Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

Práctica 1: Técnicas de Búsqueda Local y Algoritmos Greedy

Problema del Agrupamiento con Restricciones
Algoritmo greedy COPKM y Algoritmo de Búsqueda Local

Juan Emilio Martínez Manjón
77024428-G
juanemartinez@correo.ugr.es
Grupo 2 Jueves 17:30h

2019-2020

Índice

1 Descripción del problema

El problema escogido para resolver esta práctica se trata del problema del agrupamiento con restricciones (PAR).

Este problema se resume a grandes rasgos en agrupar datos en varios conjuntos dependiendo de las similitudes entre ellos. Estos conjuntos se llaman clusters, por lo que a este problema se le denomina también clustering.

Cada cluster está representado en el espacio por un centroide, el cual tiene tantas instancias como instancias tengan los datos del conjunto en cuestión. Cada dato está asignado a un centroide dependiendo de la distancia Euclidea a él. Por lo que el objetivo del problema del agrupamiento clásico era asignar cada dato a aquel cluster cuya distancia Euclidea a su centroide sea menor.

El problema PAR añade a esta ecuación las restricciones. Por lo que a parte de asignar un dato a un cluster disminuyendo la desviación general de las asignaciones, también deberemos tener en cuenta una serie de restricciones. Estas restricciones pueden ser del tipo Must Link (ML) que indican que dos datos deben pertenecer al mismo centroide. O Cannot Link (CL) que indican que dos datos deben pertenecer a centroides diferentes.

También podemos distinguir entre dos niveles de restricciones, las fuertes y las débiles. Las restricciones fuertes son las que debemos cumplir en todos los casos. Las restricciones débiles son aquellas que podemos cumplir o no. Y, en el caso de no cumplirlas, produciría un aumento en el grado de infactibilidad.

2 Descripción de la parte común de los algoritmos

A continuación se van a mostrar que forma de representación común podemos encontrar en ambos algoritmos. Además de mostrar pseudocódigo acerca de los operadores comunes y la forma de calcular la función objetivo.

2.1 Representación de los datos

Los centroides están representados como instancias de las clase Centroide. Dicha clase tiene la siguiente estructura:

```
class Centroide
        int numero
        vector datos asignados
        vector indices
```

En esta representación "numero" hace referencia a la etiqueta de dicho centroide, ya que cada conjunto de datos tiene varios centroides. El vector llamado "datos asignados" contiene todos los indices asignados a dicho centroide. El vector llamado "indices" contiene los valores de los parámetros de dicho centroide.

Además de los getters y setters, esta clase nos permite saber si un dato cualquiera está contenido en ese centroide.

Los datos están representados en un vector de listas, donde cada elemento de la lista corresponde a una instancia del dato. Este vector se rellena leyendo un fichero mediante un flujo de entrada. La función de rellenado de datos es común para varios algoritmos, se basa en leer datos y saltarse las comas almacenando los resultados en un vector de listas como he indicado previamente.

La solución final la representamos en un vector de centroides. Donde imprimiremos que indices del conjunto de datos han sido finalmente asignados a cada cluster después de ejecutar el algoritmo.

Además de representar la solución de esta manera también añadiremos sus correspondientes valores de desviación, infactibilidad y función objetivo. Dichos valores nos servirán para realizar una comparativa entre ambos algoritmos. Se imprimen estos valores tanto en la Búsqueda Local como en el COPKM.

2.2 Operadores comunes

Los operadores que son comunes a ambos algoritmos son los siguientes:

- Actualizar un centroide
- Calcular la distancia máxima del conjunto
- Calcular la desviación
- Calcular la función objetivo

2.2.1 Actualizar un centroide

Para actualizar un centroide se recorrerán todos los datos asignados al centroide en cuestión, y se calculará la media de cada uno de los parámetros. Finalmente se modificarán los parámetros de dicho centroide por los que acabamos de calcular.

