Practica 3.b
Técnicas de Búsqueda por Trayectorias para el Problema del Agrupamiento con Restricciones

Curso 2019-2020

Tercer Curso del Grado en Ingeniería Informática

Metaheurísticas

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

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 practica con los vectores he preferido su utilización.

Para representar las soluciones que voy evaluando durante la ejecución de los algoritmos, he utilizado un vector de tipo “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.
También defino una variable de tipo “int” llamada “evaluaciones”, la cual me permite llevar la cuenta del número de evaluaciones realizadas, de esta forma estableceremos como condición de parada el llegar hasta el número de evaluaciones máximas establecidas (10000 en nuestro caso).

3.2. Calcular Violaciones

A continuación establezco una serie de funciones que me ayudarán al calculo de la función objetivo. La primera de éstas me permite obtener el número total de restricciones que se incumplen en el cromosoma o individuo que estamos evaluando.
La 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.

3.3 Mayor Distancia

Presento esta función de nuevo porque en la práctica anterior realicé de forma incorrecta el cálculo. 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 distanciaMedia(centroides, individuo):
        cluster_actual = 0
        suma_distancia = 0
        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
                        FIN-SI
```

FIN-PARA

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

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

3.4 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. Su pseudocógido es sencillo y ya fue proporcionado en la práctica anterior.

3.5 Función Objetivo

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 funcionObjetivo():

mayor_distancia = distanciaMaxima()

violaciones_realizadas = calcularRestricciones(individuo)

n_violaciones = totalViolaciones(individuo)

media = distanciaMedia(centroides, individuo)

landa = cociente mayor_distancia entre n_violaciones

valoración =+ violaciones_realizadas * landa

Devuelvo valoración con la valoración de nuestra solución

Fin

3.6 Generación de soluciones aleatorias

La función “solucionInicial()” nos permite generar una solución inicial aleatoria, siguiendo la estructura fundamental del problema, es decir, la solución será un vector en el cual cada posición tomará un valor en el rango de {0, …, k-1}, siendo “k” el número total de cluster.
La función devolverá un vector del tipo “int” con esta solución aleatoria.

En pseudocógido, el algoritmo es el siguiente:

```
FUNCIÓN solucionInicial():
        PARA cada dato de nuestro conjunto de datos X
                        solucion_inicial ← Añadir entero aleatorio [0, n_cluster-1]
        FIN-PARA
        DEVOLVER evitarClusterVacio(solucion_inicial)
FIN
```

La función “evitarClusterVacio()” utilizada en la función anterior, es una función que recibe por parámetro una posible solución, y se encarga de comprobar que en ella no hay ningún cluster vacío, es decir, no hay ningún cluster sin ningún elemento asignado a él. Para evitarlo, esta función recorre los valores de cluster asignados, y en el caso de encontrar un cluster vacío tomará una posición aleatoria de nuestra solución, y a ese dato aleatorio le asignará como nuevo valor el cluster vacío.

3.6 Operador de vecino

La función “cambioCluster()” me permite la generación de una solución vecina. Este vecino se generará escogiendo aleatoriamente un elemento de mi vector de soluciones, y aplicándole un cambio al valor dicho elemento, es decir, cambiaremos el cluster asignado a este elemento por otro cluster. Esta función recibe por parámetros un vector de tipo “int” con la solución de la que partimos para generar el vecino, y el pseudocódigo es el siguiente:

```
FUNCIÓN cambioCluster(solucion):
        nuevo_cluster = entero aleatorio [0, n_cluster-1]
        //Tomo un elemento aleatorio de mi vector solucion
        elemento = entero aleatorio [0, solucion.size()-1]
        MIENTRAS nuevo_cluster == cluster_actual
                nuevo_cluster = entero aleatorio [0, n_cluster-1]
        FIN-MIENTRAS
        solucion[elemento] = nuevo_cluster
        DEVOLVER evitarClusterVacio(solucion)
FIN
```

Observamos que al final realizamos una comprobación de que este cambio no ha provocado que un cluster quede vacío.

4. Descripción esquema de búsqueda en cada algoritmo.

En este apartado trataremos la implementación de los algoritmos desarrollados en esta práctica.

4.1. Enfriamiento simulado (ES)

La primera técnica que hemos implementado en esta práctica es el Enfriamiento Simulado. Es una búsqueda basada en trayectorias inspirada en la termodinámica. Al igual que en la Búsqueda Local, esta técnica explora el entorno de la solución con el objetivo de encontrar una mejor solución. Sin embargo, la diferencia frente a la Búsqueda Local, esta técnica puede seleccionar soluciones peores que la actual respecto a una cierta probabilidad. De esta forma hacemos frente a uno de los problemas de la BL, la caída en óptimos locales.

Para llevar a cabo este procedimiento nos basamos en un modelo termodinámico en el que hay una "temperatura" que va disminuyendo (enfriamiento), de forma que a mayor temperatura, mayor probabilidad de aceptar soluciones peores. Al aumentar las iteraciones disminuimos esta probabilidad de aceptar peores soluciones que la actual. Por tanto, es una técnica que nos permite explorar más al principio de la ejecución, ya que aceptará peores soluciones, lo que nos permitirá salir de óptimos locales, pero esto no ocurrirá al final.

El esquema de enfriamiento seguido es el esquema modificado de Cauchy, el cual nos permite realizar enfriamientos de forma proporcional a un parámetro β. Observamos este esquema en el siguiendo pseudocódigo:

```
FUNCION cauchyModificado(T_K, β)
        T_{K+1} = T_K / (1 + β * T_K)
        DEVOLVER T_{K+1}
FIN
```

Este parámetro β depende de la temperatura inicial, de la temperatura final que se desea alcanzar y de un valor M, que corresponde al número de enfriamientos a realizar. Su calculo lo podemos observar en el siguiente pseudocódigo:

```
FUNCION calcularBeta(T_0, T_f, M)
        β = ( T_0 - T_f) / (M * T_0 * T_f)
        DEVOLVER β
FIN
```

Finalmente la temperatura inicial se calcula partiendo de C, que es el coste de la solución inicial y de $\Phi \in [0,1]$, que es la probabilidad de aceptar un μ por 1 peor que la inicial. En este problema partimos de $\Phi = \mu = 0.3$

