Practica 4.b
Galactic Swarm Optimization 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:

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

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

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

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.

En el caso de nuestro nuevo algoritmo GSO añadiremos una capa más a nuestro vector, es decir, en este caso tendremos un vector de vectores de vectores del tipo double, siendo el primer nivel del vector cada una de las Galaxias hasta M galaxias totales, cada una de ellas formada por Estrellas hasta N estrellas totales, y siendo cada una de estas estrellas una posible solución a nuestro problema. Este ultimo vector para representar nuestras soluciones será idéntico al comentado en el párrafo anterior, solo que será del tipo “double” en lugar de “int”, por motivos de calculo de velocidad que trataremos más adelante.

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 clúster 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 :

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

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 “inicioAleatorio()” 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 inicioAleatorio(vector):
    PARA cada dato hasta el tamaño total del vector
            vector ← Añadir entero aleatorio [0, n_cluster-1]
    FIN-PARA
    DEVOLVER evitarClusterVacio(vector)
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. Galactic Swarm Optimization.

4.1. Introducción

La metaheuristica Galactic Swarm Optimization (GSO) está inspirada en el movimiento de las estrellas, galaxias, cúmulos de galaxias, conjunto de cúmulos..., bajo la influencia de la gravedad. Observamos que esta idea posee varios niveles a los que podría expandirse, pero en GSO se queda en el segundo nivel, es decir, en el movimiento de estrellas y galaxias.

En GSO empleamos múltiples ciclos de exploración y fases de explotación en busca de un equilibrio óptimo entre la exploración de nuevas soluciones y la explotación de dichas soluciones. Tanto en exploración como en explotación se utiliza el algoritmo PSO (Particle Swarm Optimization), el cual está basado en el movimiento de una nube de partículas. El movimiento de cada partícula vendrá determinado por la siguiente fórmula:

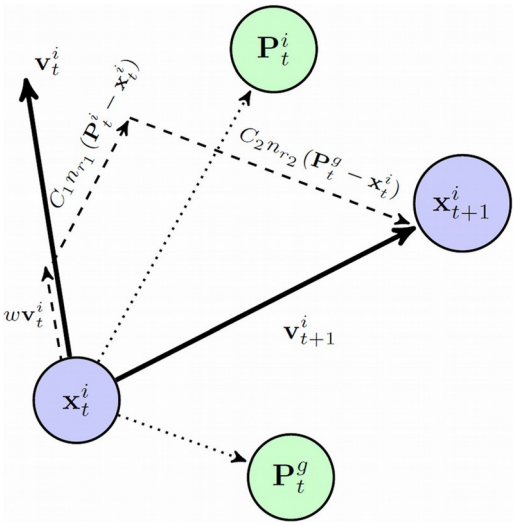
$$v_j^{(i)} \leftarrow w_1 v_j^{(i)} + c_1 r_1 \left(p_j^{(i)} - x_j^{(i)}\right) + c_2 r_2 \left(g^{(i)} - x_j^{(i)}\right)$$

Dónde:

$$x_j^{(i)} \leftarrow x_j^{(i)} + v_j^{(i)} \qquad w_1 = 1 - \frac{k}{L_1 + 1} \qquad r_1 = U(-1,1)$$

Siendo “k” el número entero actual de iteraciones que varía de 0 a L_1. Y siendo “r” un número aleatorio entre -1 y 1.

Volviendo a la fórmula del movimiento de cada partícula. La mejor solución encontrada por la partícula j-ésima corresponderá a “p_j”, “v_j” será la velocidad de dicha partícula y “g” la mejor solución global encontrada. Respecto a “c_i”, son los coeficientes de aceleración y tomarán un valor constante.



Como hemos mencionado anteriormente, GSO utilizará PSO tanto en exploración como en explotación, dos partes muy diferenciadas en nuestro algoritmo GSO puesto que este se encuentra dividido en dos niveles. Antes de comentar cada nivel repetiremos la idea comentada en el apartado de “Representación de datos”, pues recordemos que para el desarrollo de nuestro algoritmo

plantearemos un conjunto M de galaxias, formada cada una de ella por un conjunto de N estrellas con valores inicializados de manera aleatoria, siendo cada estrella una posible solución para nuestro problema.

Una vez conocemos los datos de partida podemos comentar los dos niveles que desarrollaremos. El primer nivel corresponde con la exploración, y en el las estrellas se moverán dentro de cada galaxia de la misma forma que en PSO, donde el movimiento ($x_j^{(i)}$) vendrá dado por la velocidad actual , mejor posición encontrada y la mejor solución obtenida por la galaxia en la que nos encontramos.

El segundo nivel corresponde con la explotación, y en el se tomarán las mejores soluciones de cada galaxia y se creará una galaxia con dichas mejores soluciones. Se empelará de nuevo el algoritmo PSO para la búsqueda en esta nueva población de soluciones, esta vez el movimiento vendrá dado por la velocidad actual, el mejor personal de cada solución y la mejor solución global actual.

4.2. Algoritmo

Presentamos a continuación el pseudocodigo general del algoritmo para mayor entendimiento, en el cual es fácil diferenciar los dos niveles comentados anteriormente.

