PRÁCTICA 1

TÉCNICAS DE BÚSQUEDA LOCAL Y ALGORITMOS GREEDY

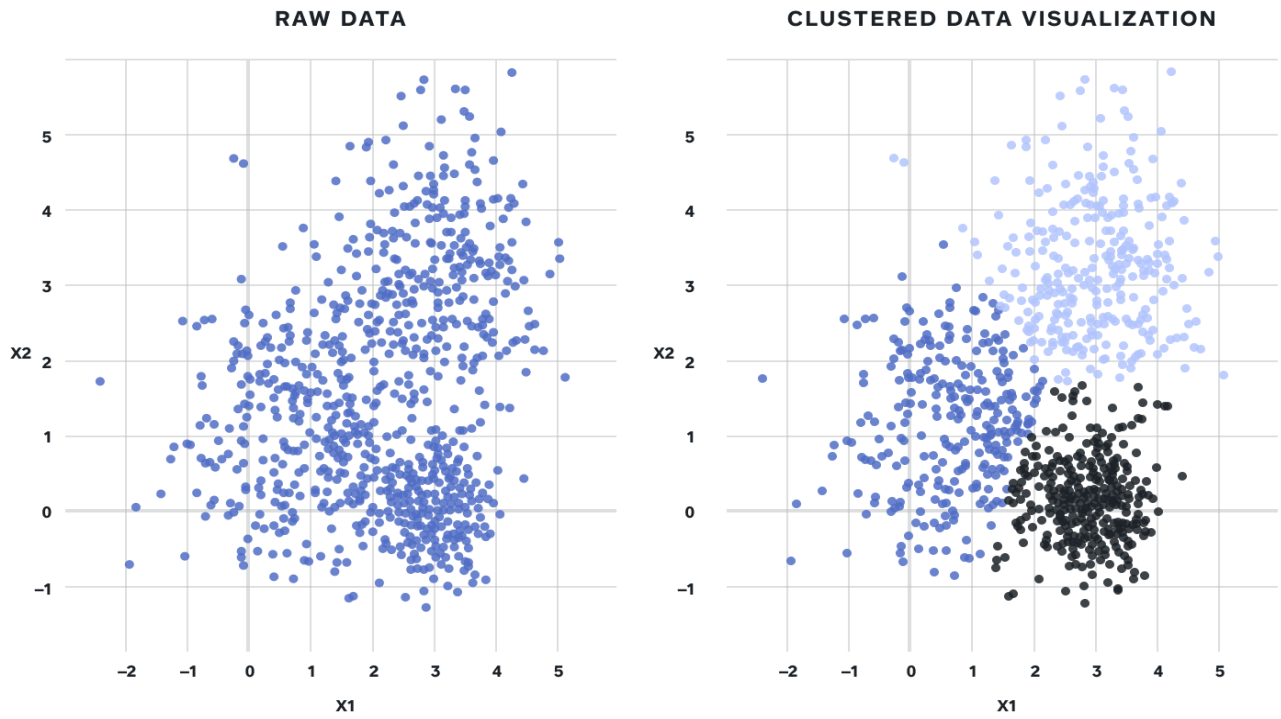
PARA EL PROBLEMA DE AGRUPAMIENTO CON RESTRICCIONES (PAR)

Antonio José Blánquez Pérez
3ºCSI METAHEURÍSTICAS
45926869D
ajose99bp@correo.ugr.com
Grupo 2 – Jueves

ÍNDICE

1. DESCRIPCIÓN DEL PROBLEMA

El agrupamiento o clustering busca clasificar objetos de un conjunto en subconjuntos o clusters a través de sus posibles similaridades. Es una técnica de aprendizaje no supervisado que permite descubrir grupos inicialmente desconocidos o agrupar objetos similares, de manera que podemos encontrar patrones en grandes grupos de datos que serían imposibles(o extremadamente difíciles) de encontrar sin esta técnica. Una muestra de agrupamiento sería la siguiente:



Un ejemplo cotidiano del problema del agrupamiento sería dividir una cantidad de frutas en verdes, poco maduras y muy maduras que, aunque trivial, nos permite entender el concepto que hay detrás de la técnica que nos ocupa. Con ésto podemos comprender otras aplicaciones del clustering, como pueden ser el análisis de caracteres escritos a mano, muestras de diálogo, huellas dactilares o imágenes, la clasificación de especies en subespecies o el agrupamiento para moléculas o proteínas.

Para poder aplicar esta técnica debemos medir d características de nuestro grupo de n objetos, por ejemplo el color o la textura en el caso de las frutas o la simetría o la intensidad en el caso de caracteres escritos a mano, obteniendo una conjunto de datos de longitud n con d dimensiones.