```
FUNCION calcularTemperaturaInicial(C)
        T_0 = ( μ * C) / - ln( Φ )
        DEVOLVER T_0
FIN
```

Importante mencionar que la generación de vecinos se realiza mediante la función anteriormente definida “cambiarCluster()”, que recordemos que cambia el cluster asignado a un elemento aleatorio del vector solución.

Recordar antes de presentar el pseudocódigo que el número de evaluaciones máximas establecido es de 100000. Una vez comentadas las bases del algoritmo, el pseudocódigo es el siguiente:

```
FUNCION enfriamientoSimulado()
        solucion_actual = solucionInicial()
        valoracion_actual = funcionObjetivo(solucion_actual)
        mejor_solucion =  solucion_actual
        mejor_valoracion = valoracion_actual
        T0 = calcularTemperaturaInicial(valoracion_actual)
        T = T0
        MIENTRAS Tf > T0
                T = T *  Tf
        FIN-MIENTRAS
        max_vecinos = 10 * datos.size();
        max_exitos = 0.1 * max_vecinos;
        M = 100000 / max_vecinos;  //Recordemos que 100000 son las evaluaciones máximas
        beta = calcularBeta(T0,  M);
        exitos = 1
        evaluaciones = 0
        MIENTRAS exitos != 0 && T > Tf
                exitos = 0
                vecinos = 0
                MIENTRAS exitos < max_exitos && vecinos < max_vecinos &&
                                                        evaluaciones <  100000
                nueva_solucion = cambioCluster(solucion_actual)
                nueva_valoracion = funcionObjetivo(nueva_solucion)
                Incremento el valor de evaluaciones en 1
                Incremento el valor de vecinos en 1
                  ∇f  =  nueva_valoracion – valoracion_actual
                //Rand genera un numero real aleatorio entre 0 y 1
                SI   ∇f  < 0 || Rand() <=   e^(−∇f/T)
                        Incremento el valor de exitos en 1
                        solucion_actual = nueva_solucion
                        valoracion_actual = nueva_valoracion
                        SI valoracion_actual < mejor_valoracion
                                mejor_solucion = solucion_actual
                                mejor_valoracion = valoracion_actual
                        FIN-SI
                FIN-SI
        FIN-MIENTRAS
        T = cauchyModificado(T, beta)
        DEVOLVER mejor_solucion
FIN
```

4.2. Búsqueda Multiarranque Básica (BMB)

El siguiente algoritmo tiene como base central el algoritmo implementado en la primera práctica, la Búsqueda Local. Nuestro algoritmo de BMB consistirá en ejecutar la BL sobre diez soluciones iniciales generadas de forma aleatoria, mediante la función ya definida anteriormente "solucionInicial()", y nos quedaremos con aquella solución con la que mejor valoración obtengamos (menor valor nos de la función objetivo sobre esta).

Recordemos que nuestra BL está basada en el primer mejor. Recordemos que la idea de este algoritmo es llamar a nuestra función "cambioCluster()", nuestro operador de generación de vecinos, en cada iteración, que como ya hemos descrito anteriormente, tomará un componente aleatorio del vector solución y cambiará el cluster que esta tiene asignado. Si tras realizar este cambio la función objetivo mejora, nos quedaremos con el cambio realizado, si es el caso contrario, desecharemos el cambio.

Para decidir sobre que componente realizar ejecutar "cambioCluster()" tenemos un vector de índices de un tamaño idéntico al vector solución, el cual barajamos de forma aleatoria y recorremos secuencialmente. En el caso de llegar al final se volverá a barajar para seguir con la generación de nuevas soluciones.

Adaptaremos la BL para nuestro algoritmo BMB, para ello establecemos que nuestro algoritmo de BL reciba por parámetros el vector de la solución aleatoria generada y cómo criterio de parada estableceremos que se desarrollen 10000 iteraciones o que las soluciones que vamos obteniendo no mejoren en el tamaño del entorno de una solución, que es n*(k-1), siendo n el tamaño de nuestra solución y k el número total de clúster. El pseudocodigo de la Búsqueda Local es el siguiente:

```
FUNCION busquedaLocal(solucion_actual)
        evaluaciones = 0
        vecinos = 0
        mejora = false
        n = datos.size()
        valoracion_actual = funcionObjetivo(solucion_actual)
        mejor_solucion = solucion_actual
        mejor_valoracion = valoracion_actual
        //Inicializo el vector de índices
        PARA i HASTA i < datos.size()
                indices[i] = i
        FIN-PARA
        //Recordemos que el tamaño del entorno es n*(k-1)
        //n_cluster es el número total de cluster
        MIENTRAS evaluaciones < 10000 && vecinos < n * (n_cluster - 1)
                SI evaluaciones % n || mejora == true
                        shuffle(indices)
                        mejora = false
                FIN-SI
                elemento = indices[evaluaciones % tamanio]
                nueva_solucion = cambioCluster(solucion_actual, elemento)
                nueva_valoracion = Evalua.funcionObjetivo(nueva_solucion)
```

```
            Incremento el valor de evaluaciones en 1
            SI valoracion_actual < mejor_valoracion
                solucion_actual = nueva_solucion
                mejor_solucion = solucion_actual
                mejor_valoracion = valoracion_actual
                mejora = true
                vecinos = 0
            FIN-SI
            Incremento el valor de evaluaciones en 1
        FIN-MIENTRAS
        DEVOLVER mejor_solucion
FIN
```

Recordamos que en la función “cambioCluster()” tenemos en cuenta que ningún cluster quede vacío.

Una vez definida la función de Búsqueda Local, la implementa del algoritmo Búsqueda Multiarranque Básica es muy sencilla. El pseudocódigo es el siguiente:

```
FUNCION busquedaMultiarranqueBasica()
        iteracion = 0
        MIENTRAS iteracion < 10
            solucion_actual = solucionInicial()
            nueva_solucion = busquedaLocal(solucion_actual)
            nueva_valoracion = funcionObjetivo(nueva_solucion)
            SI nueva_valoracion < mejor_valoracion
                mejor_valoracion = nueva_valoracion
                mejor_solucion = nueva_solucion
            FIN-SI
            Incremento el valor de iteracion en 1
        FIN-MIENTRAS
        DEVOLVER mejor_solucion
FIN
```

4.3. Búsqueda Local Reiterada (ILS)

Esta técnica consiste en la aplicación repetida del algoritmo de Búsqueda Local a una solución inicial aleatoria inicialmente, y en los demás casos se realizará la BL sobre una solución obtenida por mutación brusca de un óptimo previamente encontrado. En nuestro caso ejecutaremos la BL diez veces, cada una con un máximo de 10000 iteraciones.

Para la mutación hemos utilizado un operador de mutación por segmento, en el que el cambio consiste en reasignar de forma aleatoria los valores de los elementos asociados a unas posiciones contenidas en un segmento, que será el segmento contrario a uno que nosotros escojamos mediante la elección de un inicio y tamaño de segmento de forma aleatoria, es decir, realizaremos el cambio sobre las posiciones nos contenidas en el segmento aleatorio creado, pero solamente sobre un 10% de ellas respecto el total de elementos (v = 0.1 * n).

El pseudocódigo del operador de mutación es el siguiente:

```
FUNCION mutarBrusco(solucion)
        numero_mutaciones = 0.1 * tamaño de nuestro conjunto de datos X
        //genero r y v
        inicio_segmento = entero aleatorio en el rango (0, solucion.size() - 1)
        tamanio_segmento = entero aleatorio en el rango (0, solucion.size() - 1)
        //Creo un array con las posiciones que forman parte del segmento
        PARA i = 0 mientras i < tamanio_segmento incremento i en 1
                //Hago el modulo para que no supere el tamaño permitido
                Añadir (inicio_segmento + i) % solucion.size() en vector segmento
        FIN-PARA
        PARA cada elemento que forma mi vector solución
                //Recojo los elementos que son parte del vector y no puedo modificar
                SI el elemento forma parte del segmento
                        salida[elemento] = solucion[elemento]
                SINO
                        Añadir elemento al vector resto                 //resto = vector tipo int
                FIN-SINO
        FIN-PARA
        //Selecciono un 10% del total de elementos del vector solución
        seleccion = RandVector(resto, numero_mutaciones)        //seleccion = vector tipo int
        PARA cada elemento que forma mi vector solución
                //Tomo los elementos que son parte del resto y de los seleccionados aleatoriamente
                SI el elemento forma parte de resto
                        SI el elemento forma parte de seleccion
                                nuevo_cluster = entero aleatorio en el rango (0, n_cluster – 1)
                                //Repito hasta que tenga un valor diferente
                                MIENTRAS cluster_actual == nuevo_cluster
                                        nuevo_cluster = entero aleatorio en el rango (0, n_cluster – 1)
                                FIN-MIENTRAS
                                salida[elemento] = nuevo_cluster
                        SINO
                                salida[elemento] = salida[elemento]
```

```
                    FIN-SINO
            FIN-SI
    FIN-PARA
    DEVOLVER evitarClusterVacio(salida)
FIN
```

Una vez tenemos nuestro operador de mutación podemos definir nuestro algoritmo ILS, esta búsqueda está basada en trayectorias que, a diferencia de la búsqueda local, permite evitar óptimos locales, reiniciando la búsqueda con una nueva solución inicial. Partimos de una solucón inicial sobre la que aplicamos una búsqueda local, guardando la valoración del resultado como la mejor solución hasta el momento. A continuación se aplica la mutación definida anteriormente sobre la mejor solución actual, lo cual produce que se modifiquen algunos valores de la solución y tras ello volvemos a aplicar búsqueda local sobre esta modificación. Tras esta BL comparamos la voloración obtenida con la mejor actual, y si la nueva es mejor se convertirá en la nueva mejor solución actual. Repetiremos este proceso hasta cumplir el criterio de parada, que en este caso es ejecutar la búsqueda local un total de 10 veces.
Respecto al criterio de parada de cada búsqueda local es el mismo que el mencionada en el apartado anterior, que se desarrollen 10000 iteraciones o que las soluciones que vamos obteniendo no mejoren en el tamaño del entorno de una solución, que es n*(k-1), siendo n el tamaño de nuestra solución y k el número total de clúster.
Finalmente el pseudocódigo es el siguiente:

```
FUNCIÓN busquedaLocalReiterada()
    //Creo una solución inicial aleatoria
    solucion_inicial = solucionInicial()
    //Aplico BL y obtengo su valoración
    mejor_solucion = busquedaLocal(solucion_actual)
    mejor_valoracion = funcionObjetivo(mejor_solucion)
    it = 0
    //Aplicamos 10 veces la BL, ya lo hemos hecho una vez
    MIENTRAS it < 9
        nueva_solucion = mejor_solucion
        nueva_solucion = mutarBrusco(nueva_solucion)
        nueva_solucion = busquedaLocal(nueva_solucion)
        nueva_valoracion = Evalua.funcionObjetivo(nueva_solucion)
        SI nueva_valoracion < mejor_valoracion
            mejor_valoracion = nueva_valoracion
            mejor_solucion = nueva_solucion
        FIN-SI
        Incremento el valor de it en 1
    FIN-MIENTRAS
    DEVOLVER mejor_solucion
FIN
```

4.4. Algoritmo Híbrido (ILS-ES)

Este algoritmo tiene exactamente la misma estructura que el algoritmo anterior, ILS, la única diferencia es que el algoritmo de búsqueda por trayectorias simples, que será utilizado para refinar las soluciones iniciales, en este caso no será la búsqueda local como en el caso anterior, será el algoritmo ES definido en el apartado 3.1.

De nuevo ejecutaremos 10 veces nuestro algoritmo de ES con un número de evaluaciones de 10000, una vez se han realizado ese número de iteraciones finalizará la ejecución del algoritmo de enfriamiento simulado.

El pseudocódigo es el siguiente:

```
FUNCIÓN busquedaLocalReiterada()
        //Creo una solución inicial aleatoria
        solucion_inicial = solucionInicial()
        //Aplico ES y obtengo su valoración
        mejor_solucion = enfriamientoSimulado(solucion_actual)
        mejor_valoracion = funcionObjetivo(mejor_solucion)
        it = 0
        //Aplicamos 10 veces la ES, ya lo hemos hecho una vez
        MIENTRAS it < 9
                nueva_solucion = mejor_solucion
                nueva_solucion = mutarBrusco(nueva_solucion)
                nueva_solucion = enfriamientoSimulado(nueva_solucion)
                nueva_valoracion = Evalua.funcionObjetivo(nueva_solucion)
                SI nueva_valoracion < mejor_valoracion
                        mejor_valoracion = nueva_valoracion
                        mejor_solucion = nueva_solucion
                FIN-SI
                Incremento el valor de it en 1
        FIN-MIENTRAS
        DEVOLVER mejor_solucion
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 6 archivos .cpp, situados en la carpeta “bin” 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”

trayectorias_simples.cpp/.h: en este archivo están definidos los métodos necesarios para ejecutar el algoritmo de enfriamiento simulado (ES) ya definido.

trayectorias_multiples.cpp/.h: en este caso encontramos los métodos necesarios para ejecutar los algoritmos de búsqueda multiarranque básica (BMB) y búsqueda local reiterada (ILS) junto con la búsqueda local (BL) necesario en ambos algoritmos.

hibridos.cpp/.h: en este archivo están definidos los métodos necesarios para ejecutar el algoritmo de ILS-ES ya definido.

evaluacion.cpp/.h: encontramos las diferentes funciones que nos permiten ejecutar la función objetivo, además se encarga de ir incrementando el número de evaluaciones realizadas.

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 conjunto de datos de iris → util.leerArchivoMatriz(“iris”)
        //Inicializo ES
        trayectorias_simples ts(n_cluster, datos, restricciones);
        //inicializo BMB y ILS
        trayectorias_multiples tm(n_cluster, datos, restricciones);
        //inicializo ILS-ES
        hibrido h(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.
- Newthy Roid: contiene medidas cuantitativas tomadas sobre la glándula tiroides de 215 pacientes. Presenta 3 clases, es decir, número de clusters es 3.

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 ejecución comenzando con la Ejecución 1 hacia la 5, han sido: “49137372”, “491373”, “4913”, “49”, “100”.

Solamente adjunto las tablas que contienen la media de los resultados, la demás tablas se encuentran en el archivo “tablas_experimentos” situada en la misma carpeta. La causa es la gran cantidad.

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

Tabla 6.13: Resultados globales en el PAR con 10% de restricciones

	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	0,11	380,00	0,18	0,54	2,24	2721,00	6,86	9,17	0,1	574,80	0,2	0,54	1,29	1001,40	2,58	3,22
BL	0,11	0,00	0,11	0,10	4,98	56,00	6,40	4,85	0,12	0,00	0,12	0,09	2,54	17,40	3,12	0,29
ES	0,11	0,00	0,11	3,96	4,87	30,00	5,63	7,74	0,12	0,00	0,12	3,91	2,65	0,00	2,65	2,74
BMB	0,11	0,00	0,11	0,84	6,28	97,80	8,76	17,75	0,12	0,00	0,12	0,81	2,65	0,00	2,65	2,71
ILS	0,11	0,00	0,11	0,51	4,62	27,20	5,30	17,62	0,12	0,00	0,12	0,44	2,65	0,00	2,65	1,3
ILS-ES	0,11	0,00	0,11	3,68	7,36	211,40	12,71	10,08	0,12	0,00	0,12	3,68	2,65	0,00	2,65	5,64

Tabla 6.14: Resultados globales en el PAR con 20% de restricciones

	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	0,1	750,40	0,15	1,37	3,06	5008,80	11,94	21,13	0,1	1234,20	0,22	0,74	1,29	2031,60	2,66	5,16
BL	0,11	4,80	0,12	0,12	4,43	47,60	5,05	5,48	0,12	0,00	0,12	0,13	2,65	0,00	2,65	0,38
ES	0,11	0,00	0,11	4,93	4	45,60	4,59	9,63	0,12	0,00	0,12	4,8	2,65	0,00	2,65	3,57
BMB	0,11	0,00	0,11	0,92	5,76	111,60	7,21	24,06	0,12	0,00	0,12	0,92	2,65	0,00	2,65	3,09
ILS	0,11	0,00	0,11	0,58	4,37	50,60	5,03	23,43	0,12	0,00	0,12	0,58	2,65	0,00	2,65	1,74
ILS-ES	0,11	0,00	0,11	4,7	7,76	341,60	12,21	13,68	0,12	0,00	0,12	4,71	2,65	0,00	2,65	7,64

Comentar que en las valoraciones obtenidas tanto en Ecoli como Newthy Roid influye su elevalo valor de mayor distancia presente en el conjunto. En el caso de Ecoli la distancia entre los puntos más separados es de 150.99 y en el caso de Newthy Roid de 84.11.

Dado que debemos analizar una gran cantidad de algoritmos y datos, vamos a estructurar el análisis por bloques, en los que se compararán y explicarán distintos aspectos de los experimentos realizados:

1. **Análisis de la Tasa_C**

Recordemos que la Tasa_C corresponde a la desviación general de la partición obtenida por cada método en cada ejecución en el conjunto de datos correspondiente.

Si comenzamos comparando los resultados de los algoritmos de trayectorias simples podemos observar que en el caso de Ecoli el algoritmo ES llega a mejorar los resultados obtenidos mediante BL. La responsable de esta mejora puede ser la capacidad de salir de óptimos locales, permitiendo empeorar la solución con más probabilidad al inicio y disminuyéndola durante la ejecución.

