Practica 1.b
Técnicas de Búsqueda Local y Algoritmos Greedy para el Problema del Agrupamiento con Restricciones

Curso 2019-2020

Tercer Curso del Grado en Ingeniería Informática

Metaheurística

Rafael Vázquez Conejo
DNI: 49137372B
Email: rafavazquez99@correo.ugr.es
Grupo Prácticas 1, miercoles.

1. Índice

2. Descripción del Problema del PAR

El problema del Agrupamiento con Restricciones (PAR) consiste en una generalización del agrupamiento clásico, permitiendo la incorporación de un nuevo tipo de información (restricciones) al proceso de agrupamiento. Nuestro problema trata de optimizar la clasificación de un conjunto de datos recibidos “X” con “n” instancias cada uno en particiones C del mismo, de manera que se minimice la desviación general y se cumplan las restricciones establecidas en el conjunto de restricciones R.

Es decir, para decidir que cluster $c \in C = c_1, \ldots, c_{1k}$ le asignamos a cada instancia de nuestro conjunto de datos X tenderemos en cuenta que la distancia de nuestra instancia al cluster sea lo más mínima posible, del mismo modo el número de restricciones que incumplimos al asignar dicha instancia al cluster sea lo menor posible.

Por lo tanto respecto a las restricciones de instancia podemos establecer que dada una pareja de instancias se establece una restricción del tipo Must-Link (ML), si estas instancias debe pertenecer al mismo cluster, o del tipo Cannot-Link (CL), si estas no pueden pertenecer al mismo cluster. Respecto a las restricciones de distancia, las instancias separadas por una distancia mayor a una dada deben pertenecer a diferentes clusters, a su vez las instancias separadas por una distancia menor que una dada deben pertenecer al mismo cluster.

Existen dos variantes del problema respecto al modo de interpretar las restricciones. Si todas las restricciones deben satisfacerse en la partición C de nuestro conjunto de datos X, nos encontramos ante Restricciones fuertes (Hard), o si buscamos que la partición C de nuestro conjunto de datos X debe minimizar el número de restricciones incumplidas pero puede incumplir algunas de ellas, este es el caso de Restricciones débiles (Soft). En nuestro caso nos encontramos ante la segunda variante del problema, restricciones débiles.

El valor que debemos buscar minimizar para encontrar el mejor cluster posible para cada instancia de X, es decir, nuestra función de valoración será:

$$f = \vec{C} + (\text{infeasability} * \lambda)$$

$$\overrightarrow{\mu_i} = \frac{1}{|c_i|} \sum_{\overrightarrow{x_j} \in c_i} \overrightarrow{x_j} \qquad \overline{c_i} = \frac{1}{|c_i|} \sum_{\overrightarrow{x_j} \in c_i} \|\overrightarrow{x_j} - \overrightarrow{\mu_i}\|_2 \qquad \overline{C} = \frac{1}{k} \sum_{c_i \in C} \overline{c_i}$$

$$infeasibility = \sum_{i=0}^{|ML|} \mathbb{1}(h_C(\overrightarrow{ML_{[i,1]}}) \neq h_C(\overrightarrow{ML_{[i,2]}})) + \sum_{i=0}^{|CL|} \mathbb{1}(h_C(\overrightarrow{CL_{[i,1]}}) = h_C(\overrightarrow{CL_{[i,2]}}))$$

Los parámetro de estas fórmula son los siguientes:

- $\vec{C}$, la desviación general de la partición C, se calcula como la media de las desviaciones intra-cluster, siendo a su vez la distancia media intra-cluster c_i la media de las distancias de las instancias que conforman el cluster con su centroide .
- infeasability, es el número de restricciones que se incumplen en C.
- λ, cociente entre la máxima distancia existente en el conjunto de datos y el número de restricciones presentes en el problema, |R|.

3. Descripción de los elementos y algoritmos comunes de los algoritmos.

3.1 Representación de los datos

Comentar que el lenguaje de programación que he utilizado es C++.

Tanto los datos de entrada X como las restricciones las he almacenado en un vector de vectores del tipo double. Todas las estructuras que he utilizado en la resolución del problema han sido vectores de la clase “vector”, ya que en un primer análisis del problema me parecieron una forma útil de representación de los datos y no de un alto coste computacional, en muchos casos podría haber sustituido un vector por el uso de una lista, pero debido a tener más conocimientos y practicas con los vectores he preferido su utilización.
Los datos que he empleado en ambos algoritmos son:

Un vector de vectores del tipo double en el cual he almacenado el centroide correspondiente para cada cluster, llamado “*centroides*”, de esta forma cada vector en la posición “i” contiene el valor del centroide del cluster “i”. He utilizado una variable del tipo int en la cual he almacenado el número de cluster que tengo en el problema, y que establece el propio usuario, “n_*cluster*”, a su vez he creado un vector del tipo int en el cual cada posición “i” almacena el número de instancias existentes en el cluster “i”, “*total_por_cluster*”. Finalmente tengo una variable del tipo double llamada media que almacena la desviación general $\vec{C}$, “*media”*.

