METAHEURÍSTICAS - Práctica 3

Búsquedas por Trayectorias para el Problema del Agrupamiento con Restricciones

Problema de Agrupamiento con Restricciones

Pedro Gallego López
48261534J
pedrogallego@correo.ugr.es
Grupo 3: Lunes 17:30 - 19:30

Curso 2020/2021. Universidad de Granada
Doble Grado de Ingeniería Informática y Matemáticas
Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

Índice

Anotaciones previas

En esta primera sección se hablará de la estructura del documento, los cambios producidos en partes de prácticas anteriores y el acoplamiento de la nueva práctica a esta memoria.

En primer lugar se ha reciclado la memoria al completo de la práctica anterior, sobre ella se realizarán actualizaciones (las cuales se nombrarán en este apartado) y se pondrán los apartados de algoritmos y de análisis de la Práctica 2 a continuación del algoritmo greedy y Búsqueda Local.

Actualizaciones

En el manual de usuario se indica ahora un método sencillo para ejecutar según que algoritmos, en función de la práctica que se quiera ejecutar: hay una variable booleana para greedy, una para BL, una para genéticos, una para meméticos y una para las trayectorias, y de ella dependerá la ejecución o no de estos algoritmos.

Nuevos Apartados de la práctica 3

A continuación de las metaheurísticas Genéticas y Meméticas se han puesto las secciones relativas a las metaheurísticas de Búsqueda por Trayectorias. Mismo formato.

En la parte de **Experimentos y Análisis de resultados**, se ha añadido a continuación de los Genéticos y Meméticos, el análisis de las metaheurísticas de Búsqueda por Trayectorias. Aquí se analizarán los 4 métodos entre ellos. Al final de esta sección hay un análisis extra acerca de los parámetros de Enfriamiento Simulado.

Por último, en la sección **Análisis conjunto de todos los algoritmos** se han añadido comentarios a la comparación conjunto y además se ha creado una subsección específica para comparar los algoritmos de Búsqueda por Trayectorias con el algoritmo Greedy y el de BL.

1. Descripción del problema PAR.

El **problema del agrupamiento por restricciones** persigue la clasificación de objetos de acuerdo a posibles similitudes entre ellos, es decir, agruparlos según esas características en distintos **clústers** (conjunto de instancias del problema). A este agrupamiento se le añade un conjunto de restricciones que tienen que respetarse según su naturaleza (**fuertes** ó **débiles**).

Así el problema consiste en que dado un conjunto de datos X de $n \times d$ valores reales. Donde n es el número de instancias en un espacio de d dimensiones notadas como:

$$\overrightarrow{x_i} = \{x_{[i,1]}, ..., x_{[i,d]}\}, \qquad x_{[i,j]} \in \mathbb{R}, \quad \forall j \in \{1, ..., d\}$$

Encontrar una partición C de tamaño k del mismo que minimice la desviación general y cumpla extrictamente con las restricciones fuertes del conjunto de restricciones R e intente cumplir tantas restricciones débiles de R como sea posible.

Las restricciones se pueden clasificar como restricciones **fuertes** y **débiles**.

Fuertes (Hard)

Todas las restricciones deben satisfacerse en la partición C del conjunto de datos X. Se dice que una partición es factible si y solo si cumple con todas las restricciones en R. Restricciones:

- Todos los clusters deben contener al menos una instancia
- Cada instancia debe pertenecer a un solo cluster
- La unión de los clusters debe ser el conjunto de datos X

Débiles (Soft)

La partición C del conjunto de datos X debe minimizar el número de restricciones incumplidas pero puede incumplir algunas. Las restricciones de este tipo nos dirán si dos instancias **deben ir juntas, Must-Link (ML)** o **deben ir separadas, Cannot-Link (CL)** en el agrupamiento.