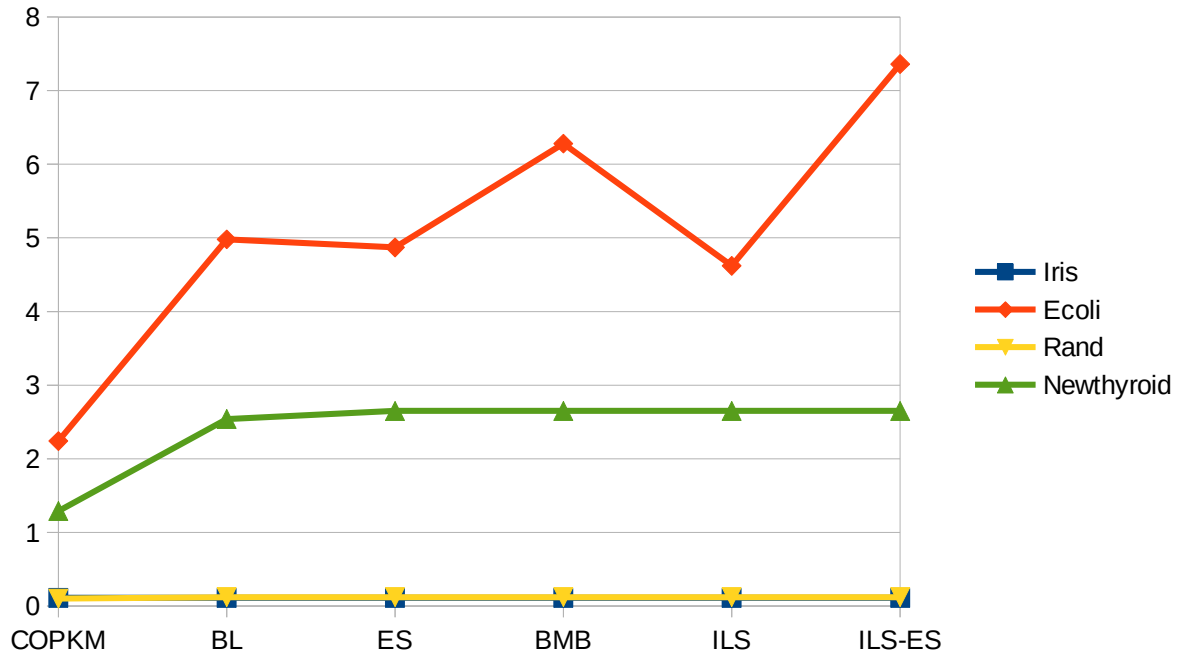
Si incluimos ahora los algoritmos de trayectorias múltiples en nuestra valoración de resultados podemos observar como los mejores resultados de distancia C (menos en Newthtroid) se obtienen con ILS, cosa que puede deberse a que este algoritmo tiene lo bueno de la BL que es la intensificación y además le añade algo de exploración al hacer BL desde puntos de arranque diferentes con mutaciones bruscas de las soluciones obtenidas en cada búsqueda local. Frente a la mejora de resultados veremos un aumento de tiempo de ejecución de ILS frente a la BL.

Respecto a BMB la causa de que aunque apliquemos 10 veces BL no mejoren los resultados o al menos no sean muy similares a los obtenidos por la BL en solitario es el número de iteraciones máximas, que disminuye en gran cantidad el espacio de búsqueda en el caso de BMB, siendo de 10000 iteraciones frente 100000 en la BL.

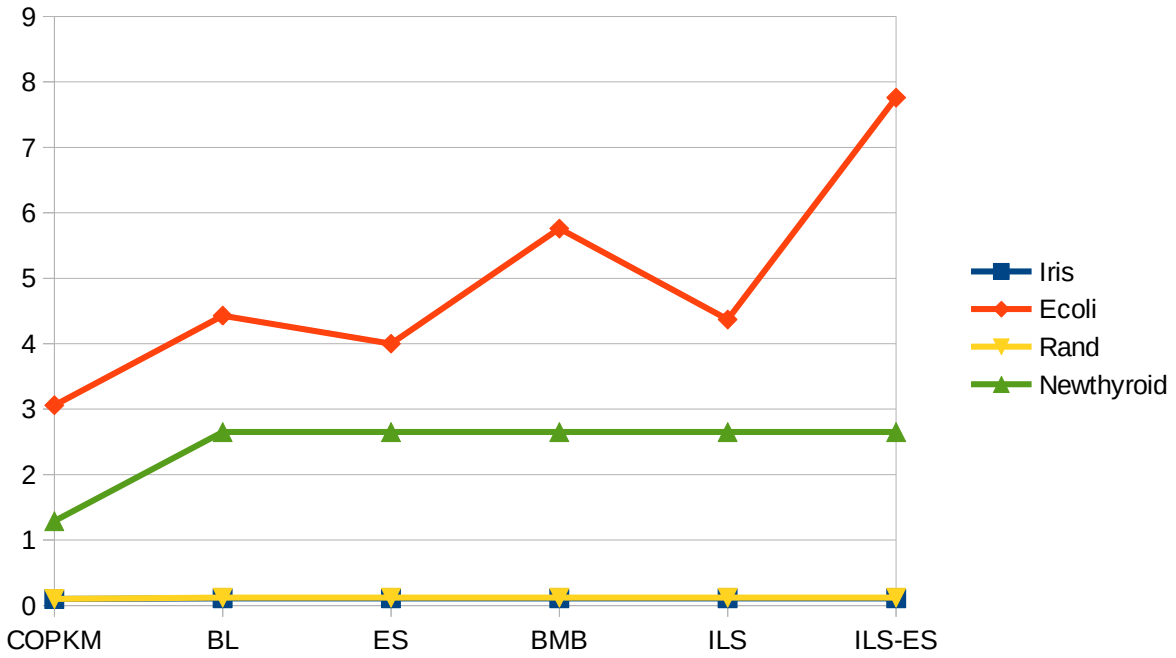
Con ILS-ES obtendremos una distancia C idéntica a las demás trayectorias, menos en el caso del conjunto de datos Ecoli, creo que se debe a un enfriamiento demasiado rápido.

Finalmente la mayor tasa_C será ocasionada por el algoritmo Greedy, algo que se debe a que este primero da prioridad a que no existan violaciones de restricciones.

Tasa_C con 10% restricciones



Tasa_C con 20% restricciones



Estas gráficas con los valores resultantes de Tasa_C nos permite comprobar visualmente lo comentado anteriormente.