En nuestro caso concreto abordaremos una variación del clustering clásico, el Problema de Agrupamiento con Restricciones(PAR), es decir, además de nuestro conjunto de datos tenemos cierta información a la que llamaremos restricciones o, más concretamente, restricciones de instancia. Ésto quiere decir que tenemos alguna información sobre objetos que tienen que pertenecer al mismo cluster(ML, Must-Link) y sobre objetos que no pertenecen al mismo cluster(CL, Cannot-Link), siendo éstas restricciones débiles, o lo que es lo mismo, podemos incumplir restricciones pero debemos minimizar el número de incumplidas al máximo.

El planteamiento combinatorio de este problema es NP-Completo, por tanto a continuación propondremos algunas alternativas de algoritmos para intentar resolver el problema de forma aproximada en un tiempo razonable. El problema se abordará con k(nº de clusters) conocida, ya que su búsqueda es otro problema complejo.

2. FORMALIZACIÓN Y DESCRIPCIÓN DE LA APLICACIÓN DE LOS ALGORITMOS

2.1 FORMALIZACIÓN DE CONCEPTOS

Vamos a definir distintos conceptos que se nombrarán a lo largo del documento y que son necesarios para entender los procedimientos en cada algoritmo.

Primeramente podemos formalizar la definición de nuestro conjunto de datos como una matriz X de n·d, n objetos en un espacio de d dimensiones(el número de medidas que tenemos sobre cada objeto). Lo podemos notar matemáticamente como:

$$\vec{x}_i = \{x_{[i,1]}, ..., x_{i,d}\} / x_{[i,d]} \in \mathbb{R} \; \forall \; j \in \{1, .., d\}$$

Llamaremos $C=\{c_1,\ldots,c_k\}$ al conjunto de los k clusters, de manera que cada c_i será un subconjunto de X y podrá asociada una etiqueta l_i que lo nombre Para cada cluster es posible calcular su centroide asociado $\vec{\mu}_i$, siendo el vector promedio de sus instancias, de la siguiente manera:

$$\vec{\mu}_i = \frac{1}{|c_i|} \sum_{\vec{x}_j \in c_i} \vec{x}_j$$

También debemos definir la distancia media intra-cluster, $\overline{c_i}$, como la media de las distancias entre cada instancia del cluster y su centroide asociado, usando en este caso la distancia euclídea, aunque es equivalente a usar, por ejemplo, la distancia Manhattan. Es calculable con la siguiente expresión:

$$\overline{c_i} = \frac{1}{|c_i|} \sum_{\vec{x}_j \in c_i} \|\vec{x}_j - \vec{\mu}_i\|_2$$

A través de estas, podemos llegar a la desviación general de C, la media de las desviaciones intra-cluster, que nos será de gran ayuda para minimizar la solución y para el posterior análisis de la solución. La expresión sería la siguiente:

$$\overline{C} = \frac{1}{k} \sum_{c_i \in C} \overline{c_i}$$

Por otro lado tenemos las restricciones, $R = ML \cup CL$, donde $ML(\vec{xi}, \vec{xj})$ indica que las instancias $\vec{x}_i \, y \, \vec{x}_j$ deben estar asignadas al mismo cluster y $CL(\vec{x}_i, \vec{x}_j)$ que las instancias $(\vec{x}_i, \vec{x}_j)$ no pueden ser asignadas al mismo cluster. Con ello, definimos la infactibilidad o infeasibility, que es un indicador de las restricciones que incumple nuestra solución, como:

$$infeasibility = \sum_{i=0}^{|ML|} 1\left(h_c(\vec{ML}_{[i,1]}) \neq h_c(\vec{ML}_{[i,2]})\right) + \sum_{i=0}^{|CL|} 1\left(h_c(\vec{CL}_{[i,1]}) = h_c(\vec{CL}_{[i,2]})\right)$$

Siendo 1 la función booleana que devolverá 1 si la expresión que toma como argumento es verdadera y 0 en otro caso.

2.2 REPRESENTACIÓN DE LA INFORMACIÓN

Ahora que tenemos algunos conceptos claros y formalizados, pasamos a ver como representaremos la información necesaria en el proceso de resolución del problema.

Primero aclarar que los datos de entrada, nuestro dataframe, estará organizado en una matriz implementada como un vector<vector<float> >(ambos vectores de la librería STL), es decir, un vector de longitud n de vectores de reales en coma flotante de longitud d.