Mi solución, si nos referimos al cluster asignado para cada instancia, la represento en un vector de int en el cual cada valor “i” corresponde al cluster asignado a la instancia “i” del conjunto de datos, llamado “clusters”. De forma que si la posición “i” vale 2, significa que esta instancia está asignada al cluster 2.

3.2 Elementos en cada Cluster

Para poder obtener la desviación general y para comprobar que no estoy dejando ningún cluster sin ninguna instancia utilizo este método, el cual me permite saber el número de instancias asignadas a cada cluster. Se hará uso de un vector del tipo int que contendrá para cada instancia en la posición “i” el cluster que posee asociado. En el caso del algoritmo greedy, este vector se pasará por parámetro. En pseudocógido, el algoritmo es el siguiente:

```
Función elementosCadaCluster ():
        cluster_actual = 0
        elementos (vector de int, donde almaceno el número de elementos de cada cluster)
        HACER
                PARA cada cluster de mi total de clusters
                        SI cluster = cluster_actual
                                Incremento el valor de elementos en el cluster_actual en una unidad
                        FIN-SI
                FIN-PARA
                Incremento el cluster_actual en una unidad
        MIENTRAS cluster_actual != total cluster
        Devuelvo un vector int con el numero de elementos en cada cluster
```

3.3 Obtener centroides

Esta función nos devolverá un vector de vectores del tipo double que nos permitirá saber el valor del centroide de cada cluster “i”, algo que nos será esencial tanto en ambos algoritmos para el calculo de la mayor distancia para obtener λ y para la obtención de la desviación general. Igual que en el caso anterior se hará uso de un vector que contendrá para cada instancia en la posición “i” el cluster que posee asociado. En el caso del algoritmo greedy, este vector se pasará por parámetro. En pseudocógido, el algoritmo es el siguiente:

```
Función obtenerCentroide():
        cluster_actual = 0
        total_por_cluster = elementosCadaCluster()
        sumas (vector de double, suma los valores de las instancias, inicializado a 0)
        Hacer
                Para cada instancia en el conjunto de datos i
                        Si el cluster de la instancia = cluster_actual
                                Para cada componente de la instancia
                                        sumas[componente] = sumas[componente]  + datos[instancia]
                                                                                [componente]
                                Fin_Para
                        Fin_Si
                Fin_Para
                Para cada componente de sumas
                        divido sumas[componente] entre el total_por_cluster[cluster_actual]
                Fin_Para
                Le añado a mi vector de vectores double, centroides el valor de sumas
                Incremento el cluster_actual en una unidad
                Para cada componente de sumas
                        sumas en cada componente toma el valor de 0
                Fin_Para
        Mientras cluster_actual != total cluster
        Devuelvo el vector de vectores de tipo double, centroides
Fin
```

3.4. Calcular Violaciones

A continuación establezco una serie de funciones que me ayudarán al calculo de la función objetivo, la primera de esta me permite obtener el número total de restricciones que se incumplen en mi solución obtenida. Esta función hace uso del vector clusters que contiene el cluster asignado a cada instancia ya mencionada anteriormente. De nuevo en el caso de greedy recibe como parámetro la distribución de los cluster para cada instancia. En pseudocógido, el algoritmo es el siguiente:

```
Función calcularViolaciones():
        no_cumplo = 0
        Para cada valor de restricciones en mi conjunto de restricciones
                Para cada componente de restricciones[valor] en mi conjunto de restricciones
                        Si restricciones[valor][componente] == -1
                                Si clusters[valor] == clusters[componente]
                                        Incremento el valor de no_cumplo en una unidad
                                Fin_Si
                        Fin_Si
```

```
                    Si restricciones[valor][componente] == 1
                        Si cluster[valor] != cluster[componente]
                            Incremento el valor de no_cumplo en una unidad
                        Fin_Si
                Fin_Para
        Fin_Para
        Devuelvo no_cumplo con el número de restricciones incumplidas
Fin
```

Podemos observar que esta función comprueba que si dos elementos están en el mismo cluster y no deberían según las restricciones aumenta el número de violaciones, lo aumenta también si dos elementos no están en el mismo cluster y según las restricciones deberían.