El pseudocódigo correspondiente es el siguiente:

Algorithm 1: actualizaCentroide(centroide, data_set)

```
Result: void
Inicializamos un array "medias" con tantas casillas como caracteristicas tengan
 los datos del dataset a 0;
for Cada dato asignado al centroide a actualizar do
    for Cada una de las caracteristicas del dato del bucle superior do
        medias[pos] += caracteristicas[pos];
    end
end
for Cada uno de los elementos en medias do
    medias[pos] = medias[pos] / numero datos asignados;
    indice_centroide[pos] = medias[pos]
end
```

2.2.2 Calcular la distancia máxima del conjunto

Para calcular la distancia máxima del conjunto debemos calcular las distancias dos a dos de todos los elementos del conjunto. Hasta finalmente devolver la mayor de todas.

El pseudocódigo correspondiente es el siguiente:

Algorithm 2: distanciaMaxima(data_set)

```
Result: distancia maxima
Inicializamos una variable distancia_maxima a 0;
for i=0; i < datos totales del dataset do
    for j=i+1; j < datos totales del dataset do
        Creamos el vector auxiliar "resultado";
        for Cada una de las caracteristicas del dato i do
            resultado.push_back((datoj[k]-datoi[k])*(datoj[k]-datoi[k]));
        end
        for Cada uno de los datos del vector resultado do
            distancia += resultado[pos];
        end
        if distancia > distancia_maxima then
            distancia_maxima = distancia;
        end
    end
end
```

2.2.3 Calcular la desviación general

Para calcular la desviación general deberemos calcular la desviación de cada uno de los centroides y sumarlas todas al final. Para calcular la desviación de un centroide deberemos calcular la media de las distancias Euclideas de cada uno de sus datos al propio centroide.

El pseudocódigo correspondiente es el siguiente:

Algorithm 3: calculaDesviacion(centroides, dataset)

```
Result: desviacion
distancia_euclidea = 0.0;
desviacion = 0.0;
sumatoria = 0.0;
for i=0; i < numero centroides do
    sumatoria = 0.0 ;
    for j=0; j < cada uno de los datos asignados al centroide i do
        distancia_euclidea = 0.0;
        distancia_euclidea += diferencias al cuadrado de cada caracteristica del
         dato menos el indice correspondiente del centroide ;
        sumatoria += distancia_euclidea;
    end
    sumatoria = sumatoria / numero de datos del centroide;
    desviacion += sumatoria;
end
desviacion = desviacion / numero_centroides;
```

2.2.4 Calcular la función objetivo

Para calcular la función objetivo necesitamos saber el valor de desviación general, de infactibilidad, y de lambda. Finalmente aplicaremos la formula:

Función = Desviación + (Infactibilidad * Lambda)

El calculo de la infactibilidad es diferente para cada uno de los algoritmos por lo que se explicará detalladamente más adelante. Lambda consiste en una constante resultante de dividir la distancia máxima del dataset entre el número máximo de restricciones (ML + CL).

El pseudocódigo correspondiente es el siguiente:

Algorithm 4: calculaFuncionObjetivo(centroides, dataset, restricciones, lambda)

```
Result: funcion_objetivo
desviacion = calculaDesviacion(centroides, dataset);
infactibilidad = calculaInfactibilidad(restricciones, dataset);
funcion_objetivo = desviacion + (infactibilidad * lambda);
```

3 Implementación y desarrollo de cada algoritmo

A continuación se van a detallar las estructuras y operador relevantes de cada algoritmo, así como la implementación de la búsqueda en cada uno de ellos.

En el caso de la búsqueda local (BL) tambíen se incluirá, en el proceso de búsqueda; la descripción en pseudocódigo del método de exploración del entorno, el operador de generación de vecino y la generación de soluciones aleatorias empleadas.

3.1 Algoritmo COPKM (Greedy)

Antes de mostrar los pseudocódigos de la parte específica de este algoritmo es necesario explicar la forma en la que se guardan las restricciones. En este caso están guardadas en un vector de vectores de enteros, que lo podemos traducir en una matriz de enteros. La forma de rellenar dicha matriz es leyendo de un fichero los diferentes datos e ir metiendolos en la matriz.

3.1.1 Cálculo de infactibilidad

Este cálculo se realiza recorriendo todos los centroides y mirando cuantas restricciones CL y ML se incumplen en ellos, indexando en la matriz de restricciones.