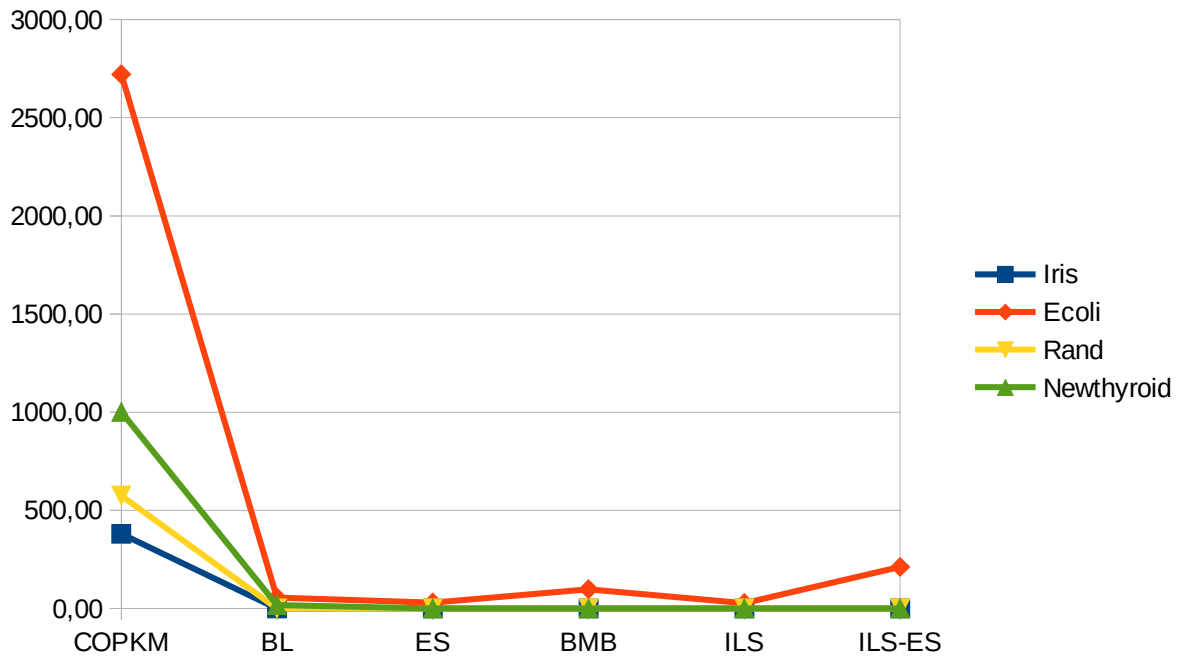
2. Análisis de la Tasa_inf

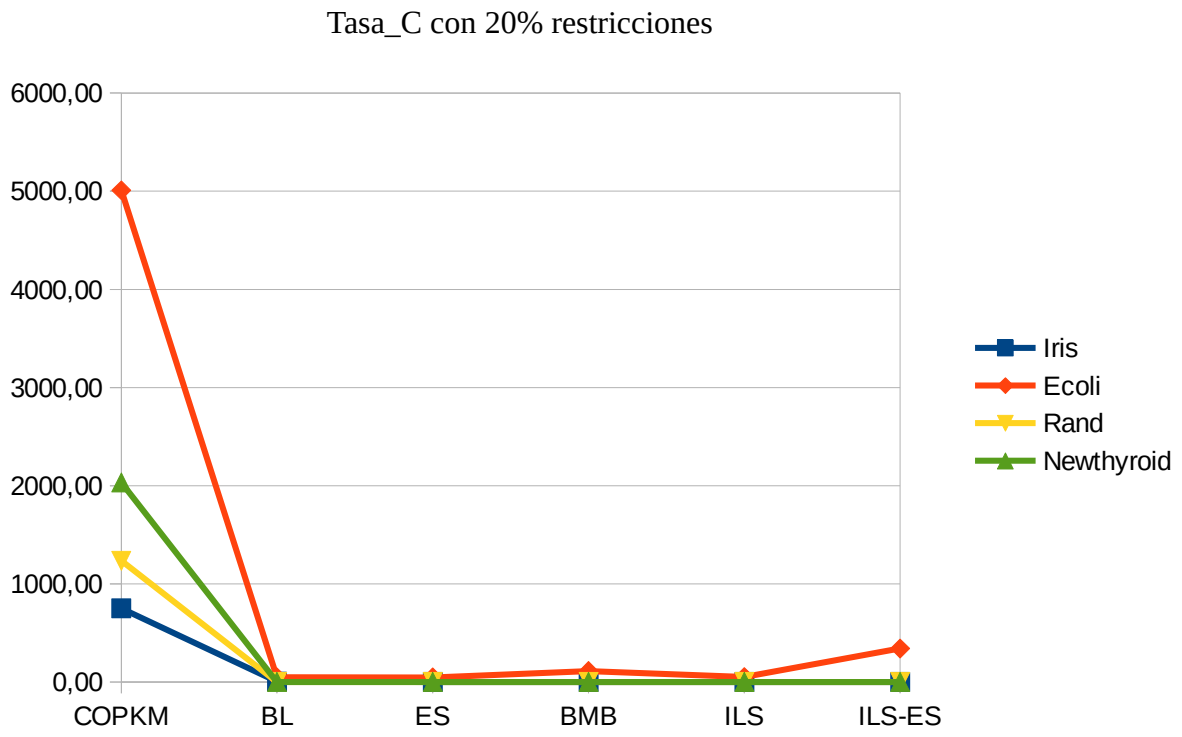
Recordemos que la Tasa_inf corresponde al valor de “infeasability”, es decir, el número de restricciones violadas en la solución obtenida.

Observamos que a pesar de que el algoritmo ES sea capaz de disminuir este valor frente al algoritmo de BL, cuando pasamos a las trayectorias múltiples e híbrida, el algoritmo ILS-ES que utiliza ES para refinar las soluciones iniciales, obtiene peores resultados que el algoritmo ILS que utiliza BL, esto puede estar ocasionado por el número de iteraciones, ya que al ejecutarlos solos, como trayectorias simples, el número máximo de iteraciones es de 100000 mientras que al ejecutarlos en trayectorias múltiples este valor se ve reducido a 10000. Esto ocasiona que no se alcance un valor tan óptimo como el obtenido por las trayectorias simples.

Por esta misma causa de las iteraciones el valor de “infeasability” en BMB es tan grande.

Tasa_infcon 10% restricciones





Podemos observar en las gráficas que representan los diferentes resultados de Tasa_inf como la mayor siempre es obtenida por el algoritmo Greedy, y como esta es muy similar en los demás algoritmos.