3.5 Mayor Distancia

Esta función también es necesaria para la obtención de la función evaluación, esta nos permite obtener tanto la desviación general cómo la mayor distancia de una instancia a un cluster, necesaria para la obtención de λ. Esta hace uso de "*clusters*" para saber el cluster asignado a cada instacia y de "centroides" gracias a la función obtenerCentroides() obtengo los centroides de cada cluster.
En pseudocógido, el algoritmo es el siguiente:

```
Función mayorDistancia():
        cluster_actual = 0
        suma_distancia = 0
        max = -9999
        distancia_por_cluster (vector para almacenar la distancia media de cada cluster)
        Hacer
                Para cada instancia en el conjunto de datos i
                    Si el cluster de la instancia = cluster_actual
                        Para cada componente de la instancia
                            realizo la suma de la diferencia de cada instancia al centroide
                        Fin_Para
```

distancia = $\sqrt{(centroides[cluster\ actual][componente] - datos[instancia][componente])^2}$

```
                        incremento suma_distancia con el valor de distancia
                        Si distancia > max
                            max = distancia
                        Fin_Si
                    Fin_Si
                Fin_Para
                Divido suma_distancia entre el total de elementos del cluster actual
                Añado suma distancia al vector distancia_por_cluster
                suma_distancia = 0
                Incremento el valor de cluster_actual en una unidad
        Mientras cluster_actual != total cluster
        media = 0
        Para cada distancia en distancia_por_cluster
                media más la distancia_por_cluster[distancia]
        Fin_Para
        Divido la media entre el número total de clusters
        Devuelvo la distancia maxima, max
Fin
```

El valor de media lo almaceno en una variable de la clase

3.6 Total de Violaciones

La última función para poder implementar la función de evaluación, esta recorre las restricciones y devuelve el número total de restricciones establecidas. Es una función sencilla que simplemente suma una unidad si encontramos un 1 o un -1 en las restricciones. En pseudocógido, el algoritmo es el siguiente:

```
Funcion totalViolaciones():
        violaciones = 0
        Para cada valor del conjunto de restricciones
                Para cada componente de cada restricciones[valor] del conjunto de restricciones
                        Si (restricciones[valor][componente] == 1) OR
                                                        (restricciones[valor][componente] == -1)
                                Incremento violaciones en una unidad
                        Fin_Si
                Fin_Para
        Fin_Para
        Devuelvo violaciones con el número total de violaciones en el conjunto restricciones
Fin
```

3.7 Función Valoración

La evaluación de la solución se realiza según la formula establecida anteriormente :

$$f = \vec{C} + (infeasability * \lambda)$$

Recordando que $\vec{C}$ será la desviación general que contiene nuestra variable “media”, infeasability es el número de restricciones violadas y nuestra λ será la mayor distancia entre el número total de restricciones en el conjunto. Una vez hemos establecido las funciones anteriores simplemente debemos llamarlas de forma correcta. En pseudocógido, el algoritmo es el siguiente:

```
Función funcionValoracion():
        mayor_distancia = mayorDistancia()
        violaciones_realizadas = calcularViolaciones()
        n_violaciones = totalViolaciones()
        landa = cociente mayor_distancia entre n_violaciones
        valoración = getMedia() + violaciones_realizadas * landa
        Devuelvo valoración con la valoración de nuestra solución
Fin
```

getMedia() es una función que simplemente nos devuelve “media” que recordemos es una variable de nuestra clase y que contiene la desviación general.

4. Descripción del método de búsqueda

Nuestro algoritmo de búsqueda utiliza una estrategia del tipo “el primer mejor” para realizar la exploración del entorno de una solución. Esto consiste en aceptar el primer vecino que mejore nuestra solución actual y establecerla como nuestra nueva solución.

De esta forma en nuestro algoritmo iremos explorando todos los vecinos y para cada uno de ellos le asignaremos cada uno de los posibles clusters. Si al establecerle un nuevo cluster la solución mejora nos quedaremos con esa como nueva solución.

Como criterio de parada se tendrá que haber realizado 100000 evaluaciones de búsqueda de vecinos (número establecido en el guión de prácticas) o cuando se han explorado todos los vecinos, es decir cuando el algoritmo de evaluación nos devuelve el mismo valor que en la evaluación anterior.

El pseudocódigo de la búsqueda local es el siguiente:

```
Funcion funcionBusquedaLocal():
        asignacionAleatoria()                   //Asigno un cluster aleatorio a cada instancia
        obtenerCentroide()                      //Obtengo los centroides de la solución aleatoria inicial
        primera_valoracion = funcionValoracion()        //Evalúo la solución aleatoria
        PARA contador HASTA contador < 100000
                //realizo la busqueda de vecino para obtener una nueva solución
                nueva_valoracion = busquedaVecino(primera_valoracion)
                SI nueva_valoracion == primera_valoracion
                        FIN             //Hemos explorado todos los posibles vecinos, finalizamos
                FIN-SI
                SINO
                        primera_valoracion = nueva_valoracion
                FIN-SINO
        FIN-PARA
FIN
```

Como podemos observar he empleado cuatro funciones exteriores para que me fuera más cómoda y legible la implementación de la función, además de la función “funcionValoracion” y la función “obtenerCentroide”, ya explicadas anteriormente, encontramos otras dos.

