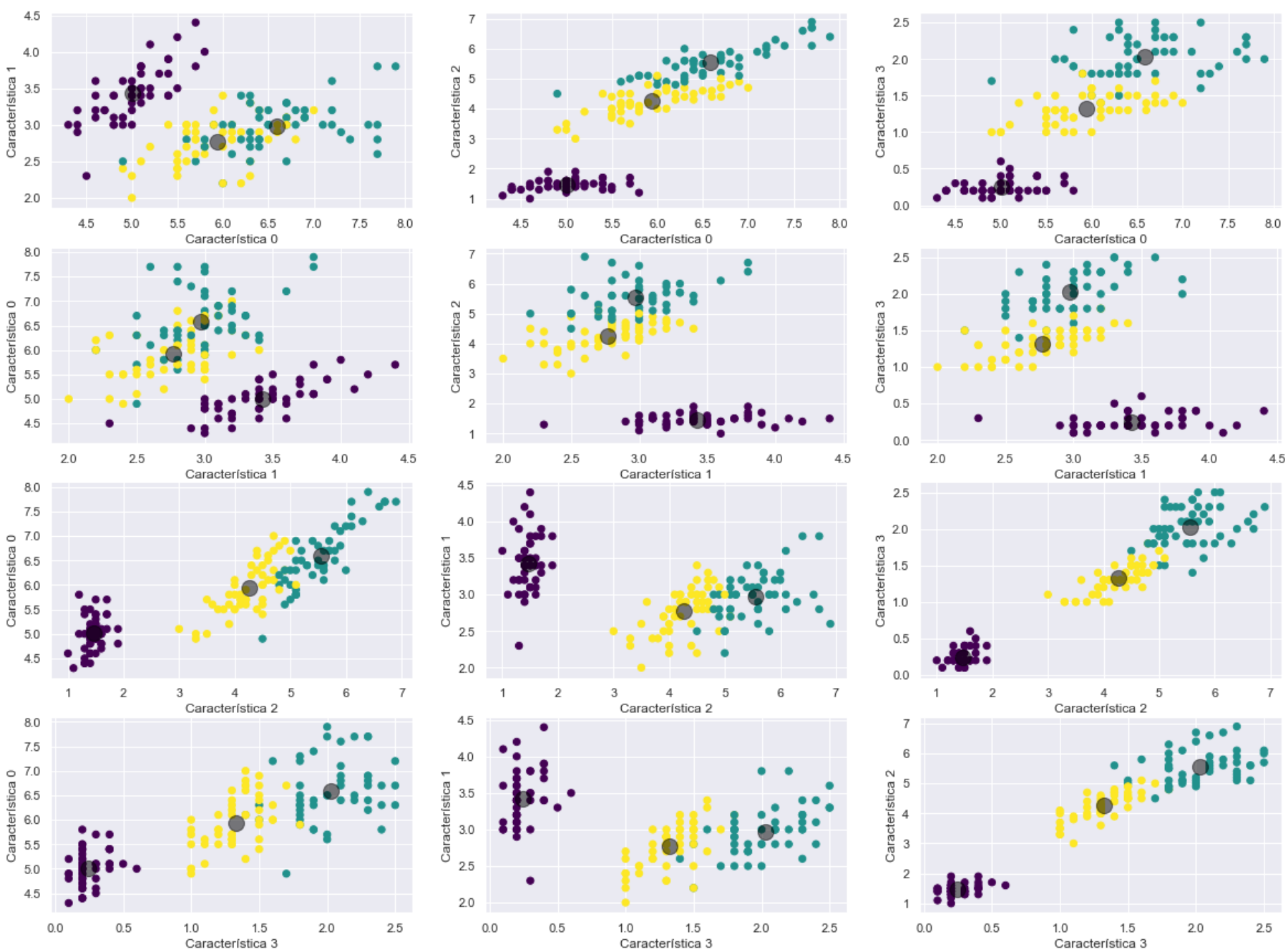
In [17]:
```
#Leemos los datos
X = read_dat("./data/rand_set.dat")
const_matrix = read_constraints_matrix("./data/rand_set_const_10.const")
const_list = constraints_matrix_to_list(const_matrix)
#Fijamos semilla
np.random.seed(0)
#Ejecutamos el algoritmo:
partition_sol = simulated_annealing_algo(X, const_matrix, const_list, k=3)
# Visualizamos la solución obtenida
visualise_rand_clusters(X, partition_sol)
```