Algorithm 1: GSO(f)

Level 1 Initialization: $\mathbf{x}_j^{(i)}, \mathbf{v}_j^{(i)}, \mathbf{p}_j^{(i)}, \mathbf{g}^{(i)}$ within $[x_{min}, x_{max}]^D$ randomly.
Level 2 Initialization: $\mathbf{v}^{(i)}, \mathbf{p}^{(i)}, \mathbf{g}$ within $[x_{min}, x_{max}]^D$ randomly.
for $EP \leftarrow 1$ **to** EP_{max}
do
- Begin PSO: Level 1
- **for** $i \leftarrow 1$ **to** M
  - **do**
    - **for** $k \leftarrow 0$ **to** L_1
      - **do**
        - **for** $j \leftarrow 1$ **to** N
          - **do**
            - $\mathbf{v}_j^{(i)} \leftarrow \omega_1 \mathbf{v}_j^{(i)} + c_1 r_1 (\mathbf{p}_j^{(i)} - \mathbf{x}_j^{(i)}) + c_2 r_2 (\mathbf{g}^{(i)} - \mathbf{x}_j^{(i)})$;
            - $\mathbf{x}_j^{(i)} \leftarrow \mathbf{x}_j^{(i)} + \mathbf{v}_j^{(i)}$;
            - **if** $f(\mathbf{x}_j^{(i)}) < f(\mathbf{p}_j^{(i)})$
              - **then**
                - $\mathbf{p}_j^{(i)} \leftarrow \mathbf{x}_j^{(i)}$;
                - **if** $f(\mathbf{p}_j^{(i)}) < f(\mathbf{g}^{(i)})$
                  - **then**
                    - $\mathbf{g}^{(i)} \leftarrow \mathbf{p}_j^{(i)}$;
                    - **if** $f(\mathbf{g}^{(i)}) < f(\mathbf{g})$
                      - **then** $\mathbf{g} \leftarrow \mathbf{g}^{(i)}$;
- Begin PSO: Level 2
- Initialize Swarm $\mathbf{y}^{(i)} = \mathbf{g}^{(i)} : i = 1, 2, \cdots, M$;
- **for** $k \leftarrow 0$ **to** L_2
  - **do**
    - **for** $i \leftarrow 1$ **to** M
      - **do**
        - $\mathbf{v}^{(i)} \leftarrow \omega_2 \mathbf{v}^{(i)} + c_3 r_3 (\mathbf{p}^{(i)} - \mathbf{y}^{(i)}) + c_4 r_4 (\mathbf{g} - \mathbf{y}^{(i)})$;
        - $\mathbf{y}^{(i)} \leftarrow \mathbf{y}^{(i)} + \mathbf{v}^{(i)}$;
        - **if** $f(\mathbf{y}^{(i)}) < f(\mathbf{p}^{(i)})$
          - **then**
            - $\mathbf{p}^{(i)} \leftarrow \mathbf{y}^{(i)}$;
            - **if** $f(\mathbf{p}^{(i)}) < f(\mathbf{g})$
              - **then** $\mathbf{g} \leftarrow \mathbf{p}^{(i)}$;

Return $\mathbf{g}, f(\mathbf{g})$

4.3. Implementación

Presentamos a continuación el pseudocódigo implementado para la adaptación de la metaheuristica para mi problema de agrupamiento con restricciones (PAR).

Primero comentar la importancia de los diferentes valores que le establecemos a las variables del algoritmo. Estas determinaran unos mejores o peores resultados y dependerán del tamaño del conjunto de entrada del problema (X), lo cual nos origina un problema, puesto que nos encontramos con unos conjuntos de entrada con un gran tamaño (sobre todo en el caso de Ecoli), lo que equivale a que si queremos ajustar nuestras variables para obtener una solución cercana a la óptima tendremos que aumentar mucho nuestros valores, lo que incrementará bastante el tiempo de ejecución del algoritmo. Tras varias pruebas he decidido establecer las siguientes variables al problema, las que he considerado más adecuadas tanto en calidad de la solución como en el tiempo de duración de la ejecución.

Se ha establecido M = 20 , siendo M el total de galaxias. N = 10 , siendo N el total de estrellas. L1 = 450, L2 = 4500, EP_max = 10, siendo EP_max el total de épocas. Y los coeficientes de aceleración a un valor de 2.5.

Tras ello presentamos el pseudocódigo:

```
FUNCIÓN algoritmo_GSO()
    //Comenzamos generando las poblaciones aleatorias de cada población y subpoblación
    //Para el nivel 1 inicializo aleatoriamente  x_j^(i)  ,  v_j^(i)  ,  p_j^(i)  y  g^(i)
    PARA cada galaxia hasta las M galaxias totales
        PARA cada estrella hasta las N estrellas totales
            inicioAleatorio(x[galaxia][estrella])
            inicioAleatorio(v1[galaxia][estrella])
            inicioAleatorio(p1[galaxia][estrella])
            valoracion_p1[galaxia][estrella] = funcionObjetivo(p1[galaxia][estrella])
        FIN-PARA
    FIN-PARA
    //Para el nivel 2 inicializo aleatoriamente  v^(i)  ,  p^(i)  y  g
    PARA cada galaxia hasta las M galaxias totales
        inicioAleatorio(g[galaxia])
        inicioAleatorio(v2[galaxia])
        inicioAleatorio(p2[galaxia])
        valoracion_p2[galaxia] =  funcionObjetivo(p2[galaxia])
        valoracion_g[galaxia] =  funcionObjetivo(g[galaxia])
    FIN-PARA
    inicioAleatorio(mejor_g)
    mejor_g_valoracion =  funcionObjetivo(mejor_g)
```

```
PARA cada epoca hasta EP_max incrementa el valor de época en una unidad
    //Comenzamos el nivel 1
    PARA cada galaxia hasta M galaxias totales
        PARA cada k siendo k = 0 hasta k <= L1 incremento k en una unidad
```