Para las restricciones usaremos dos formas de representación, una para facilitar el acceso cuando conocemos la restricción que queremos comprobar y otra para cuando queramos recorrer todas las restricciones. Para el primer caso se usará la misma representación que para el dataframe, mientras que para el segundo se construirá una lista con elementos del tipo [x,y,{1,-1}], siendo x e y los dos elementos que tiene la restricción y el último elemento 1 si conforman una restricción ML y -1 si es CL. Esta lista se ha implementado usando un vector<vector<int> >, también de la STL. De ésta manera, al recorrer esa lista evadiremos todos los pares que no tienen una restricción y nuestra búsqueda será mucho más eficiente que en una matriz.

Por último, representaremos la solución con un vector<int> de longitud n, de manera que la posición i contendrá el número del cluster asignado al objeto i.

2.3 FUNCIÓN OBJETIVO

La función objetivo es la función que debemos minimizar, es decir, la función que dice como de buena es la solución que le pasamos como parámetro.

De todas las definiciones que hemos hecho, desviación general($\overline{C}$) e infactibilidad(infeasibility) nos dan información acerca de la bondad de la solución, la primera a través de la distancia promedio de cada dato a su centroide correspondiente y la segunda a través de la medida del incumplimiento de las restricciones de la solución dada. Ambas han de ser minimizadas, pero hay que darle más o menos importancia a una y a otra, de manera que debemos introducir un parámetro con el que controlar ésta importancia que debatiremos posteriormente. Así, nuestra función objetivo queda definida como:

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

Dicho ésto queda definir el parámetro λ, el cual se ha decidido proponer como el entero superior a la distancia máxima D multiplicada por un parámetro rel(relevancia) y dividida por el número de restricciones totales R.

$$\lambda = \frac{\lceil D \rceil * rel}{|R|}$$

Posteriormente, en el punto 6, se hará un estudio de λ en función del parámetro rel.

A continuación se expone la descripción en pseudocódigo de todos los operadores y de la función objetivo.

calular_infeasibility

```
begin
    Para cada instancia de lista_restricciones
    begin
        Si ( solucion[lista_restricciones[i][0]] = solucion[lista_restricciones[i][1]] ) and
    ( lista_restricciones[2] = -1 )
            entonces infactibilidad ← infactibilidad +1

        Si ( solucion[lista_restricciones[i][0]] != solucion[lista_restricciones[i][1]] ) and
    ( lista_restricciones[2] = 1 )
            entonces infactibilidad ← infactibilidad +1
    end

    return infactibilidad
end
```

calcular_centroides

```
begin
    Para cada instancia i de solucion
        centroide[i] ← centroide[i] + dataframe[i.posicion]

    Para cada centroide
        centroide ← centroide / cluster.size

    return centroides
end
```

calcular_distancia_media_intracluster

```
begin
    Para cada instancia i de solucion
    begin
        distancia_intracluster[i] ← distancia_intracluster[i] +
    distancia_euclidea(centroide[i],dataframe[i.posicion])
    end

    Para cada cluster
        distancia_intracluster[cluster] ← distancia_intracluster[cluster] / cluster.size
    return distancia_intracluster
end
```

calcular_desv_general
```
begin
    Para cada cluster
        desv_general ← desv_general + distancia_intracluster[cluster]

    desv_general  ← desv_general / #clusters

    return desv_general
end
```

calcular_lamda
```
begin
    dist_max ← calcular_máxima_distancia(dataframe)
    lambda ← dist_max * rel / lista_restricciones.size

    return lambda
end
```

calcular_f_objetivo
```
begin
    inf ← calcular_infeasibility()
    dist_intra ← calcular_distancia_intracluster()
    desv ← calcular_desv_general()
    lamb ← calcular_lambda()

    return desv + inf * lamb
end
```

2.4 DATAFRAMES

Los conjuntos de datos usados para testear la bondad de los algoritmos son los siguientes:

- Iris: Contiene información sobre las características de tres tipos de flores de Iris. Tiene 3 clases(k=3) y 6 dimensiones.

- Ecoli: Contiene medidas sobre las ciertas características de diferentes tipos de células que pueden ser empleadas para predecir la localización de ciertas proteínas. Tiene 8 clases(k=8) y 7 dimesiones.

- Rand: Conjunto de datos artificial formado por tres agrupamientos bien diferenciados generados en base a distribuciones normales. Tiene 3 clases(k=3) y 2 dimensiones.