Algorithm 5: calculaInfactibilidad(centroides, restricciones)

```
Result: infactibilidad
infactibilidad = 0;
for i=0; i < numero de centroides do
    for j=0; j < datos asignados al centroide i do
        for k=j+1; k < datos asignados al centroide i do
            if restricciones[dato j][dato k] == CL then
                infactibilidad++;
            end
        end
        for k=i+1; k < numero de centroides do
            for l=0; l < datos asignados al centroide k do
                if restricciones[dato j][dato l] == ML then
                    infactibilidad++;
                end
            end
        end
    end
end
```

3.1.2 Calcular distancia de dato a centroide

Esta función calcula la distancia que hay desde un dato del dataset pasado como parámetro al centroide también pasado como parámetro.

Algorithm 6: distanciaCentroide(caracteristicas, centroide)

```
Result: distancia
distancia = 0.0;
for i=0; i < cada una de las características pasadas como parámetro do
    distancia += diferencias al cuadrado de cada caracteristica del dato menos
     el indice correspondiente del centroide ;
end
```

3.1.3 Comprobar si dos asignaciones de centroides son iguales

Esta función comprueba si dos vectores de centroides son iguales, es decir, tienen las mismas asignaciones de índices.

Algorithm 7: esIgual(centroides1, centroides2

```
Result: boolean son_iguales
son_iguales = true;
for i=0; i < numero de elementos en centroides1 do
    if numero de elementos asignados a centroides1i != numero de elementos
     asignados a centroides2i then
        son_iguales = false;
    else
        if Alguno de los elementos asignados a centroides1i no coincide con
         centroides2i then
            son_iguales = false;
        end
    end
end
```

3.1.4 Proceso de búsqueda

Esta función es la principal del algoritmo COPKM. Su funcionamiento se basa en ir recorriendo todos los indices del dataset y ver con qué centroide obtiene menos infactibilidad. En el caso de que coincidan, nos quedamos con aquel centroide cuya distancia a él sea

menor. Este proceso se repite hasta que no se produzca ningún cambio en el vector de centroides.

Algorithm 8: procesa(data_set, indices, restricciones, centroides, lambda)

```
Result: void
hay_cambio = true;
while hay_cambio do
    hay_cambio = false;
    for i=0; i < numero de indices do
        mejor_infactibilidad = Inf;
        mejor_distancia = Inf;
        for j=0; j < numero de centroides do
            distancia = distanciaCentroide(indice i, centroide j);
            infactibilidad = calculaInfactibilidad(centroide j, restricciones);
            if infactibilidad < mejor_infactibilidad or (infactibilidad ==
              mejor_infactibilidad and distancia < mejor_distancia) then
                mejor_infactibilidad = infactibilidad;
                mejor_distancia = distancia;
                cluster_elegido = centroide j;
            end
        end
        Aniadir a cluster elegido el indice i;
    end
    if !esIgual(nueva_asignacion, antigua_asignacion) then
        hay_cambio = true;
    end
    if hay_cambio then
        actualizamos todos los centroides;
    end
end
```

3.2 Algoritmo de búsqueda local (BL)

En este algoritmo, al contrario que el COPKM, las restricciones se guardan en un vector de structs. Esto se hace así porque se realizan muchas más llamadas al método calculaInfactibilidad que en el otro algoritmo, y el método anterior es demasiado ineficiente como para llamarlo tantas veces.

El struct tendría los siguientes atributos:

```
struct Restriccion
        int indice1
        int indice2
        int valor
```

Para rellenar el vector de restricciones, se crea una instancia de este struct por cada restricción ML o CL. En el atributo indice1 se guarda la posición de la fila y en el atributo indice2, la posición de la columna. En valor se guarda el tipo de restricción que obtendríamos al indexar dicha fila y columna en la matriz.

Además, tenemos un vector donde en cada casilla ponemos el centroide al que asignamos el indice de dicha casilla. Este vector lo llenamos de forma aleatoria.

3.2.1 Cálculo de infactibilidad

Esta función calcula la infactibilidad de un conjunto de indices, usando la codificación de struct de restricciones.