$$w_1 = 1 - \frac{k}{L_1 + 1}$$

```
            PARA cada estrella hasta N estrellas totales
                r1 = valor real aleatorio entre -1 y 1
                r2 = valor real aleatorio entre -1 y 1
                PARA cada d siendo d = 0 hasta d < datos.size() incremento d
                    v[i][j][d] = w1v[i][j][d] + c1r1(p[i][j][d] - x[i][j][d]) +
                                                              c2r2(g[i][d] - x[i][j][d])
                    x[i][j][d] = x[i][j][d] +  v[i][j][d]
                FIN-PARA
                //Función que evita que x tome valores de cluster que no son válidos (será más
                //detallada al tras este algoritmo
                x[i][j] = comprobarCluster(x[i][j])
                nueva_valoracion = funcionObjetivo(x[i][j])
                //Comienzo el proceso de comprobación si he obtenido mejores valoraciones
                SI nueva_valoracion < valoracion_p1[i][j]
                    PARA cada d siendo d = 0 hasta d < datos.size() incremento d
                        p1[i][j][d] = x[i][j][d]
                    FIN-PARA
                    valoracion_p1[i][j] = nueva_valoracion
                    SI nueva_valoracion < valoracion_g[i]
                        PARA cada d siendo d = 0 hasta d < datos.size() incremento d
                            g[i][d] = p1[i][j][d]
                        FIN-PARA
                        valoracion_g[i] = nueva_valoracion
                        SI nueva_valoracion < mejor_g_valoracion
                            PARA cada d siendo d=0 hasta d < datos.size() incremento d
                                mejor_g[d] = g[i][d]
                            FIN-PARA
                            mejor_g_valoracion = nueva_valoracion
                        FIN-SI
                    FIN-SI
                FIN-SI
            FIN-PARA
        FIN-PARA
    FIN-PARA

    //Hemos finalizado el primer nivel, seguimos en el bucle de las épocas y comienza el
    //segundo nivel, la explotación
```

```
        //Comenzamos el nivel 2
        Creo el vector de vectores de tipo double "y" con tamaño "g"
        PARA cada k siendo k = 0 hasta k <= L2 incremento k en una unidad
            w2 = 1 - k / (L2 + 1)
            PARA cada galaxia hasta M galaxias totales
                r3 = valor real aleatorio entre -1 y 1
                r4 = valor real aleatorio entre -1 y 1
                PARA cada d siendo d = 0 hasta d < datos.size() incremento d
                    v2[i][d] = w2v2[i][d] + c3r3(p2[i][d] - y[i][d]) + c4r4(mejor_g[d] - y[i][d])
                    y[i][d] = y[i][d] +  v2[i][d]
                FIN-PARA
                //Función que evita que x tome valores de cluster que no son válidos (será más
                //detallada al tras este algoritmo
                y[i] = comprobarCluster(y[i])
                    nueva_valoracion = funcionObjetivo(y[i])
                    //Comienzo el proceso de comprobación si he obtenido mejores valoraciones
                    SI nueva_valoracion < valoracion_p2[i]
                        PARA cada d siendo d = 0 hasta d < datos.size() incremento d
                            p2[i][d] = y[i][d]
                        FIN-PARA
                        valoracion_p2[i] = nueva_valoracion
                        SI nueva_valoracion < mejor_g_valoracion
                            PARA cada d siendo d=0 hasta d < datos.size() incremento d
                                mejor_g[d] = p2[i][d]
                            FIN-PARA
                            mejor_g_valoracion = nueva_valoracion
                        FIN-SI
                    FIN-SI
                FIN-PARA
            FIN-PARA
        FIN-PARA
    FIN-PARA
    DEVOLVER mejor_g
FIN
```

La función "comprobarCluster()" utilizada en ambos niveles se encarga de asegurar que se le ha asignado un cluster valido y existente a cada elemento. Para ello, si se le ha asignado un movimiento (x) superior al número total de cluster - 1, le asignaremos a ese elemento el valor del número total de cluster - 1 , es decir, si tenemos 3 cluster y un elemento toma el valor 5, cambiaremos ese valor por 2 (ya que 3 cluster equivale a cluster 0, 1 y 2).
Si estamos ante el caso de que se le ha asignado un cluster negativo, se sustituirá ese valor por 0.
Además en esta función tendremos en cuenta que ningún cluster queda vacio con la función "evitarClusterVacio()" ya comentada.

4.4. GSO con BL

Recordemos que nuestra BL está basada en el primer mejor. La idea de este algoritmo era 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 GSO, para ello haremos un cambio en la forma de explotación de nuestro algoritmo. Esta dejará de hacerse mediante PSO y en su lugar utilizaremos nuestro algoritmo de BL con una condición de parada de 7000 evaluaciones 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 en nuestro algoritmo GSO ya presentado es muy sencilla. Simplemente tendremos que dirigirnos al nivel 2 que corresponde con la explotación, se mantedrá la idea de tomar las mejores soluciones de cada galaxia, pero en lugar de ejecutar el algoritmo de PSO se ejecutará la BL sobre cada una de dichas soluciones.