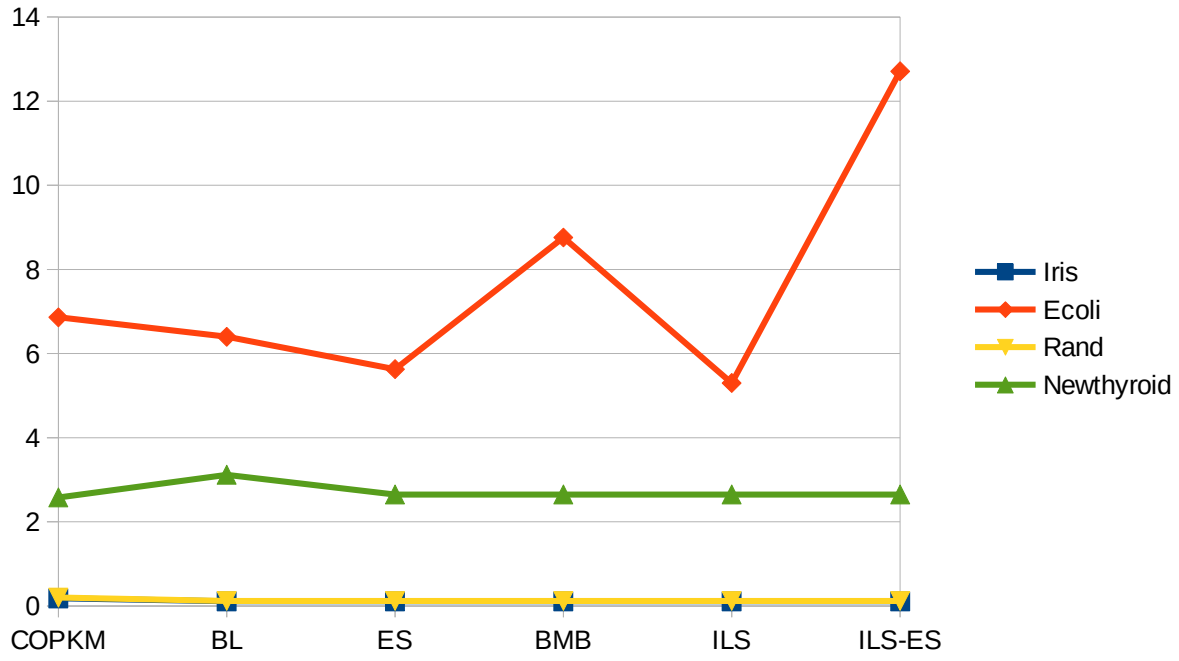
3. Análisis del Agregado

Recordamos que el Agregado corresponde al valor obtenido por la función objetivo definida, f = C + (infeasability * λ). Siendo C la desviación general y λ el cociente entre la distancia máxima existente en el conjunto de datos y el número de restricciones presentes en el problema.

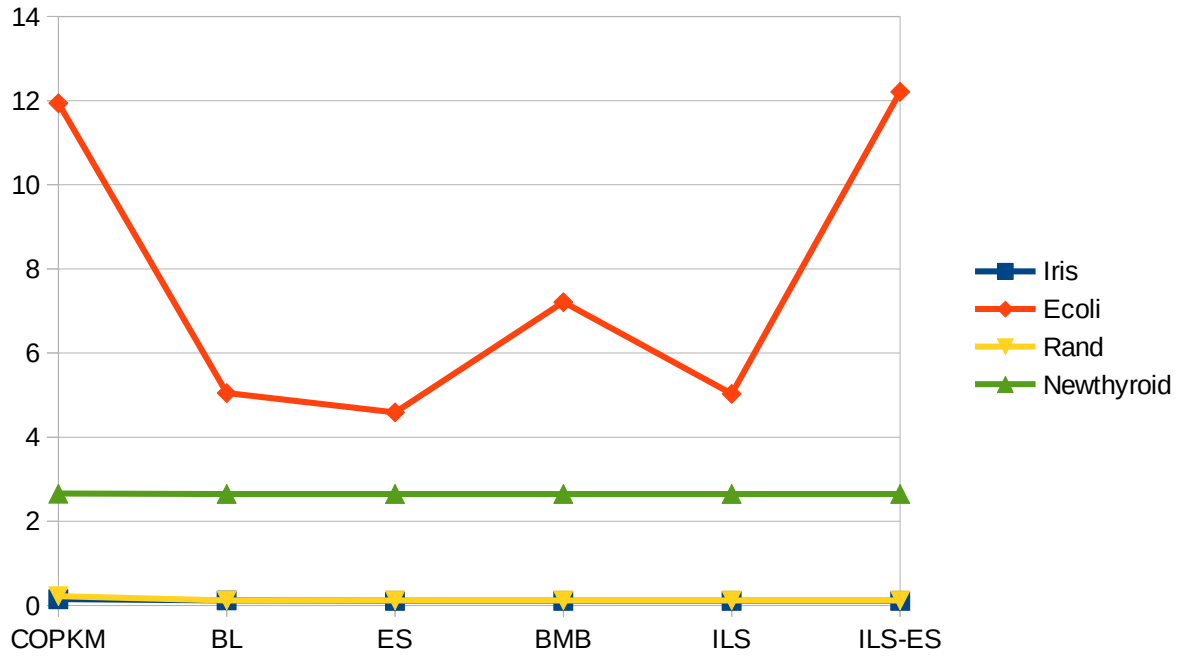
De nuevo la mayor diferencia la notaremos observando el conjunto de datos Ecoli, al igual que con la Tasa_C observamos primero las trayectorias simples, donde ES ha obtenido mejores resultados que la BL, sin embargo en las trayectorias múltiples la agregación de BMB se ve muy afectada, obteniendo la peor valoración, sólo siendo superada por el algoritmo ILS-ES y Greedy. La BL no funciona bien en el algoritmo BMB, seguramente porque son BL demasiado pequeñas de solo 10000 iteraciones. La BL si funciona en el algoritmo ILS, el cual obtiene el mejor resultado de todos por la misma causa explicada al tratar los resultados de la Tasa_C.

En el caso del algoritmo híbrido ILS-ES este nos aportará resultados similar a ES, mejores en algunos casos, cómo cuando se aplica sobre el conjunto Newthtroid, y en Rand e Iris similares a los obtenidos por los demás algoritmos, sin embargo, éste tendrá malos resultados en Ecoli seguramente ocasionado por un enfriamiento rápido en un conjunto de datos muy grande.