Algorithm 9: calculaInfactibilidad(indices, restricciones)

```
Result: infactibilidad
infactibilidad = 0;
for i=0; i < cada una de las instancias del vector de restricciones do
    if restricciones[i].valor == ML and indices[restricciones[i].indice1] !=
     indices[restricciones[i].indice2] then
        infactibilidad++;
    end
    if restricciones[i].valor == CL and indices[restricciones[i].indice1] ==
     indices[restricciones[i].indice2] then
        infactibilidad++;
    end
end
```

3.2.2 Comprobar si la asignación de centroides es válida

Esta función comprueba si las asignaciones a centroides del vector de asignaciones es correcta. Es decir, que no se quede ningún centroide vacío. Esta función es necesaria porque las asignaciones se hacen al comienzo de forma aleatoria.

Algorithm 10: compruebaIndicesCorrectos(indices)

Result: boolean es_valida
es_valida = true;
Inicializamos un array de booleanos llamado comprobaciones a false;
for *cada uno de los indices* **do**
  comprobaciones[indices i] = true;
end
Si al recorrer el array de comprobaciones, todas las posiciones son true,
devolvemos true. Si alguna de las posiciones es false, devolvemos false.

3.2.3 Proceso de búsqueda

Esta es la función principal del algoritmo de Búsqueda Local. En ella generaremos un vector de pairs que actuará como nuestro vecindario virtual. Lo barajaremos e iremos cambiando valores de nuestro vector de índices por los de nuestro vecindario. Una solución será mejor que la actual si su función objetivo es menor. El proceso terminará cuando se recorra el vector de índices y no se cambie ningún valor por alguno de sus vecinos.

Algorithm 11: procesa(dataset, indices, restricciones, centroides, lambda, semilla)

Result: void
mejor_valor = calculaFuncionObjetivo(indices, centroides, dataset, restricciones);
contador = 0;
Metemos en un vector de pairs de dos enteros todos los posibles cambios que se podrían hacer en el vector de indices.
Barajamos de forma aleatoria el vector de pairs usando la semilla.
for *i=0; i < numero de elementos en el vector de pairs* **do**
    vector_auxiliar = indices;
    vector_auxiliar[vector_pairs[i].first] = vector_pairs[i].second;
    contador++;
    if *Los indices del vector auxiliar son correctos* **then**
        podemos calcular la funcion objetivo
    end
    if *Podemos calcular la funcion objetivo* **then**
        Metemos en un vector auxiliar de centroides la información de nuestros centroides actuales por si es necesario restaurarlos.
        Rellenamos y actualizamos nuestros centroides con los indices del vector auxiliar.
        if *FuncionObjetivo(nuevos_centroides) < mejor_valor* **then**
            mejor_valor = FuncionObjetivo(nuevos_centroides);
            indices = vector_auxiliar;
            Reiniciamos el for principal usando un booleano o poniendo su indice i a 0.
            Volvemos a rellenar el vector de pairs de dos enteros con los nuevos posibles cambios que se podrían hacer en el vector de índices y lo barajamos, usando de semilla el contador actual.
        else
            Restauramos el valor de nuestros centroides, que habiamos guardado en un vector auxiliar.
        end
    end
    if *contador == 100000* **then**
        Salimos del bucle for más externo y nos quedamos con la mejor solución que hayamos sacado hasta ese momento.
    end
end

4 Desarrollo de la práctica

4.1 Procedimiento seguido para desarrollar la práctica

La práctica se ha desarrollado en el lenguaje de programación C++ sin usar ningún tipo de framework de metaheurísticas. La programación de la práctica ha sido realizada al completo de forma manual, a excepción de los métodos del fichero "random.h", que pertenecen a la web de la asignatura.

Se han programado dos ficheros cpp, uno para cada algoritmo, donde en cada ejecución podremos ver los resultados de los tres datasets de esta práctica. Esto lo he hecho así para poder facilitar la visualización de datos sin necesidad de realizar muchas ejecuciones.

Se comenzó la práctica creando los contenedores necesarios y diseñando los algoritmos de rellenado de estos. Después se fueron creando todas las funciones necesarias para que el proceso de búsqueda principal quedase lo más legible posible.

En ambos ficheros, antes de llamar al proceso de búsqueda, se cargan los vectores de datos y restricciones de todos los datasets con la información leída desde un flujo de entrada.