El pseudocódigo es el siguiente:

```
    //Muestro solamente el nivel 2, pues es donde se produce el único cambio
    Creo el vector de vectores de tipo double “y” con tamaño “g”
    PARA cada galaxia hasta M galaxias totales
        y[i] = busquedaLocal(y[i])
        PARA cada d siendo d = 0 hasta d < datos.size() incremento d
            p2[i][d] = y[i][d]
        FIN-PARA
        valoracion_p2[i] = funcionObjetivo(y[i])
        SI valoracion_p2[i] < mejor_g_valoracion
            PARA cada d siendo d=0 hasta d < datos.size() incremento d
                mejor_g[d] = p2[i][d]
            FIN-PARA
            mejor_g_valoracion = valoracion_p2[i]
        FIN-SI
    FIN-PARA
    DEVOLVER mejor_g
FIN
```

4.4. GSO con operador de Mutación

Como forma de mejorar mi algoritmo he decidido cambiar la forma de exploración, es decir, he decidido sustituir la exploración con PSO por un operador de mutación que fue implementado en la práctica anterior para la realización del algoritmo ILS.

Es evidente lo costoso en tiempo que es nuestro algoritmo GSO para conjunto de datos de entrada de gran tamaño, por ello debemos hacer lo posible para reducir los tiempos de exploración y explotación. En el caso de explotación ya lo he conseguido gracias a la BL, cambio que mantendré en este nuevo algoritmo, ahora me falta reducir el coste de la explotación, lo cual consigo con este cambio.

Para la mutación hemos utilizado un operador de mutación por segmento, en este, 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 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 presentado el algoritmo del operador mutación solamente nos falta ver su implementación. Como ya hemos mencionada esta se utilizará para la exploración, por ello el único cambio se realizará en el nivel 1.

```
PARA cada epoca hasta EP_max incrementa el valor de época en una unidad
        //Comenzamos el nivel 1
        PARA cada galaxia hasta M galaxias totales
            PARA cada estrella hasta N estrellas totales
                x[i][j] = mutarBrusco(x[i][j]);
                nueva_valoracion = funcionObjetivo(x[i][j])
                //Comienzo el proceso de comprobación si he obtenido mejores valoraciones
                SI nueva_valoracion < valoracion_p1[i][j]
                    PARA cada d siendo d = 0 hasta d < datos.size() incremento d
                        p1[i][j][d] = x[i][j][d]
                    FIN-PARA
                    valoracion_p1[i][j] = nueva_valoracion
                    SI nueva_valoracion < valoracion_g[i]
                        PARA cada d siendo d = 0 hasta d < datos.size() incremento d
                            g[i][d] = p1[i][j][d]
                        FIN-PARA
                        valoracion_g[i] = nueva_valoracion
                        SI nueva_valoracion < mejor_g_valoracion
                            PARA cada d siendo d=0 hasta d < datos.size() incremento d
                                mejor_g[d] = g[i][d]
                            FIN-PARA
                            mejor_g_valoracion = nueva_valoracion
                        FIN-SI
                    FIN-SI
                FIN-SI
            FIN-PARA
        FIN-PARA
```

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”

GSO.cpp/.h: en este archivo están definidos los métodos necesarios para ejecutar el algoritmo de Galacti Swarm Optimization (GSO), el algoritmo GSO-BL 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 GSO
    GSO gso(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
GSO	0,16	198,80	1,27	83,15	5,01	1207,40	35,58	345,86	0,2	150,80	1,16	83,58	1,36	95,00	4,54	143,21
GSO-BL	0,11	0,00	0,11	58,75	5,19	118,40	8,18	428,58	0,12	0,00	0,12	56,45	1,85	58,20	3,79	83,32
GSO-Mutacion	0,11	0,00	0,11	17,77	7,24	148,40	11,00	260,25	0,12	0,00	0,12	17,18	2,65	0,00	2,65	52,51

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
GSO	0,12	341,60	1,13	106,68	4,49	2226,00	33,48	460,63	0,21	372,40	1,47	109,67	1,36	223,00	5,26	191,26
GSO-BL	0,11	0,00	0,11	72,33	5,29	159,80	7,37	576,1	0,12	0,00	0,12	72,14	1,36	223,00	5,26	111,4
GSO-Mutacion	0,11	0,00	0,11	19,51	5,98	206,60	8,68	344,97	0,12	0,00	0,12	19,91	2,65	0,00	2,65	58,27

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.

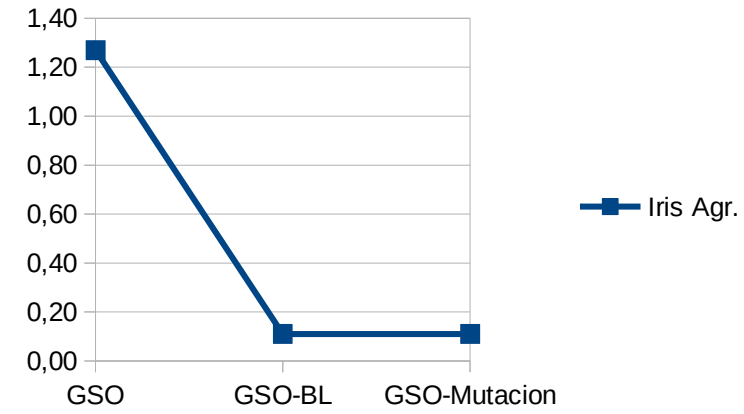
He presentado los resultados de la media de los resultados tras 5 ejecucciones de cada algoritmo (con 5 semillas diferentes) sobre cada conjunto de datos. En este caso me centraré en comparar los resultados de la BL y Greedy frente a los resultados con GSO y sus variantes.