Comenzaremos definiendo un método que nos permite asignarle a cada instancia un cluster aleatorio, dándole valores a nuestro vector de tipo int “*clusters*”.

```
Función asignaciónAleatoria():
        alguno_vacio = true
        MIENTRAS alguno_vacio == true HACER
                PARA cada instancia del conjunto de datos
                        Genero un número aleatorio entre [0 , total de clusters]
                        Añado el número al vector clusters
                FIN-PARA
        //La función clusterVacio simplemente recorre el vector de “clusters” comprobando que
        //todo cluster tiene al menos un elemento
        alguno_vacio = clusterVacio()
```

```
        FIN-MIENTRAS
FIN
```

Finalmente encontramos la función que nos permite generar vecinos, en la cual para cada instancia observo el cluster asignado, modifico este valor y compruebo si la solución nueva obtenida es mejor que la anterior, si es así me quedo con la nueva mejor y sigo buscango una mejor con la nueva obtenida. Si miro todas las instancias, es decir exploro todos los vecinos posibles la "funcionBusquedaLocal" se encargará de finalizar la generación de vecinos.
Esta función devolverá la valoración de la nueva mejor solución obtenida y recibirá como parámetro el valor de la mejor solución actual.

```
Función busquedaVecino(double primera_valoracion):
        //almaceno en un vector la situación anterior a la generación de vecinos
        antes_modificar = clusters
        i = 0
        j = 1
        MIENTRAS no estemos en el ultimo valor del vector clusters HACER
                //cambio el cluster de la instancia haciendo módulo para no superar el número total
                //de clusters
                clusters[i] = (antes_modificar[i] + j) módulo n_cluster
                //He probado todos los clusters para esta instancia, paso a la siguiente instacia
                SI cluster[i] == antes_modificar[i]
                        Incremento i en una unidad
                        Incremento la posición en el vector de clusters
                FIN-SI
                SINO-SI clusterVacio() == false
                        Incremento j en una unidad
                        //evalúo la nueva solución
                        nueva_valoracion = funcionValoracion()
                        //si es mejor que la actual la almaceno como la nueva
                        SI nueva_valoracion < primera_valoracion
                                Devuelvo nueva_valoracion
                        FIN-SI
                FIN-SINO-SI
                SINO
                        Incremento j en una unidad
                FIN-SINO
        FIN-MIENTRAS
        //si llego a esta situación significa que he explorado todos los vecinos
        Devuelvo primera_valoracion
FIN
```

5. Descripción de los algoritmos de comparación

Nuestro algoritmo greedy comienza generando unos centroides aleatorios para todos nuestro clusters, tras ello asignaré cada instancia al cluster elegido. El criterio de elección será minimizar el valor de infeasibility, es decir, la instacia se asociará al cluster dónde el número de restricciones incumplidas sea menor. Esto nos plantea un caso de empate, en el que dos cluster ocasionen el mismo valor de infeasibility para una instancia. Ante este caso estableceremos una función de desempate, la cual priorizará el cluster cuyo centroide se encuentre más cerca de la instancia.
Una vez hemos asignado todas las instancia siguiendo este criterio obtendremos los verdaderos centroides de cada cluster en función del reparto de cluster obtenido. Tras obtener los verdaderos centroides repetiremos el proceso anterior para asignar cada instancia al cluster más adecuado.

Nuestro criterio de parada se activará cuando el algoritmo de evaluación nos devuelva la misma evaluación que en el caso anterior, lo que nos indica que no le es posible mejorar la solución obtenida.

```
Funcion greedy():
    //busco el máximo y mínimo de cada variable de las instancias
    maximoMinimo()
    //aleatoriamente asigno centroides para cada cluster
    centroides = centroidesAleatorios()
    //mediante swap desordeno los índices de 0 a n, siendo n el total de instancias
    crearIndicesAleatorios(datos.size());
    violaciones = 0
    min = 99999
    seguimos = false

    HACER
        //Guardo la forma en la que estaban repartidas las instancias en los clusters
        //Me servirá para establecer el criterio de parada
        asignaciones_clusters_anterior = asignacion_clusters
        PARA cada instancia desordenadas por mi vector de indices aleatorios
            PARA cada cluster del total de clusters
                //obtengo el número de violaciones cometidas por la instancia si le
                //asigno el cluster actual
                violaciones = violacionesCluster(instancia, cluster,
                                                 asignaciones_clusters)
                SI violaciones < min
                    min = violaciones          //nuevo mínimo establecido
                    limpio el vector de empate //un nuevo min para comparar
                FIN-SI
                //caso de que el número de violaciones de nuestra instancia sea igual
                //en un cluster que en otro, tendremos que aplicar la función de
                //desempate
```

```
                    SI violaciones == min
                        Añadimos a empate_violaciones el cluster empatado
                        //caso de que no haya otro valor min igual, nos quedamos con
                        //el cluster actual
                        cluster_elegido = cluster actual
                    FIN-SI
                FIN-PARA
                //caso de más de una opción de cluster para una instacia, llamo a la función de
                //desempate
                SI el tamaño de empate_violaciones != 1
                    cluster_elegido = obtenerCentroideCercano(empate_violaciones,
                                                              instancia, centroides)
                    //obtendré el cluster cuyo centroide está a menor distancia de la
                    //instancia actual
                FIN-SI
                Añado a asignaciones_clusters[instancia] el cluster_elegido
                Limpio el vector de empate_violaciones para la próxima iteración
                Asigno un valor muy elevado a min para la próxima iteración

                //Ahora debemos comprobar que ningún cluster ha quedado vacio
                Obtengo todos los elementos que hay en cada cluster
                Si alguno de los cluster no posee ningún elemento tendré que asignarle uno
                Para ello obtengo la instancia más cercano al centroide de dicho cluster y
                asigno esta instancia a mi cluster vacio
                Repetiré este proceso hasta que ningún cluster esté vacio

                //obtengo los verdaderos centroides
                centroides = obtenerCentroide(asignacion_clusters)
                //finalmente evalúo el resultado obtenido
                resultado = funcionValoracion(centroides, asignacion_clusters)
                //Esta es mi nueva valoración
                //compruebo si la situación de los clusters es igual a la anterior
                SI asignaciones_clusters != asignaciones_clusters_anterior
                    seguimos = true
                FIN-SI
                SINO
                    seguimos = false
                FIN-SINO
            MIENTRAS(seguimos == true)
FIN
```