4.2 Manual de usuario

Para ejecutar los programas pertinentes es necesario usar la orden "make" para compilar los ficheros del makefile que adjunto en la práctica.

Para ejecutar cualquiera de los programas, es necesario usar la siguiente sintaxis:

```
./nombre_programa <semilla> <porcentaje_restricciones>
```

El nombre del programa puede ser o "BL" si se quiere ejecutar la Búsqueda Local, o "COPKM" si se quiere ejecutar el algoritmo k-medias con restricciones.

La semilla puede ser cualquier número al azar

El porcentaje de restricciones es un número que puede ser o 10 o 20.

5 Experimentos y análisis de resultados

Se han realizado un total de 20 ejecuciones divididas de la siguiente forma:

```
5 ejecuciones BL con 10% de restricciones
5 ejecuciones COPKM con 10% de restricciones
5 ejecuciones BL con 20% de restricciones
5 ejecuciones COPKM con 20% de restricciones
```

Cada una de las ejecuciones anteriormente descritas nos devuelve el valor de "Desviacion", "Infactibilidad", "Función objetivo" y "Tiempo de ejecución" de los tres datasets.

Las semillas que se han usado para las 5 ejecuciones son las siguientes:

- 545
- 650
- 17
- 1010
- 1234

Se utiliza la misma semilla para la misma ejecución de ambos algoritmos, es decir, semilla 545 para la ejecución 1 tanto de COPKM como de BL. Esto se hace así para que se puedan comparar los resultados.

5.1 Tablas de resultados y análisis de las mismas

Resultados obtenidos por el algoritmo COPKM en el PAR con un 10 por ciento de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T(s)	*Tasa_C*	*Tasa_inf*	*Agr.*	T(s)	*Tasa_C*	*Tasa_inf*	*Agr.*	T(s)
Ejecución 1: Seed 545	0,98	104,00	1,64	0,13	39,87	42,00	41,00	4,51	0,76	0,00	0,76	0,26
Ejecución 2: Seed 650	0,67	0,00	0,67	0,10	33,79	33,00	34,67	5,26	0,76	0,00	0,76	0,28
Ejecución 3: Seed 17	0,67	0,00	0,67	0,52	34,95	7,00	35,14	5,46	0,76	0,00	0,76	0,09
Ejecución 4: Seed 1010	0,67	20,00	0,80	0,24	32,25	4,00	32,36	4,70	0,76	0,00	0,76	0,42
Ejecuión 5: Seed 1234	0,67	0,00	0,67	0,54	32,04	32,00	32,90	6,65	0,76	0,00	0,76	0,12
Media	0,73	24,80	0,89	0,31	34,58	23,60	35,21	5,32	0,76	0,00	0,76	0,24

Resultados obtenidos por el algoritmo COPKM en el PAR con un 20 por ciento de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T(s)	*Tasa_C*	*Tasa_inf*	*Agr.*	T(s)	*Tasa_C*	*Tasa_inf*	*Agr.*	T(s)
Ejecución 1: Seed 545	0,67	0,00	0,67	0,10	31,63	0,00	31,63	3,55	0,76	0,00	0,76	0,17
Ejecución 2: Seed 650	0,67	0,00	0,67	0,07	33,46	0,00	33,46	4,47	0,76	0,00	0,76	0,15
Ejecución 3: Seed 17	0,67	0,00	0,67	0,54	29,61	0,00	29,61	5,86	0,76	0,00	0,76	0,07
Ejecución 4: Seed 1010	0,67	0,00	0,67	0,13	28,40	0,00	28,40	3,71	0,76	0,00	0,76	0,14
Ejecuión 5: Seed 1234	0,67	0,00	0,67	0,56	25,49	0,00	25,49	2,39	0,76	0,00	0,76	0,08
Media	0,67	0,00	0,67	0,28	29,72	0,00	29,72	4,00	0,76	0,00	0,76	0,12