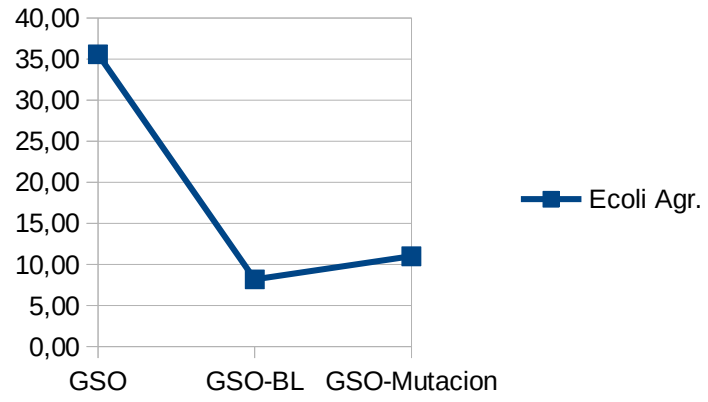
Presento una tabla que recoja estos datos:

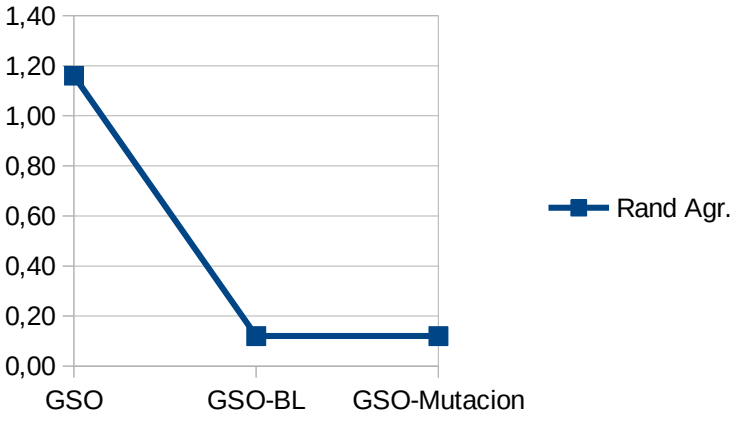
	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
COPKM	0,11	380,00	0,18	0,54	2,24	2721,00	6,86	9,17	0,1	574,80	0,2	0,54	1,29	1001,40	2,58	3,22
BL	0,11	0,00	0,11	0,10	4,98	56,00	6,40	4,85	0,12	0,00	0,12	0,09	2,54	17,40	3,12	0,29
GSO	0,16	198,80	1,27	83,15	5,01	1207,40	35,58	345,86	0,2	150,80	1,16	83,58	1,36	95,00	4,54	143,21
GSO-BL	0,11	0,00	0,11	58,75	5,19	118,40	8,18	428,58	0,12	0,00	0,12	56,45	1,85	58,20	3,79	83,32
GSO-Mutacion	0,11	0,00	0,11	17,77	7,24	148,40	11,00	260,25	0,12	0,00	0,12	17,18	2,65	0,00	2,65	52,51

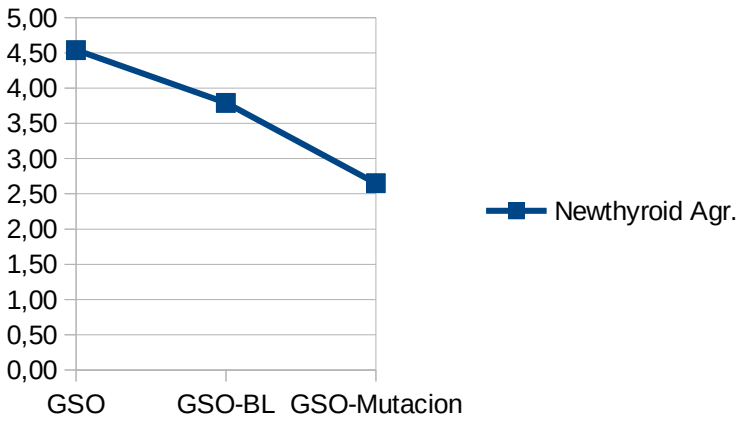
	Iris				Ecoli				Rand				Newthyroid			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
COPKM	0,1	750,40	0,15	1,37	3,06	5008,80	11,94	21,13	0,1	1234,20	0,22	0,74	1,29	2031,60	2,66	5,16
BL	0,11	4,80	0,12	0,12	4,43	47,60	5,05	5,48	0,12	0,00	0,12	0,13	2,65	0,00	2,65	0,38
GSO	0,12	341,60	1,13	106,68	4,49	2226,00	33,48	460,63	0,21	372,40	1,47	109,67	1,36	223,00	5,26	191,26
GSO-BL	0,11	0,00	0,11	72,33	5,29	159,80	7,37	576,1	0,12	0,00	0,12	72,14	1,36	223,00	5,26	111,4
GSO-Mutacion	0,11	0,00	0,11	19,51	5,98	206,60	8,68	344,97	0,12	0,00	0,12	19,91	2,65	0,00	2,65	58,27

Comentemos primero las diferencias entre los tres tipos de algoritmo GSO desarrollados, para ello presentamos gráficas con las agregaciones medias resultantes por cada uno de ellos en cada conjunto, con un 10% de restricciones:



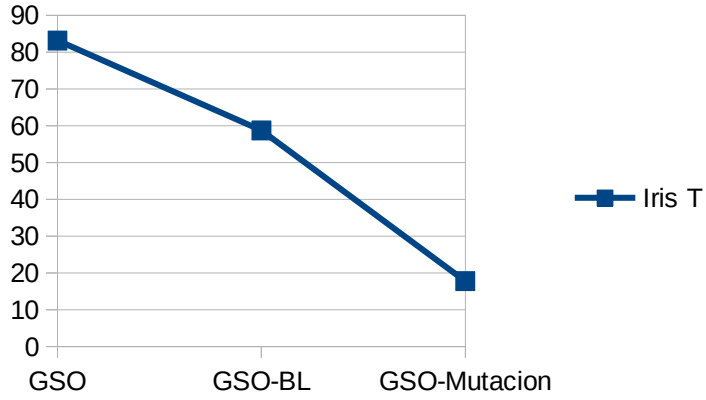


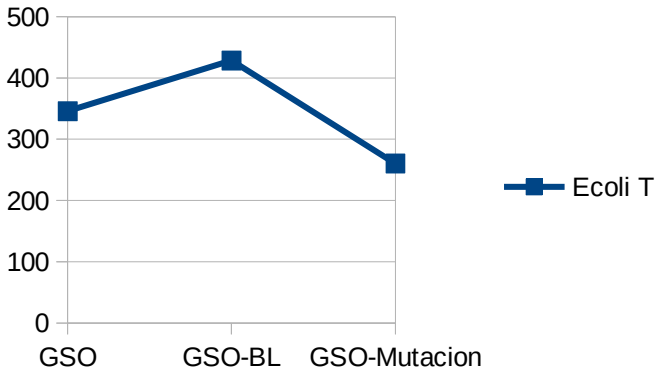


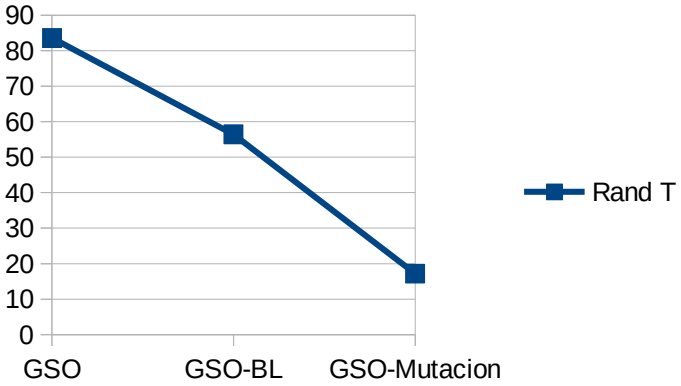


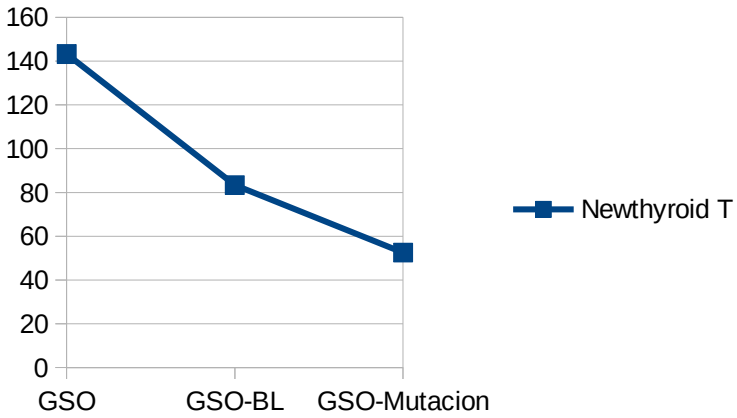
Podemos observar como nuestras modificaciones sobre el planteamiento inicial de GSO han mejorado el resultado final. En el caso del conjunto Iris y Rand hemos obtenido el óptimo tanto con el caso de añadir solamente la búsqueda local, GSO-BL, cómo en el caso de además añadir el operador de mutación, GSO-Mutación. En el caso de Ecoli es el único donde los resultados empeoran al añadir la mutación respecto al utilizar PSO para la exploración, aunque no es una gran diferencia. Sin embargo en Rand ocurre lo contrario, los resultados mejoran al cambiar la forma de exploración, por lo que podemos intuir que la diferencia se debe al tamaño del conjunto de datos de entrada, actuando en este caso PSO mejor con un conjunto de mayor tamaño que nuestro operador de mutación.

Observemos de la misma forma los tiempos obtenidos:









Fácilmente observamos como los tiempos mejoran por cada modificación nueva que le añadimos a nuestro planteamiento inicial, menos en el caso de Ecoli con GSO-BL, tiempo que aumentan por la gran cantidad de tiempo que consume la exploración, que en GSO-Mutacion mejoraremos.

Una vez presentadas las diferencias entre estos tres algoritmos, procedemos a compararlos con los resultados obtenidos con la BL y Greedy.