3. DESCRIPCIÓN DE METAHEURÍSTICAS

3.1 ALGORITMO DE BÚSQUEDA LOCAL EL PRIMER MEJOR

Este algoritmo de búsqueda local se basa en comenzar con una solución generada aleatoriamente e ir explorando vecinos, los cuales deben ser válidos, y elegir el primero mejor. Los vecinos se exploran de manera aleatoria. A continuación se describirá el pseudocódigo de la generación de la solución aleatoria y de las dos versiones implementadas de la búsqueda y selección de vecino.

genera_sol_aleatoria
```
begin
    Hacer
    begin
        Para cada i entre 0 y dataframe.size
            solucion[i] ← rand() % k
    end
    Mientras algun cluster esté vacío
end
```

escoger_primer_mejor_vecino_v1
```
begin
    aleatorio ← rand()
    Para i entre 0 y (solucion.size)*(k-1) y mientras f_mejor >= f_actual
    begin
        acceso ← (i+aleatorio)%c.size
        solucion[acceso] ←  (solucion[acceso] + (i/solucion.size) + 1) % k

        f_actual ← calcular_f_objetivo(solucion)
    end
end
```

escoger_primer_mejor_vecino_v2
```
begin
    Para cada posicion i de la solucion y para cada cluster j
        vecinos.add([ i, (solucion[i] + j)%k ])

    vecinos.shuffle

    Para cada vecino y mientras f_mejor >= f_actual
        f_actual ← calcular_f_objetivo(vecino)
end
```

El procedimiento de generar la solución aleatoria es más bien trivial, por lo que no requiere una explicación muy compleja, simplemente creamos el vector solución con tamaño #dataframe con el generador de números aleatorios.

La función escoger_primer_mejor_vecino se repite hasta que no se encuentre una f más pequeña o se completen 100000 evaluaciones de f.

La v1 funciona tal que: en cada iteración se accede a una posición al azar del vector y se itera secuencialmente a partir de ahí, sumando algún número entre 1 y k-1(accediendo de manera secuencial a todos los vecinos) dependiendo del azar. La v2 simplemente genera todos los vecinos, los baraja y busca el primer mejor vecino, además funciona mejor que v1, por lo que es la que se usará para la comparación entre algoritmos.

4. DESCRIPCIÓN DEL ALGORITMO DE COMPARACIÓN COPKM

El algoritmo de comparación COPKM, mediante una política greedy, está basado en el algoritmo k-medias, pero adaptado al uso de restricciones, de manera que lo que minimiza en cada inserción es el aumento de la infactibilidad. Se han implementado dos variantes, se detallan más adelante las diferencias. La descripción de ambas en pseudocódigo sería la siguiente:

copkm_v1

```
begin
    indices ← (0,1,…,dataframe.size -1)
    indices.shuffle
    Para cada cluster
        Generar centroide con coordenadas aleatorio en el rango del dataframe

    Mientras cambie la asignación de cluster
    begin
        Para cada objeto i en el dataframe
        begin
            i2 ← indices[i.posicion]
            Para cada objeto j en solucion
            begin
                Si restricciones[i2][j] = -1
                    entonces infactibilidad[solucion[j]]++
                Si restricciones[i2][j] = 1
                    entonces para cada cluster m != solucion[j] infactibilidad[m]++
            end

            min ← cluster con menor infactibilidad, desempatando con la mínima distancia
            solucion.add(min)
        end

        copia_sol ← solucion
        Para cada i entre 0 y la longitud de la solucion
            solucion[indices[i]] ← copia_sol[i]

        calcular_centroides()
    end
end
```

copkm_v2

```
begin
    indices ← (0,1,…,dataframe.size -1)
    indices.shuffle
    Para cada cluster
        Generar centroide con coordenadas aleatorio en el rango del dataframe

    Mientras cambie la asignación de cluster
    begin
        Para cada objeto i en el dataframe
        begin
            i2 ← indices[i.posicion]
            Si es primera vez, entonces para cada objeto j en solucion
            begin
                Si restricciones[i2][j] = -1
                    entonces infactibilidad[solucion[j]]++
                Si restricciones[i2][j] = 1
                    entonces para cada cluster m != solucion[j] infactibilidad[m]++
            end

            Si no, para cada cluster j
            begin
                solucion[i2] = j
                infactibilidad[j] = calcular_infactibilidad(solucion)
            end0

            min ← cluster con menor infactibilidad, desempatando con la mínima distancia
            Si es primera vez, solucion.add(min)
            Si no, c[i2] = min
        end

        Si es primera vez, entonces
        begin
            copia_sol ← solucion
            Para cada i entre 0 y la longitud de la solucion
                solucion[indices[i]] ← copia_sol[i]
        end

        calcular_centroides()
    end
end
```

La v1 es exactamente lo que dice el pseudocódigo, para cada iteración va introduciendo los objetos(en un orden aleatorio) en el cluster que sume menor infactibilidad, se recalculan los centroides en base a esa solución y se vuelve a generar otra solución con los clusters en su nueva posición.