Resultados obtenidos por el algoritmo BL en el PAR con un 10 por ciento de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T(s)	*Tasa_C*	*Tasa_inf*	*Agr.*	T(s)	*Tasa_C*	*Tasa_inf*	*Agr.*	T(s)
Ejecución 1: Seed 545	0,67	0,00	0,67	1,12	23,83	72,00	25,76	40,36	0,76	0,00	0,76	0,64
Ejecución 2: Seed 650	0,67	0,00	0,67	1,07	23,42	92,00	25,89	32,32	0,77	0,00	0,77	0,77
Ejecución 3: Seed 17	0,67	0,00	0,67	2,43	23,16	108,00	26,06	48,00	0,76	0,00	0,76	0,89
Ejecución 4: Seed 1010	0,67	20,00	0,67	1,19	24,03	77,00	26,10	41,60	0,77	0,00	0,77	0,89
Ejecuión 5: Seed 1234	0,67	0,00	0,67	1,11	21,75	142,00	25,56	42,76	0,76	0,00	0,76	0,87
Media	0,67	4,00	0,67	1,38	23,24	98,20	25,87	41,01	0,76	0,00	0,76	0,81

Resultados obtenidos por el algoritmo BL en el PAR con un 20 por ciento de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T(s)	*Tasa_C*	*Tasa_inf*	*Agr.*	T(s)	*Tasa_C*	*Tasa_inf*	*Agr.*	T(s)
Ejecución 1: Seed 545	0,68	0,00	0,68	0,95	22,71	204,00	25,44	40,43	0,76	0,00	0,76	0,70
Ejecución 2: Seed 650	0,67	0,00	0,67	1,13	24,33	112,00	25,83	63,06	0,77	0,00	0,77	0,75
Ejecución 3: Seed 17	0,67	0,00	0,67	1,01	24,68	97,00	25,98	43,49	0,76	0,00	0,76	0,76
Ejecución 4: Seed 1010	0,67	0,00	0,67	1,26	23,99	180,00	26,41	47,48	0,77	0,00	0,77	0,89
Ejecuión 5: Seed 1234	0,67	0,00	0,67	0,98	23,23	150,00	25,24	46,18	0,76	0,00	0,76	0,78
Media	0,67	0,00	0,67	1,07	23,79	148,60	25,78	48,13	0,76	0,00	0,76	0,78

Resultados globales obtenidos con un 10 por ciento de restricciones

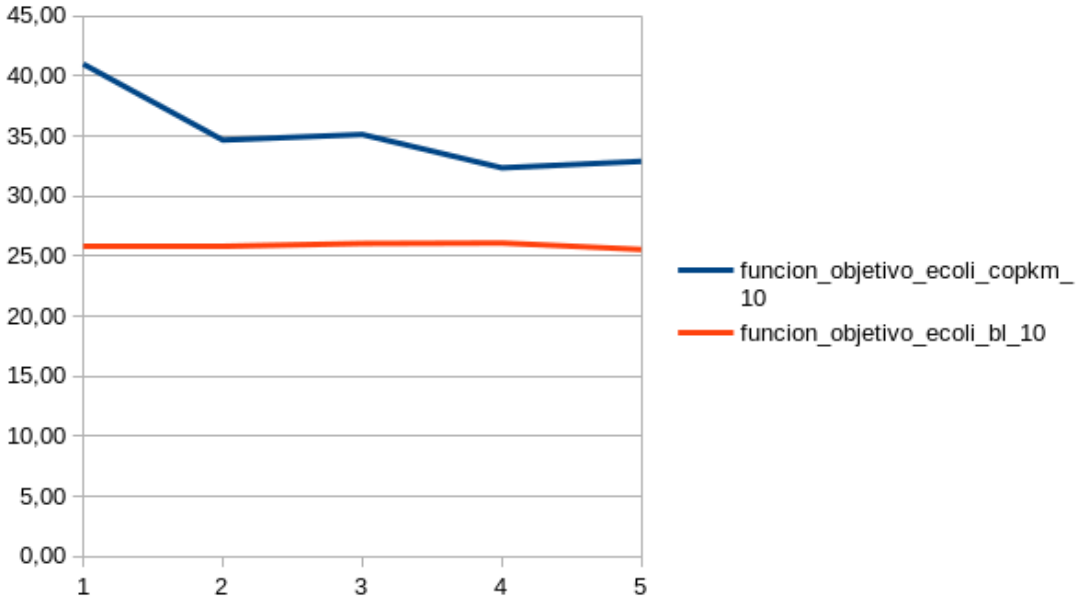
	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
COPKM	0,73	24,80	0,89	0,31	34,58	23,60	35,21	5,32	0,76	0,00	0,76	0,24
BL	0,67	4,00	0,67	1,38	23,24	98,20	25,87	41,01	0,76	0,00	0,76	0,81