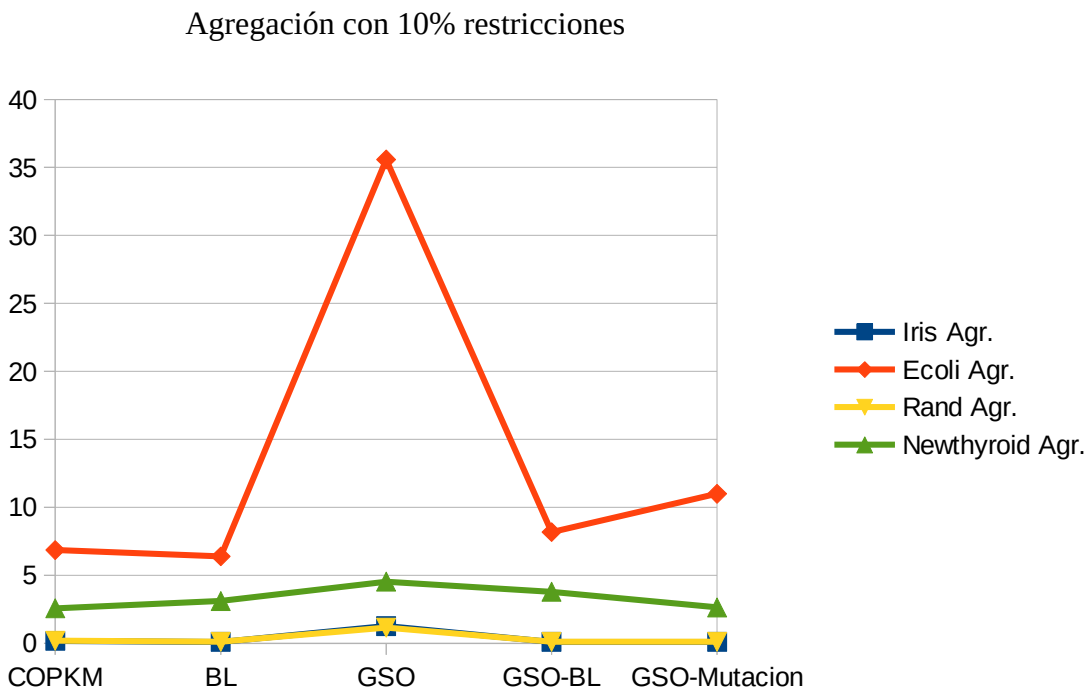
Dado que debemos analizar una gran cantidad de algoritmos y datos, vamos a estructurar el análisis por bloques, en los que se compararán y explicarán distintos aspectos de los experimentos realizados:

1. Análisis del Agregado

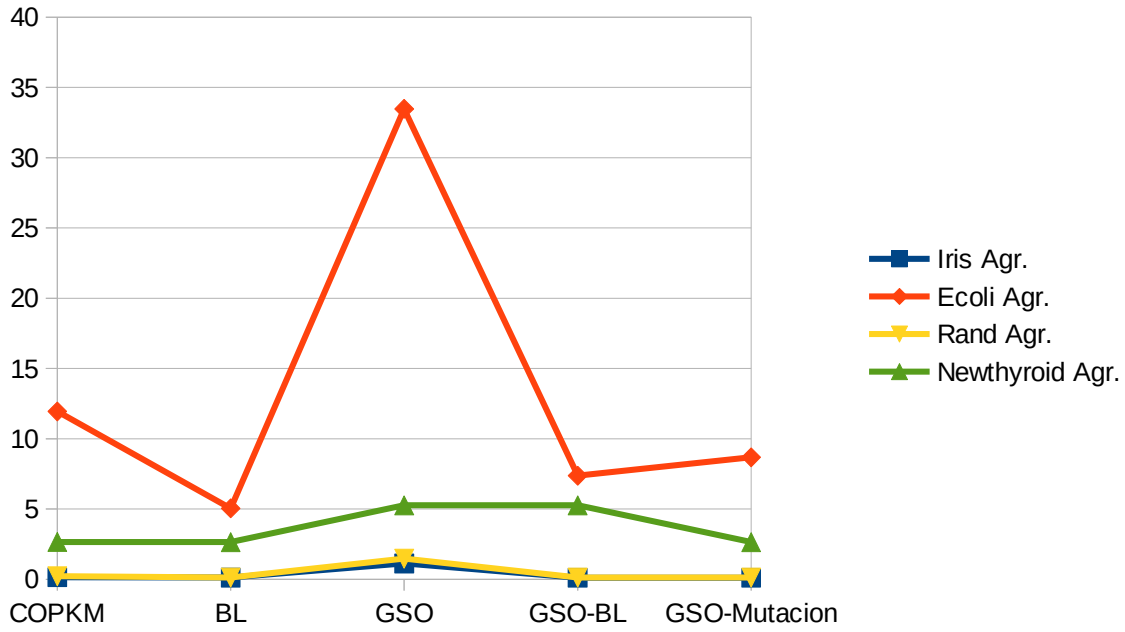
Recordamos que el Agregado corresponde al valor obtenido por la función objetivo definida, f = C + (infeasability * λ). Siendo C la desviación general y λ el cociente entre la distancia máxima existente en el conjunto de datos y el número de restricciones presentes en el problema.

De nuevo la mayor diferencia la notaremos observando el conjunto de datos Ecoli, al igual que en las evaluaciones anteriores. Aún así en todos los resultados de cada conjunto podemos observar un pico en el caso de GSO, siendo el que peor resultados obtiene, esto creo que se debe a que este algoritmo es bueno para encontrar zonas prometedoras, pero es demasiado influenciado por las mejores soluciones obtenidas, lo cual provoca una convergencia demasiado rápida en óptimos locales. Frente a este defecto podemos observar que al convinar esta capacidad de encontrar zonas prometedoras con otros algoritmos se puede llegar a dar unos resultados realmente buenos, lo que podemos comprobar con GSO-BL, y aún con más claridad con GSO-Mutación.