La v2 es tan solo una versión algo mejorada de la v1, aunque la mejora cambia totalmente el resultado como veremos posteriormente. La mejora es básicamente que en cada iteración no se empieza de nuevo, si no que se tiene en cuenta la solución anterior, de manera que al introducir los

objetos(en un orden aleatorio) se calcula la infactibilidad de cada cluster conforme a la solución anterior con el cambio de cluster del objeto. Esta v2 tiene un parecido razonable con BL, sin embargo, en ningún sitio se nos dice que no se pueda guardar la solución anterior, por lo que, aunque compararé ambas versiones, he mantenido ésta para las comparativas con otros algoritmos.

5. PROCEDIMIENTO CONSIDERADO PARA EL DESARROLLO DE LA PRÁCTICA

5.1 CÓDIGO Y LIBRERÍAS USADAS

La práctica ha sido desarrollada en C++, siendo todo el código implementado a mano, con ayuda de las explicaciones dadas en clase y en el guión, y usando las siguientes librerías:

- iostream: para entrada y salida por teclado y pantalla respectivamente.
- fstream: para entrada y salida por ficheros.
- STL: todos los vectores usados provienen de ésta librería.
- cmath: para algunos cálculos necesarios los algoritmos.
- algorithm: para usar la función shuffle()
- ctime: para medir tiempos de ejecución
- cstdlib: para la generación de números aleatorios

5.2 MANUAL DE USUARIO

Compilación:

Se incluye un makefile, por lo que se compila tan solo ejecutando la orden ‘make’ desde la terminal.

Ejecución completa:

La ejecución completa planteada para el ejercicio(6 conjuntos(3*2 % de restricciones)*5 ejecuciones distintas) se pueden hacer con ayuda del script incluido. Tan solo es necesario darle permisos de ejecución y ejecutarlo con el algoritmo como parámetro, ‘copkm’ para el algoritmo de comparación COPKM o ‘bl’ para el algoritmo de búsqueda local primero el mejor.

Ejecución manual:

Si se desea ejecutar a mano un sólo conjunto, se puede hacer llamando a ‘clustering’ de la siguiente forma:

./clustering <dataframe> <restricciones> <k> <{0:COPKM, 1:BL}> <seed>

También es posible llamarlo sin argumentos, entonces pedirá por teclado la información necesaria.

6. ANÁLISIS DE RENDIMIENTO

6.1 DISCUSIÓN DE VERSIONES EN COPKM

Como se ha hablado más arriba, de las dos versiones implementadas para el COPKM se ha usado para este análisis la segunda(la primera sigue en el código por si fuera de interés) ya que funciona mucho mejor. El cambio, como ya se ha explicado, es tan solo guardar la solución obtenida en la iteración anterior para posteriormente tenerla en cuenta en la siguiente, y esto le proporciona una gran potencia al algoritmo. Conceptualmente lo que se está haciendo es dotar de información a un procedimiento que apenas contaba con ninguna, lo que hace tenga muchas más coherencia en sus ejecuciones. Podemos darnos cuenta de ello a través de varios parámetros de la solución:

La diferencia más importante probablemente es que la v1 cicla en todas las ejecuciones con Ecoli, mientras que v2 incluso es capaz de sacar soluciones cerca del óptimo. Además, en el resto de conjuntos encuentra el óptimo, cuando sacaba alrededor de 400 de infactibilidad con 10% de restricciones y más de 800 con el 20%, y proporcionaba una solución aparentemente incoherente teniendo en cuenta que conocemos la óptima.

La razón, como ya se ha dicho antes, parece ser que el algoritmo cuenta con mucha más información, por lo que es capaz de tomar soluciones mucho mejores con los primeros objetos que son asignados.

6.2 DISCUSIÓN DE VERSIONES EN BL

También se ha hablado de dos versiones para el algoritmo de búsqueda local. Ambas versiones de la función de búsqueda y selección de vecinos hacen el mismo trabajo pero con una sutil diferencia, la v1 al acceder a una posición aleatoria sigue buscando secuencialmente mientra que la v2 busca vecinos en un orden totalmente aleatorio.

En cuanto a rendimiento, aunque la diferencia no es tan grande como en el caso del COPKM, pero v2 es capaz de reducir la infactibilidad de Ecoli más de un 1000% y la desviación general unas 2 unidades con ambos porcentajes de restricciones. La primera versión encontraba soluciones con una infactibilidad de más de 1000 para el 10% de restricciones y más de 2000 para el 20%, además en algunas ocasiones no sacaba el óptimo en Iris y Ecoli. Se pensó como primera solución con la idea de evitar tener que generar y almacenar todo el vecindario virtual, sino hacerlo de forma dinámica mientras se iban comprobando, pero como hemos visto funciona bastante peor que cuando generamos todo el vecindario. Por todo esto es que obviamente se ha utilizado v2 para el análisis comparativo, aunque se ha dejado v1 comentada en el código por si es de interés probarla.