6. Breve manual de usuario

Como ya he mencionada anteriormente la práctica está realizada en el lenguaje de programación C++. Para su realización he utilizado el guión de la práctica así como las diapositivas del seminario.

El código se encuentra dividido en 5 archivos .cpp, situados en la carpeta “src” con su correspondiente “.h” en la carpeta “include”. Los datos descargados de la web correspondientes a los conjuntos de datos y restricciones se sitúan en la carpeta “datos”.

A continuación detallaré la información contenida en cada archivo:

random.cpp/.h: este fichero contiene el mismo código que el publicado en la web. Para utilizar esta clase, utilizo desde el “main” la función “Set_random” inicializando la semilla a 49137372.

utiles.cpp/.h: en estos archivos he definido los métodos para leer los datos de entrada, tanto del conjunto de datos como el de la matriz de restricciones. Métodos que utilizaré en el “main” para inicializar los vectores de datos y restricciones. La lectura la he realizado gracias a “fstream”

busquedaLocal.cpp/.h: como es de esperar en este archivo están definidos los métodos necesarios para ejecutar el algoritmo de búsqueda local.

COPKM.cpp/.h: en este caso encontramos los métodos necesarios para ejecutar desde el “main” el algoritmo “greedy”.

main.cpp: es el programa principal desde el que se llama a los métodos anteriormente descritos. Su estructura de forma resumida es:

```
    Set_random(49137372)
    datos = util.leerArchivoMatriz(“conjunto de datos”)
    restricciones = util.leerArchivoMatriz(“conjunto de restricciones”)
//Encontramos variables definidas en el main que nos permite acceder a cada conjunto de datos,
//de esta forma si queremos utilizar el cnjunto de datos de iris → util.leerArchivoMatriz(“iris”)
    //Inicializo la busqueda local
    busquedaLocal busq_local(n_cluster, datos, restricciones);
    //inicializo greedy
    COPKM greedy(n_cluster, datos, restricciones);

//Tras esto llamaré a las funciones para ejecutar ambos algoritmos
```

He realizado un makefile que se encarga de la compilación de todos los archivos.
Para lanzar el programa se debe lanzar desde una terminal situada en la carpeta raíz los comandos:

- make
- ./PAR

7. Experimentos y análisis de resultados

Antes de analizar los resultados obtenidos voy a explicar los conjunto de datos y restricciones que se han utilizado. Respecto a los conjuntos de datos:

- Iris: Posee información sobre las características de tres tipos de flores Iris, por lo tanto tendremos 3 clases, es decir, el número de clusters es 3.
- Ecoli: contiene características sobre diferentes tipos de células utilizadas para predecir la localización de ciertas proteínas. Tiene 8 clases, es decir, número de clusters es 8.
- Rand: Es un conjunto de datos artificial, se encuentra formado por 3 clusters bien diferenciados en base a distribuciones normales.

Respecto al conjunto de restricciones, para cada conjunto de datos tenemos 2 conjuntos de restricciones generadas aleatoriamente, correspondientes al 10% y 20% del total de restricciones posibles.

Mencionar que los tiempos reflejados en la tabla están en segundos y los he obtenido gracias a la librería “chrono” y “ctime”. Las semillas utilizadas en cada ejecucción comenzando con la Ejecución 1 hacia la 5, han sido: “49137372”, “491373”, “4913”, “49”, “100”.