Representación de la partición solución obtenida por el algoritmo **ILS** en el dataset **Iris** con el 10% de las restricciones:

In [18]:
```
#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 = iterated_local_search(X, const_matrix, const_list, k=3)
# Visualizamos la solución obtenida
visualise_iris_clusters(X, partition_sol)
```

Representamos las instancias y los clusters según las diferentes características (2D)



6.- Extra: Hibridación de ILS con la Búsqueda Local Suave

(*ILS-BLS*)

Como pequeña tarea extra he decidido hibridar la búsqueda local reiterada con la búsqueda local suave que implementamos en la práctica anterior. Esto conecta con la parte extra de mi práctica 2, en la que comparé el uso de la búsqueda local y el de la búsqueda local suave dentro de un algoritmo memético y comprobé (como era esperable) que el número de búsquedas locales suaves que se pueden ejecutar de forma completa con 100000 evaluaciones de la función objetivo es mucho mayor que el número de búsquedas locales (no suaves) que podemos ejecutar, pues cada *BL* consume de media muchas más evaluaciones que cada *BLS*. Este hecho justificaba 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). Esta vez cambiamos la *BL* usada dentro de *ILS* por la *BLS* para dar lugar al algoritmo que hemos llamado *ILS-BLS*. Este cambio le dará más peso a las mutaciones que ejecuta la búsqueda local reiterada (ejecuta una tras cada búsqueda local).

Para una información más detallada sobre el algoritmo de búsqueda local suave se hace referencia a la memoria de la práctica 2, en donde se explica y se describe en pseudocódigo. A continuación, se hacen unas pequeñas modificaciones en los argumentos y las salidas del algoritmo de búsqueda local suave original (el de la práctica 2) para que pueda sustituir al algoritmo de búsqueda local usado dentro de *ILS*. Además se añaden algunas variables en el algoritmo de *ILS-BLS* para monitorizar el número de llamadas a la función objetivo que realiza el nuevo algoritmo.

Finalmente, ejecutamos los algoritmos *ILS* e *ILS-BLS* sobre el mismo conjunto de datos (*Ecoli* con 20% de restricciones).

In [19]:
```
##Búsqueda local suave
def smooth_local_search_extra(X, const_list, partition, current_func_value,
                              assignations_counter, k, max_failures, counter, lambda_, max_evaluations):
    shuffle_indices = list(range(len(partition)))
    random.shuffle(shuffle_indices)
    failures = 0
    improvement = True
    best_func_value = current_func_value
    internal_counter = 0
    i = 0
    while (improvement or (failures < max_failures)) and i < len(partition) and internal_counter < max_evaluations:
        improvement = False
        #Asignar el mejor cluster posible al gen shuffle_indices[i]
        if assignations_counter[partition[shuffle_indices[i]]] > 1: #Si cambiar este gen no rompe una restricción fuerte...
            best_cluster = partition[shuffle_indices[i]]
            j = 0
            while j < k and internal_counter < max_evaluations:
                if j != partition[shuffle_indices[i]]:
                    partition[shuffle_indices[i]] = j
                    func_value = objective_func(X, partition, const_list, centroids = None, lambda_ = lambda_)
                    counter += 1
                    internal_counter += 1
                    if func_value < best_func_value:
                        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
                        partition[shuffle_indices[i]] = best_cluster
                j += 1
        if improvement == False:
            failures += 1
        i += 1
    return partition, best_func_value, assignations_counter, counter
```