Las razones aquí son un poco menos obvias, pero tiene que ver con lo que se ha dicho en el primer párrafo. La primera versión, aunque consiga no tener que almacenar en memoria todo el vecindario virtual, sólo tiene aleatoria la búsqueda del primer vecino, mientras que v2 hace uso de una búsqueda completamente aleatoria en todas sus iteraciones. Esto provoca que v2 tenga más “agilidad” para evitar óptimos locales en los que sí cae v1, por lo tanto v2 termina siendo más potente aunque en teoría ambas versiones se basen en el mismo concepto.

6.3 DESCRIPCIÓN DE LOS CASOS DEL PROBLEMA

Se van utilizar todos los dataframes anteriormente mencionados para analizar como de buenos son los algoritmos planteados para resolver el problema del agrupamiento con restricciones, con el 10 y el 20% de restricciones en cada conjunto de datos. Los datos organizan en tablas y se analizarán posteriormente.

Para ello se han realizado con cada uno de los tres dataframes, Iris, Ecoli y Rand, tanto con el 10% como con el 20% de restricciones, un total de 5 veces, lo que hace un total de 30 ejecuciones(3 conjuntos x 2 conjuntos de restricciones x 5 veces).

Para poder replicar el total de ejecuciones todas ellas se han realizado utilizando semillas para la generación de números aleatorios, por lo que conociendo estas semillas se puede realizar el estudio exactamente de la misma manera y con los mismos resultados. Las semillas usadas en este estudio son las siguientes:

Ejecución 1	**Ejecución 2**	**Ejecución 3**	**Ejecución 4**	**Ejecución 5**
798245613	123456789	25022020	17042026	459268694

6.4 DISCUSIÓN DEL PARÁMETRO REL

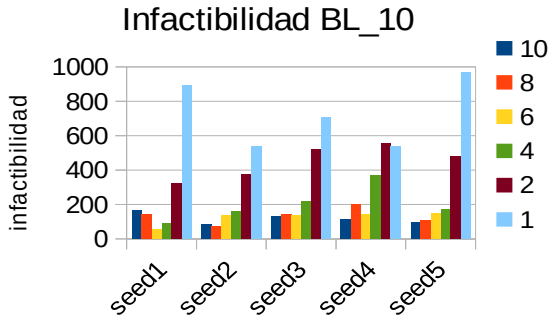
Como se dijo en el punto 2, en este apartado vamos a razonar cual es la función del parámetro rel y por qué se ha usado en esta implementación.

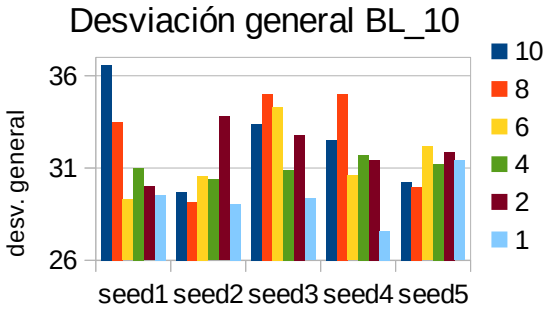
El parámetro rel, recordemos, forma parte del cálculo del parámetro lambda en la función objetivo:

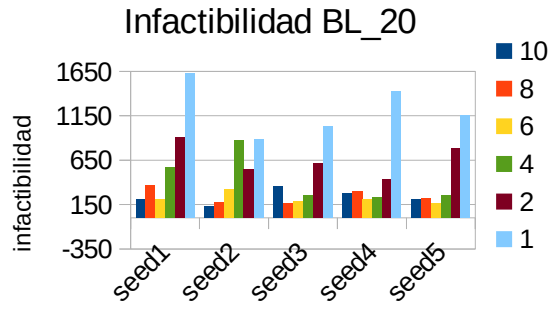
$$f = \overline{C} + (infeasibility) * \lambda \qquad \lambda = \frac{\lceil D \rceil * rel}{|R|}$$

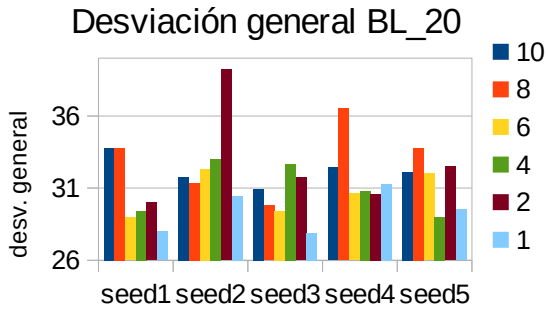
La función del parámetro rel, como podemos ver, es la de calibrar la importancia que tienen la desviación general y la infactibilidad en la función objetivo, de tal manera que si utilizamos un rel con valor muy grande la infactibilidad será mucho más importante que la desviación general a la hora de minimizar f, por lo que el algoritmo tenderá a minimizar más la primera antes que la segunda, y viceversa en caso de usar un rel con un valor muy pequeño. Es obvio que ésto es una manera artificial de variar lambda, pero se ha planteado así ya que el criterio para calcular lambda parece correcto y sólo buscamos corregirlo ligeramente para intentar optimizar la minimización(valga la redundancia).