Exceptuando el caso de Rand y Ecoli, las versiones modificada de GSO llegan a alcanzar un óptimo al igual que BL, y en el caso de Ecoli llegan a obtener unos valores bastante bajos cercanos al obtenido con BL. En el caso de Greedy se genera una menor Tasa_C que respecto a los algoritmos GSO, pero frente a esto, se generan unos valores de Tasa_inf muy elevados en Greedy que son muy reducido en GSO y sus variantes.



Agregación con 20% restricciones

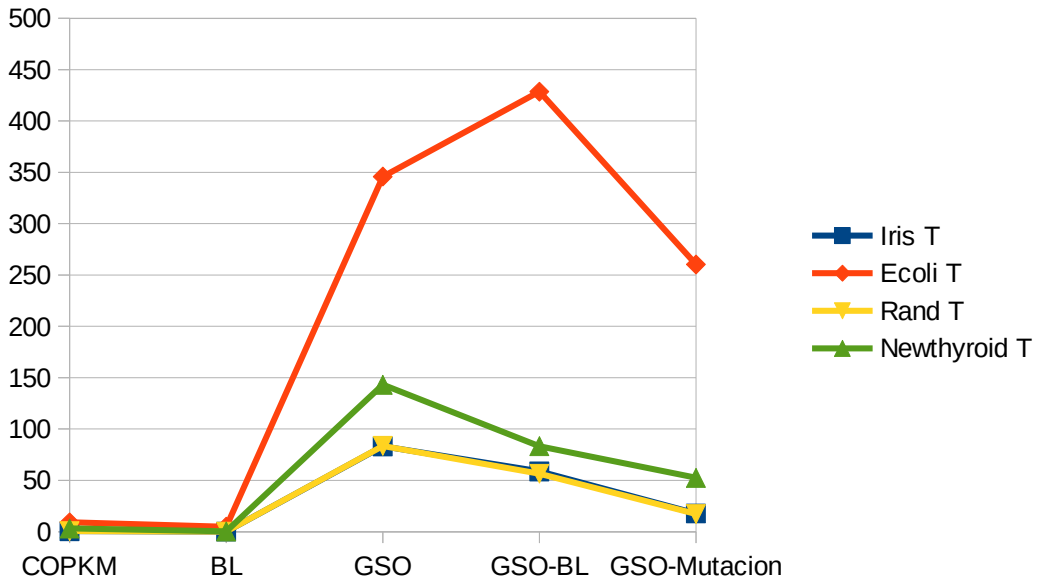


2. Análisis del Tiempo

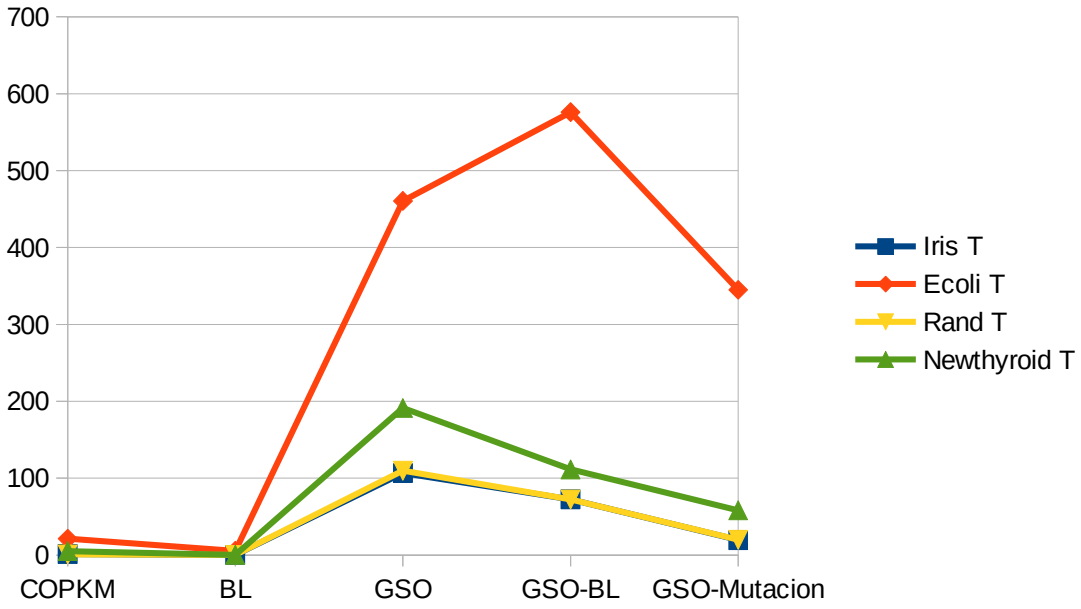
La diferencia en tiempo con respecto a los algoritmos BL y Greedy es muy notable, siendo en todos los casos de GSO un tiempo de ejecución medio mucho mayor que en los dos anteriores. Esta diferencia de tiempos sólo se reduce en el caso de GSO-Mutación, menos en el caso de Ecoli, siendo los tiempos de esta versión de GSO los mejores, pero aún así en el mejor de los casos manteniendo una diferencia de aproximadamente 50 segundos.

Es evidente que aunque los resultados con GSO-Mutación lleguen a ser muy parecidos a los obtenidos con la BL, al observar los tiempos el mejor algoritmo con diferencia sigue siendo la BL.

Tiempo con 10% restricciones



Tiempo con 20% restricciones



8.Bibliografía

Toda la información sobre el GSO ha sido obtenida de https://www.sciencedirect.com/science/article/pii/S1568494615006742#eq0050

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