Agregación con 10% restricciones



Agregación con 20% restricciones

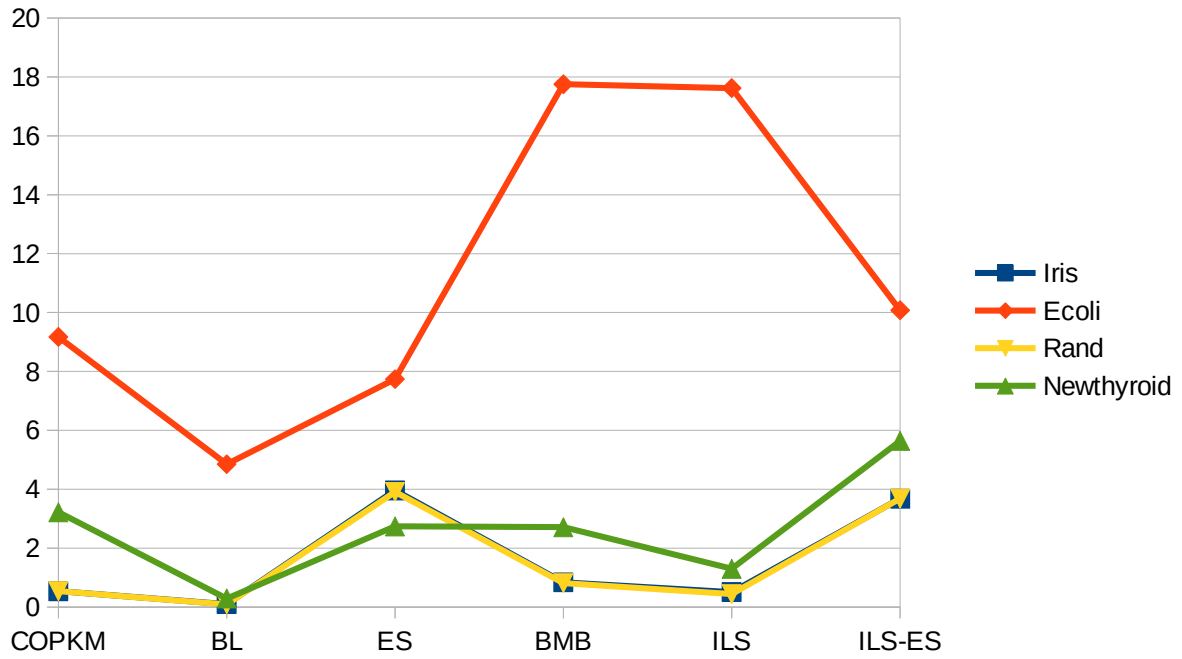


En estas gráficas representamos los resultados de agregación, observamos que en los conjuntos iris y rand obtenemos los resultados más bajos, y posiblemente los óptimos en todos sus casos.

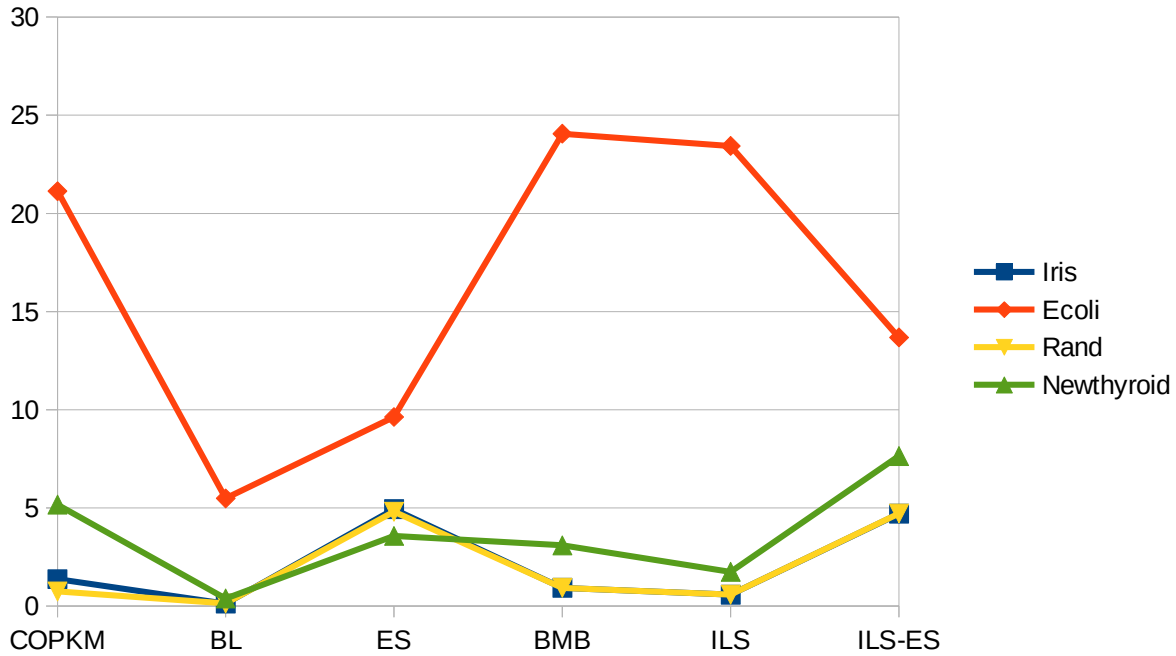
4. Análisis del Tiempo

Observando las gráficas anteriores podemos ver que ILS y ES, obtienen resultados similares pero con una gran diferencia de tiempo de ejecución, sobretodo en el caso de ILS, por lo que podríamos decir que es preferible la BL. En el caso de ILS esta diferencia de tiempo es notable sobretodo cuanto más grande es el conjunto de datos, así que para conjuntos pequeños ILS podría ser mejor que BL, ya que nos da un valor de agregación mejor. Si el tiempo tuviera menor importancia podríamos decir que ES es mejor que la BL, ya que la agregación obtenida es mejor con ES.

Tiempo con 10% restricciones



Tiempo con 20% restricciones



8. Bibliografía

La bibliografía que he utilizado para desarrollar la práctica ha sido básicamente la que aparece en la web de la asignatura, tanto el guión como las diapositivas del seminario 3 de prácticas sobre los poblemas PAR y MDP.

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