En este caso, en COPKM no tiene sentido usarlo ya que solo hace uso de la infactibilidad para buscar la solución, y BL encuentra el óptimo para Iris y Rand con cualquier valor de rel, por lo que lo que buscamos es la mejor solución posible para Ecoli. A continuación se hace un estudio para establecer el valor de rel para este caso en concreto.









A través de éste estudio podemos deducir que los mejores valores para rel podrían ser 4 o 6, habiéndose seleccionado 6. También podemos ver que si no se hubiera hecho este estudio y no se hubiera usado el parámetro rel, es decir, se hubiera usado un valor 1, las soluciones obtenidas habrían sido mucho peores, ya que, aunque la desviación general sea en algunos casos un poco más pequeña que con otro valor, la infactibilidad se dispara para valores pequeños.

Terminado el estudio se establece el valor rel a 6 y se procede a la ejecución explicada en el punto anterior.

6.5 RESULTADOS OBTENIDOS

Los resultados se muestran a continuación en el formato indicado:

COPKM 10%	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,669	0	0,669	0,020	36,919	2	36,959	0,268	0,757	0	0,757	0,004
Ejecución 2	0,669	0	0,669	0,004	39,523	64	40,740	0,234	0,757	0	0,757	0,004
Ejecución 3	0,669	0	0,669	0,009	39,584	89	41,450	0,265	0,757	0	0,757	0,004
Ejecución 4	0,669	0	0,669	0,004	33,933	1	33,954	0,203	0,757	0	0,757	0,003
Ejecución 5	0,669	0	0,669	0,003	39,410	3	39,466	0,273	0,757	0	0,757	0,004
Media	0,67	0,00	0,67	0,008	37,87	31,80	38,51	0,25	0,76	0,00	0,76	0,00

COPKM 20%	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,669	0	0,669	0,007	35,730	8	35,821	0,327	0,757	0	0,757	0,008
Ejecución 2	0,669	0	0,669	0,008	36,721	9	36,803	0,265	0,757	0	0,757	0,006
Ejecución 3	0,669	0	0,669	0,006	35,360	0	35,360	0,328	0,757	0	0,757	0,006
Ejecución 4	0,669	0	0,669	0,006	35,360	0	35,360	0,329	0,757	0	0,757	0,007
Ejecución 5	0,669	0	0,669	0,007	37,584	12	37,706	0,264	0,757	0	0,757	0,009
Media	0,67	0,00	0,67	0,007	36,15	5,80	36,21	0,30	0,76	0,00	0,76	0,01

BL 10%	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,669	0	0,669	0,151	29,336	57	36,142	5,950	0,757	0	0,757	0,116
Ejecución 2	0,669	0	0,669	0,120	30,591	135	47,286	1,241	0,757	0	0,757	0,254
Ejecución 3	0,669	0	0,669	0,218	34,316	136	51,280	4,410	0,757	0	0,757	0,127
Ejecución 4	0,669	0	0,669	0,222	30,610	143	47,380	4,137	0,757	0	0,757	0,196
Ejecución 5	0,669	0	0,669	0,148	32,187	149	50,770	4,354	0,757	0	0,757	0,103
Media	0,67	0,00	0,67	0,17	31,41	124,00	46,57	4,02	0,76	0,00	0,76	0,16

BL 20%	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,669	0	0,669	0,162	28,967	204	41,146	1,916	0,757	0	0,757	0,152
Ejecución 2	0,669	0	0,669	0,171	32,345	316	51,885	1,962	0,757	0	0,757	0,430
Ejecución 3	0,669	0	0,669	0,422	29,404	190	41,253	7,133	0,757	0	0,757	0,206
Ejecución 4	0,669	0	0,669	0,351	30,643	210	42,957	7,329	0,757	0	0,757	0,244
Ejecución 5	0,669	0	0,669	0,192	32,005	169	45,545	8,000	0,757	0	0,757	0,130
Media	0,67	0,00	0,67	0,26	30,67	217,80	44,56	5,27	0,76	0,00	0,76	0,23