Medición de bondad de la solución

Siendo $\mu_i = \frac{1}{|c_i|} \sum_{\overrightarrow{x_j} \in c_i} \overrightarrow{x_j}$ el centroide del cluster i, $\bar{c_i} = \frac{1}{|c_i|} \sum_{\overrightarrow{x_j} \in c_i} ||\overrightarrow{x_j} - \mu_i||_2$ su distancia intra-clúster, se define $\bar{C} = \frac{1}{k} \sum_{c_i \in C} \bar{c_i}$ como la desviación general de la partición. Además definimos $infeasibility$ como el número de restricciones débiles incumplidas. Nuestra función objetivo que querremos minimizar será pues

$$fitness\_function(C) = \bar{C} + \lambda infeasibility(C), \qquad \lambda \in \mathbb{R}^+$$

Conjuntos de datos

- Zoo: agrupar animales en función de 16 atributos. Tiene 7 clases (k=7) y 101 instancias.
- Glass: agrupar distintos vidrios en función de 9 atributos sobre sus componentes químicos. Tiene 7 clases (k=7) y 214 instancias.
- Bupa: agrupar personas en función de sus hábitos de consumo de alcohol, definido con 5 atributos. Tiene 16 clases (k=16) y 345 instancias.

A partir de cada conjunto de datos se obtienen 2 instancias distintas generando 2 conjuntos de restricciones, correspondientes al 10 % y 20 % del total de restricciones posibles. Los llamaremos CS_10 y CS_20, respectivamente.

2. Consideraciones comunes

En este apartado hablaremos de la representación del problema que se ha usado. Se hablará de temas como representación de los datos, como de las funciones que nos indicarán la bondad de la solución y de todas aquellas cuestiones que sean independientes del algoritmo utilizado.

2.1. Representación de los datos

En primer lugar los datos vienen dados en ficheros. Los ficheros de datos, es decir, de las instancias vienen Los ficheros de datos contienen en cada fila una instancia del conjunto en cuestión con sus características separadas por comas (sin espacios). Los ficheros de restricciones almacenan una matriz de restricciones con los valores separados por comas y sin espacios donde un 1 en la posición (i, j) representa una restricción $Must - Link$, un -1 una $Cannot - Link$ y un 0 en caso de no haber restricción. En la diagonal principal tendremos un 1 en todas las posiciones. Sin embargo, esta diagonal la daremos por triviales y solo miraremos por encima de la diagonal para averiguar las restricciones totales. Al leer de los ficheros, almacenamos los datos en matrices: de `double` para los datos y `int` para las restricciones.

Para representar el problema dentro del código se ha definido una clase con todos los atributos necesarios para definir el problema de forma que contiene todos los métodos necesarios para tratar los datos y llevar a cabo cualquier manipulación sobre ellos. La clase se ha llamado **PAR**. Entre los atributos más importantes tenemos:

- **Instancias del problema**: matriz $n \times d$ de tipo `double` donde n es el número de instancias y d el número de características de las mismas.
- **Clústers y soluciones**: en la representación de los clústers se le ha quitado importancia a la memoria para darle importancia al acceso rápido de los datos y se han llevado a cabo dos estructuras distintas para ello:
  - **Clúster**: representación más intuitiva de un clúster. Es un vector de tamaño k el número de clases, que en cada posición contiene un vector dinámico de enteros, donde cada valor hace referencia a una instancia. Esto es, si en la posición j del vector *i-ésimo* hay un elemento n, esto quiere decir que en el **clúster *i-ésimo* está la instancia *n-ésima***. Hay que notar que la posición j no influye, es decir, el orden de las instancias dentro del clúster no van a tener ninguna importancia.
  - **Instancia_pertenece_a**: vector de tamaño n número de instancias, donde en cada posición puede haber un valor de $\{1, 2, ..., k\}$ que hace referencia al clúster que pertenece. Esto es, si en la posición i está el valor c esto quiere decir que la **instancia *i-ésima* pertenece al clúster *c-ésimo*.** Habrá otro atributo **mejor_solucion** que tendrá el valor del **Instancia_pertenece_a** con mejor fitness.
- **Centroides**: matriz de tamaño $k \times d$, k clases y d características. Se actualizarán tras la llamada a un método específico para ello.
- **Población**: representación de cromosomas(soluciones) para algoritmos genéticos y meméticos. Se usará un vector de tamaño $TAM\_POBLACION$ de *soluciones completas*. Estas *soluciones completas* son vectores de tamaño $n + 2$ donde n es el número de instancias: las n primeras componentes hacen referencia a una solución con la misma representación que"*Instancia_pertenece_a*", la penúltima componente hace referencia a la época generacional en la que se creó esta *solución completa* y su última componente guarda el valor de la función objetivo que se tiene con esa solución.