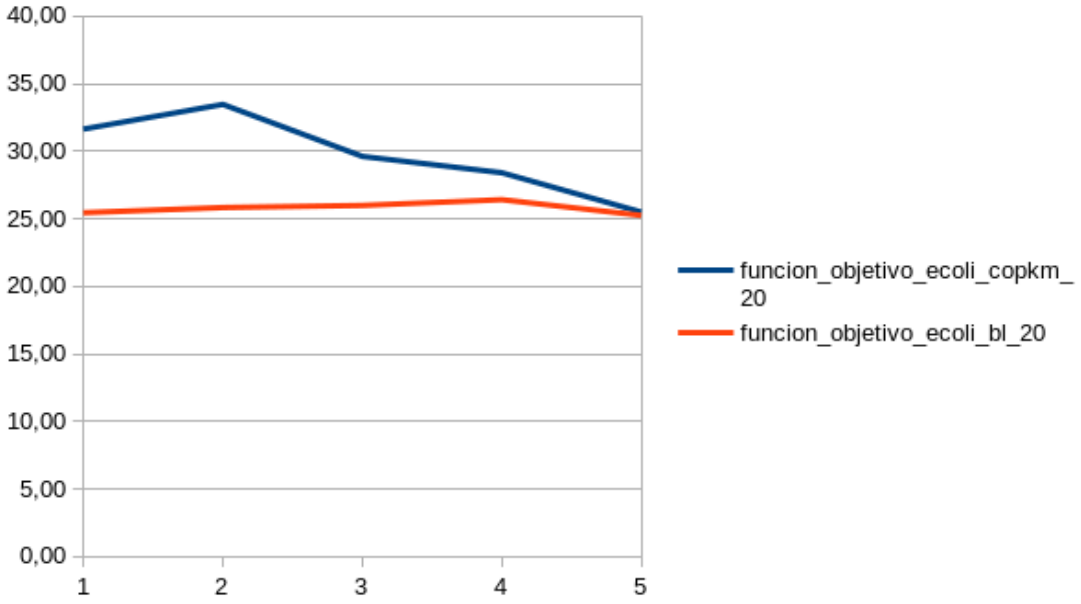
Resultados globales obtenidos con un 20 por ciento de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
COPKM	0,67	0,00	0,67	0,28	29,72	0,00	29,72	4,00	0,76	0,00	0,76	0,12
BL	0,67	0,00	0,67	1,07	23,79	148,60	25,78	48,13	0,76	0,00	0,76	0,78

Como podemos apreciar en las tablas, el algoritmo de búsqueda local tarda bastante más que el algoritmo greedy para datasets con un gran número de datos y clusters como puede ser Ecoli. Pero, este exceso de tiempo se ve recompensado con una gran mejora en la desviación general y por tanto, en la función objetivo. Cuanto mejor sea nuestra desviación, mas íntegro y robusto es el algoritmo.

A continuación se mostraran unas gráficas para ilustrar la mejora de BL en los resultados con respecto a COPKM, usando el dataset Ecoli como ejemplo.





Al observar las gráficas podemos ver como efectivamente BL obtiene unos mejores valores de función objetivo que COPKM aunque, como las restricciones están cogidas al azar y sin ningún criterio, podemos dar con resultados donde la infactibilidad dependa mucho de la semilla usada. Es por esto que en la segunda gráfica la función objetivo de COPKM se acerca mucho a la de BL.

Además, al ejecutar BL usando los datasets Iris y Rand, las asignaciones con clusters converge la gran mayoría de las veces con la solución óptima. Esto pasa en estos datasets porque el número de clusters es bastante reducido. En cambio, para que pase esto con Ecoli hay que probar varias semillas hasta dar con una buena.

A continuación adjunto un scatter plot para ilustrar cuál es la solución que obtiene BL usando el dataset Iris. Se podrá apreciar que las asignaciones a clusters son perfectas.