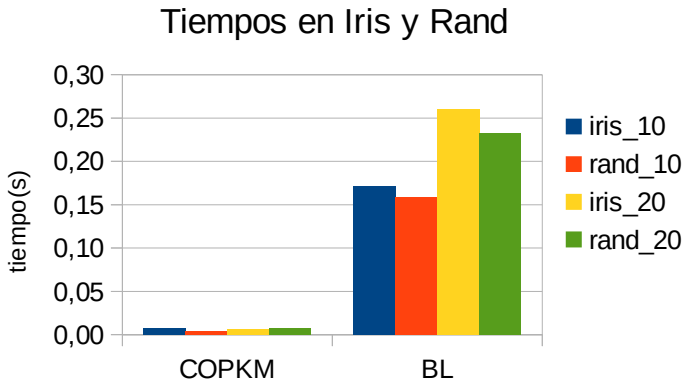
Y de ésto obtenemos los siguientes valores medios:

10%	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	0,01	37,87	31,80	38,51	0,25	0,76	0,00	0,76	0,00
BL	0,67	0,00	0,67	0,17	31,41	124,00	46,57	4,02	0,76	0,00	0,76	0,16

20%	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	0,01	36,15	5,80	36,21	0,30	0,76	0,00	0,76	0,01
BL	0,67	0,00	0,67	0,26	30,67	217,80	44,56	5,27	0,76	0,00	0,76	0,23

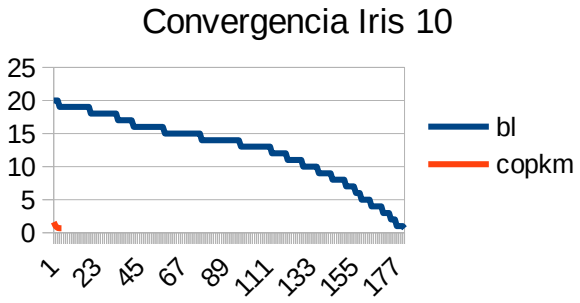
6.6 ANÁLISIS DE RESULTADOS

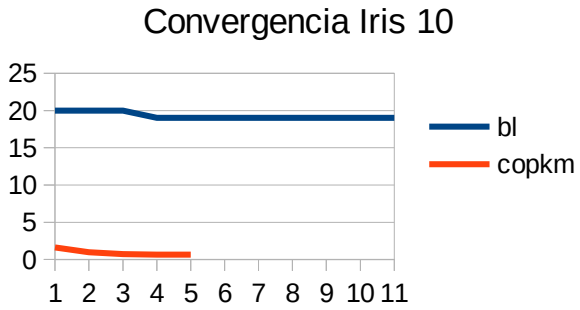
Ahora vamos a analizar los resultados obtenidos, empezando por ver que ambos algoritmos encuentran el óptimo tanto en Iris como en Rand, por lo que solo nos queda comparar el tiempo en el que lo hacen.



Como podemos ver COPKM encuentra la solución óptima, de media, en un tiempo mucho menor que BL, lo cual es lógico ya que el principal objetivo de los algoritmos greedy es buscar una solución lo mejor posible en un tiempo reducido. En nuestro caso la potencia del COPKM es suficiente para llegar a la solución óptima, debido a que es un algoritmo greedy y hace uso de técnicas mucho más simples que BL, lo que lo convierte en una muy buena opción para conjuntos poco complejos de agrupar.

Podemos observar éste comportamiento observando como converge f, es decir, que valores va tomando f en cada iteración hasta que encuentra el óptimo.





En cuanto a Ecoli, podemos ver que COPKM encuentra la solución mucho más rápido y con una f bastante más alta, sin embargo fijémonos en la f desagregada. Podemos ver como BL, aunque devuelve una solución con una infactibilidad más alta, la desviación general es bastante más baja, por lo tanto obviando f(recordemos también que hemos multiplicado lamba por 6 a través del parámetro rel), podemos ver que devuelve una solución algo más coherente.

Esto se debe claramente a que COPKM no tiene en cuenta la desviación general, si no que basa su uso completamente en la infactibilidad, y sin embargo BL sí que tiene en cuenta ambos valores a la hora de buscar una solución.

Sin embargo, cuando podamos comparar estas soluciones con las que proporcionen otros algoritmo a lo largo del resto de las prácticas podremos hacer un estudio más riguroso de como de buenas son estas soluciones.

7. BIBLIOGRAFÍA

- http://www.cplusplus.com/reference/vector/vector/
- http://www.cplusplus.com/reference/cstdlib/
- http://www.cplusplus.com/reference/fstream/
- http://www.cplusplus.com/reference/ctime/
- http://www.cplusplus.com/reference/algorithm/count/