Analizados los conjuntos de datos y restricciones, pasamos a mostrar las tablas para cada algoritmo:

Tabla 6.1: Resultados obtenidos por el algoritmo BL en el PAR con 10% de restricciones

	Iris				**Ecoli**				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	1,78	0,00	1,78	0,94	33,80	83,00	34,89	188,86	1,29	0,00	1,29	1,05
Ejecución 2	1,69	0,00	1,69	1,00	35,94	13,00	36,11	219,58	1,43	0,00	1,43	0,97
Ejecución 3	1,70	0,00	1,70	1,27	33,20	132,00	34,94	202,71	1,42	0,00	1,42	1,05
Ejecución 4	1,55	0,00	1,55	1,06	35,46	11,00	35,60	197,33	1,65	0,00	1,65	1,41
Ejecuión 5	1,90	0,00	1,90	0,83	39,16	7,00	39,25	228,34	1,20	0,00	1,20	0,91
Media	1,72	0,00	1,72	1,02	35,51	49,20	36,16	207,36	1,40	0,00	1,40	1,08

Tabla 6.2: Resultados obtenidos por el algoritmo COPKM en el PAR con 10% de restricciones

	Iris				**Ecoli**				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	0,73	129,00	1,04	0,00	16,34	166,00	18,67	0,01	0,55	0,00	0,55	0,00
Ejecución 2	0,66	91,00	0,84	0,00	21,04	619,00	29,66	0,01	0,55	0,00	0,55	0,00
Ejecución 3	0,64	0,00	0,64	0,00	18,43	264,00	22,07	0,01	0,55	0,00	0,55	0,00
Ejecución 4	0,64	0,00	0,64	0,00	20,88	454,00	27,14	0,01	0,55	0,00	0,55	0,00
Ejecuión 5	0,67	23,00	0,74	0,00	13,24	1176,00	30,24	0,01	0,55	0,00	0,55	0,00
Media	0,66	48,60	0,78	0,00	17,99	535,80	25,55	0,01	0,55	0,00	0,55	0,00

Estas dos primeras tablas corresponden a la ejecución del algoritmo Búsqueda Local (Tabla 6.1) y el algoritmo greedy, COPKM (Tabla 6.2) ambos con el conjunto de resolución del 10%. Al compararlas lo más destacable son sus tiempos, en el caso del algoritmo greedy estos son bastante más inferiores. Esta gran diferencias de tiempos de debe principalmente a la búsqueda de vecinos en el algoritmo de Búsqueda Local, ya que en ésta, por cada cambio en el cluster asociado a una

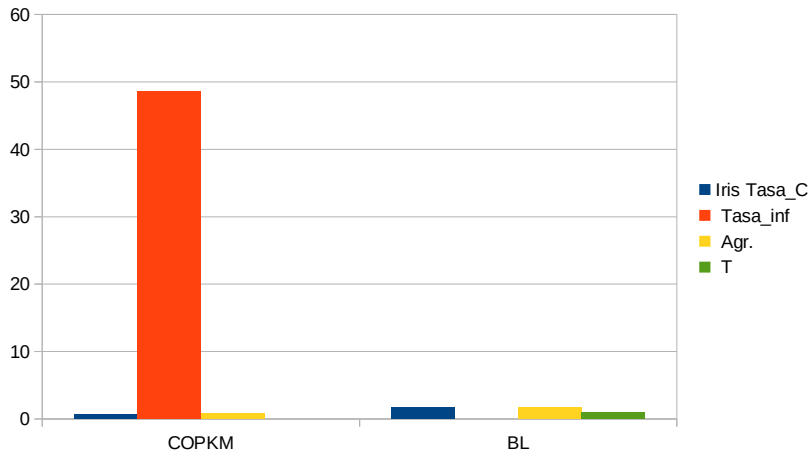
instancia se llama a la función evaluación, mientras que en el algoritmo greedy sólo se llama una vez hemos situado cada instancia en un cluster en función de minimizar las restricciones incumplidas, y si debemos desempatar, en función a la distancia de la instancia al cluster correspondiente.

Respecto a la calidad de las soluciones podemos observar que en el algoritmo de Búsqueda Local la Tasa_inf siempre es menor, en el caso del conjunto de datos ecoli e iris existe una gran diferencia entre estos valores. Esto nos muestra que el primer algoritmo le da una mayor prioridad a disminuir el número de restricciones incumplidas, por lo que podemos decir que el algoritmo de búsqueda local obtiene mejores soluciones respecto a minimizar el número de restricciones violadas.

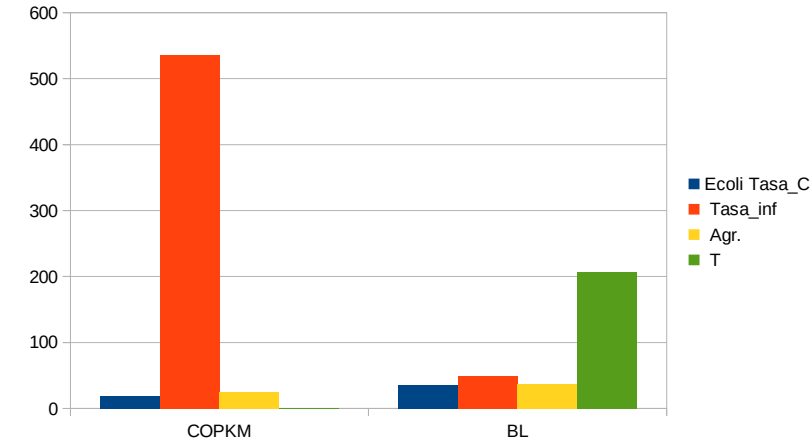
Tabla 6.3: Resultados globales en el PAR con 10% de restricciones