In [20]:
```
#Algoritmo ILS-BLS
def iterated_local_search_extra(X, const_matrix, const_list, k, lambda_ = None, n_ls = 250, evaluations_per_ls = 400,
                          search_algo = smooth_local_search_extra):
    n_instances = X.shape[0]
    max_failures = int(0.1 * n_instances)
    if lambda_ == None:
        lambda_ = max_dist(X) / len(const_list)
    best_partition, assignations_counter = generate_valid_initial_sol(X, k)
    best_func_value = objective_func(X, best_partition, const_list, centroids = None, lambda_ = lambda_)
    counter = 1
    best_partition, best_func_value, assignations_counter, counter = search_algo(X, const_list, best_partition,
                                                                                 best_func_value, assignations_counter, k,
                                                                                 max_failures, counter, lambda_,
                                                                                 max_evaluations = evaluations_per_ls)
    print(counter, end="\r", flush=True)
    for i in range(n_ls-1):
        candidate_partition = segment_mutation_operator(best_partition, segment_length = int(n_instances * 0.1), k=k)
        unique, counts = np.unique(candidate_partition, return_counts=True)
        assignations_counter = dict(zip(unique, counts))
        if len(unique) != k:
            repair_partition(candidate_partition, assignations_counter, k)
        func_value = objective_func(X, candidate_partition, const_list, centroids = None, lambda_ = lambda_)
        candidate_partition, func_value, assignations_counter, counter = search_algo(X, const_list, candidate_partition,
                                                                                     func_value, assignations_counter, k,
                                                                                     max_failures, counter, lambda_,
                                                                                     max_evaluations = evaluations_per_ls)
        print(counter, end="\r", flush=True)
        if func_value < best_func_value: #Criterio de aceptación el mejor.
            best_func_value = func_value
            best_partition = candidate_partition
    print("Número total de evaluaciones de la función objetivo: ", counter)
    return best_partition
```

Cabe comentar que dentro de *ILS-BLS* ejecutamos `n_ls = 250` búsquedas locales suaves con un límite máximo de `evaluations_per_ls = 400` evaluaciones de la función objetivo cada una (mientras que en el *ILS* original, ejecutabamos `n_ls = 10` búsquedas locales con un límite de `evaluations_per_ls = 10000` evaluaciones cada una).

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

In [21]:
```
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 la búsqueda local reiterada hibridada con la búsqueda local suave (*ILS-BLS*).

In [22]:
```
np.random.seed(0)
partition_sol1 = iterated_local_search_extra(X, const_matrix, const_list, k=8)
```

```
Número total de evaluaciones de la función objetivo:  95593
```

Ejecutamos el algoritmo *ILS* original, que usa la búsqueda local (*BL*).

In [23]:
```
np.random.seed(0)
partition_sol2 = iterated_local_search(X, const_matrix, const_list, k=8)
```

Comparamos la calidad de las respectivas soluciones.

In [24]:
```
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:  175.89325041812933
Desviación general usando BL (no suave):  84.36040522126567
```

In [25]:
```
print("Infactibilidad usando BLS: ", infeasibility(partition_sol1, const_list))
print("Infactibilidad usando BL (no suave): ", infeasibility(partition_sol2, const_list))
```

```
Infactibilidad usando BLS:  3152
Infactibilidad usando BL (no suave):  1731
```

In [26]:
```
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:  218.17661598311406
Agregado usando BL (no suave):  107.58137787132549
```

Vemos que los resultados obtenidos por esta nueva hibridación son bastante pobres (tan solo son comparables a los obtenidos por el Greedy), y se quedan muy por debajo del nivel de los resultados obtenidos por el *ILS* original. La explicación podría ser que al usar la *BLS* hemos roto en cierta medida el equilibrio entre exploración y explotación que tiene el algoritmo original *ILS*. Como ya hemos comentado, la *BLS* necesita menos evaluaciones de la función objetivo y por ello la podemos ejecutar muchas más veces que la *BL*. Así, tenemos que en el nuevo algoritmo *ILS-BLS* le estamos dando más protagonismo a la mutación realizada tras cada búsqueda local suave (al ejecutar más búsquedas locales suaves, ejecutamos también más mutaciones) y menos protagonismo a la propia explotación que realiza cada búsqueda local. Recordamos que el operador de mutación usado es el de mutación por segmento, el cual es bastante fuerte (en el sentido de que cambia bastante la solución mutada). Por tanto lo que hemos hecho es aumentar en gran medida la exploración de este algoritmo a costa de disminuir su explotación. Los resultados parecen dejar claro que este cambio ha sido a peor (incluso aunque esta comparación no es del todo determinante al basarse en una sola ejecución en un solo conjunto de datos, en las tablas de la sección anterior podemos comprobar que el *ILS* obtuvo en todas sus ejecuciones un resultado mucho mejor en términos del agregado).

In [ ]:
```
-
```