	Iris				**Ecoli**				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
COPKM	0,66	48,60	0,78	0	17,99	535,80	25,55	0,01	0,55	0,00	0,55	0
BL	1,72	0,00	1,72	1,02	35,51	49,20	36,16	207,36	1,40	0,00	1,40	1,08

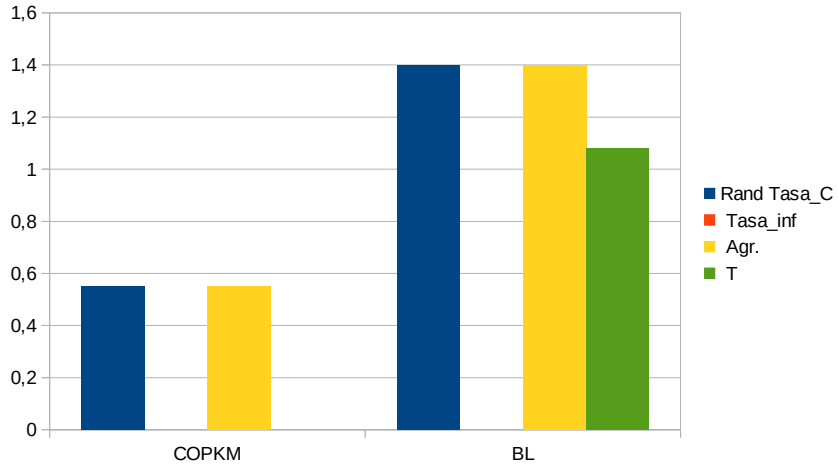
La Tabla 6.3 nos agrupa los valores medios de cada conjunto de ambos algoritmos. Comparemos cada conjunto en gráficas:



Esta gráfica compara los valores medios en el conjunto Iris para ambos algoritmos. Claramente podemos observar que la mayor diferencia es lo superior que es el valor de infeasibility en el algoritmo greedy.



Al comparar el conjunto de datos Ecoli observamos más claramente la diferencia de tiempos entre estos, siendo claramente más optimo el algoritmo greedy.



Finalmente en el conjunto Rand vemos claramente como el algoritmo greedy minimiza la desviación general y el tiempo. En este caso claramente es más óptimo el algoritmo greedy, ya que en ambos las restricciones incumplidas es igual a 0.

Tabla 6.4: Resultados obtenidos por el algoritmo BL en el PAR con 20% de restricciones

	Iris				**Ecoli**				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	1,78	0,00	1,78	1,43	33,71	46,00	34,02	286,95	1,42	0,00	1,42	1,29
Ejecución 2	1,76	0,00	1,76	1,41	36,76	13,00	36,85	281,05	1,43	0,00	1,43	1,43
Ejecución 3	1,70	0,00	1,70	1,78	34,11	23,00	34,27	316,07	1,43	0,00	1,43	1,64
Ejecución 4	1,83	0,00	1,83	1,77	36,06	5,00	36,09	304,30	1,51	0,00	1,51	1,26
Ejecuión 5	1,90	0,00	1,90	1,24	37,66	3,00	37,68	265,81	1,20	0,00	1,20	1,32
Media	1,79	0,00	1,79	1,53	35,66	18,00	35,78	290,84	1,40	0,00	1,40	1,39

Tabla 6.5: Resultados obtenidos por el algoritmo COPKM en el PAR con 20% de restricciones

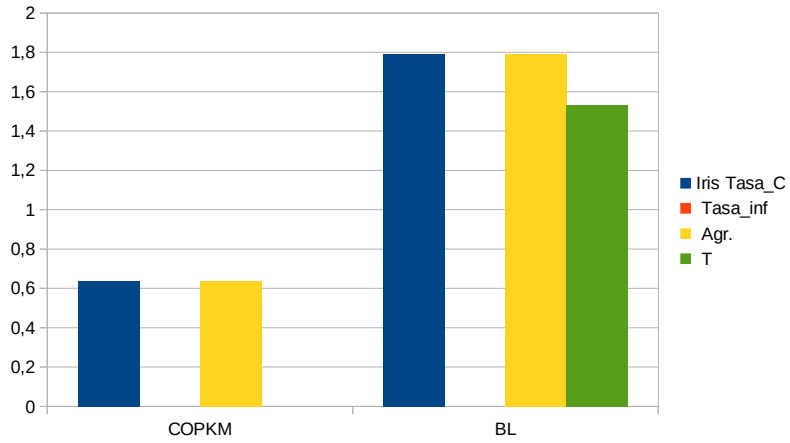
	Iris				**Ecoli**				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	0,64	0,00	0,64	0,00	19,98	134,00	20,90	0,01	0,55	0,00	0,55	0,00
Ejecución 2	0,64	0,00	0,64	0,00	19,97	227,00	21,59	0,01	0,55	0,00	0,55	0,00
Ejecución 3	0,64	0,00	0,64	0,00	15,69	327,00	18,01	0,01	0,55	0,00	0,55	0,00
Ejecución 4	0,64	0,00	0,64	0,00	15,43	787,00	21,05	0,01	0,55	0,00	0,55	0,00
Ejecuión 5	0,64	0,00	0,64	0,00	30,57	160,00	31,69	0,01	0,55	0,00	0,55	0,00
Media	0,64	0,00	0,64	0,00	20,33	327,00	22,65	0,01	0,55	0,00	0,55	0,00

A continuación observamos los resultados de los mismo conjuntos a los mismo algoritmos, pero esta vez con un conjunto de restricciones de un 20% respecto el total. Podemos observar situaciones muy similares que en los casos anteriores

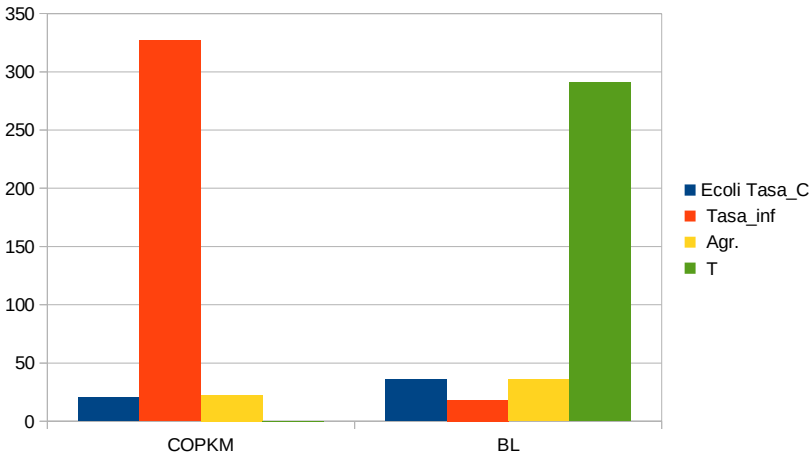
Tabla 6.6: Resultados globales en el PAR con 20% de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
COPKM	0,64	0,00	0,64	0	20,33	327,00	22,65	0,01	0,55	0,00	0,55	0
BL	1,79	0,00	1,79	1,53	35,66	18,00	35,78	290,84	1,40	0,00	1,40	1,39

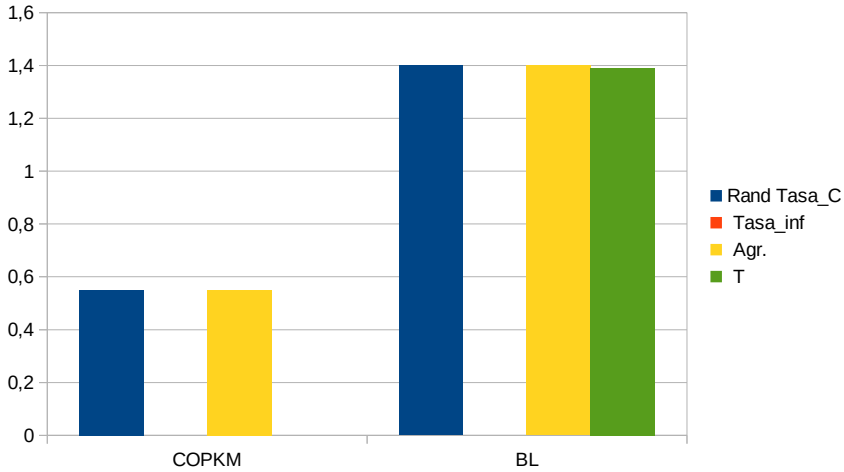
De nuevo presentamos una media de los valores obtenidos.



En este caso frente a los resultado con restricciones 10%, observamos que ya no visualizamos una notable diferencia entre la Tasa_inf (infeasibility) y se incremento la diferencia de tiempos de ejecución



En el conjunto Ecoli obtenemos unos resultados visualmente muy similares al mismo conjunto con restricciones 10%. Se conserva una gran diferencia en la Tasa_inf a la vez que aumenta la diferencia de tiempos de ejecución entre ambos, manteniéndose cómo más veloz el algoritmo greedy.



Finalmente al comparar los resultados de Rand observamos una clara similitud, de nuevo se establece que en este caso el más óptimo seria el algoritmo greedy, teniendo una menor desviación general y un numero de restricciones 0 en ambos casos.

8. Bibliografía

La bibliografía que he utilizado para desarrollar la práctica ha sido basicamente la que aparece en la web de la asignatura, tanto el guión como las diapositivas del seminario 2 de prácticas sobre los poblemas PAR y MDP.

También he obtenido un gran información sobre el uso de vectores de http://www.cplusplus.com/