- **Restricciones**: para representar las restricciones se han llevado a cabo dos representaciones siguiendo la misma filosofía que con la representación del clúster; sacrificar memoria para ganar velocidad. Las representaciones son:
  - **Matriz**: matriz de tamaño $n \times n$ donde en la posición (i, j) tiene el mismo valor que el fichero de restricciones en la posición (i, j) anteriormente comentado.
  - **Vectores ML y CL**: a partir de la estructura anterior, se han creado dos vectores nuevos de tamaño $r_{ML} \times 2$ y $r_{CL} \times 2$. en las posiciones (i, j) se encuentran instancias. En el vector ML (CL) tenemos que las dos instancias que pertenecen a una misma fila estarán restringidas a una restricción $Must - Link$ ($Cannot - Link$).

Además de estos hay otros atributos que se usan de forma auxiliar para dar velocidad al proceso de cálculo como el almacenamiento de la distancia más grande, o un vector de índices para recorrer de forma aleatoria las instancias.

2.2. Recorrido con aleatoriedad

Para dar aleatoriedad a los algoritmos y hacer así que no sean deterministas se recorre, en cada punto de la ejecución en donde haya que tomar una decisión entre varias, las distintas opciones posibles en orden aleatorio. Esto se ha implementado creando un vector de índices que contiene los elementos $\{1, 2, ..., n-1\}$ donde n es el número de opciones posibles para elegir en el punto de la ejecución en el que estamos.

```
1: ARGS:
2:
3: vector v ={0,1,...,n-1}
4: vector indices_shuffled ={}
5: while not v.empty do
6:     i = entero aleatorio en {0,...,size(v)-1}
7:     indices_shuffled.insertaElemento(v[i])
8:     v.eliminaElemento(i)
9: end while
10: return indices_shuffled
```

Función que desordena índices de un vector de tamaño n.

La elección aleatoria del número entero aleatorio se realiza con la función `Randint()` del archivo *random.h*. De la misma forma esta función al usar la aleatoriedad dada por el archivo *random.h*, se ha incorporado dentro del mismo.

2.3. Evaluación de bondad de los resultados

Distancia intra-clúster

Para esta función hay que tener en cuenta las estructuras de representación de datos vistas en el apartado **2.1**, en donde se hace uso del vector de clústers. Además, se hace uso de una función auxiliar: `distanciaEntreDosPuntos(p1,p2)` que te calcula la distancia euclídea entre dos puntos de misma dimensión.

La función hace la media de las distancias de cada punto del clúster al centroide del clúster.

```
1: ARGS: 1: entero cluster
2:
3: flotante distancia = 0
4: for instancia ∈ clusters[cluster] do
5:     distancia += distanciaEntreDosPuntos(instancia, centroides[cluster])
6: end for
7: distancia /= size(clusters[cluster])
8: return distancia
```

Función que nos devuelve la distancia intra-clúster.

Desviación de la partición

Calculamos la desviación de la partición $C = \{c_1, c_2, ..., c_k\}$ como la media de las distancias intra-clústers.

```
1: ARGS:
2:
3: flotante desviacion = 0
4: for c ∈ {0, ..., k − 1} do
5:     desviacion += distanciaIntraCluster(c)
6: end for
7: desviacion /= k
8: return desviacion
```

Función que nos devuelve la desviación de la partición actual de la clase.

Infeasibility

```
1: ARGS:
2:
3: entero infeasibility = 0
4: for i ∈ {0, ..., size(ML) − 1} do
5:     if inst_belong(ML[i,0])!=inst_belong(ML[i,1]) then
6:         infeasibility+=1
7:     end if
8: end for
9: for i ∈ {0, ..., size(CL) − 1} do
10:    if inst_belong(CL[i,0])==inst_belong(CL[i,1]) then
11:        infeasibility+=1
12:    end if
13: end for
14: return infeasibility
```

Función que nos devuelve el infeasibility actual de la clase.

La infeasibility de la partición C la calculamos como:

$$infeasibility(C) = incumplidas\_ML(C) + incumplidas\_CL(C)$$

donde:

$$incumplidas\_ML(C) = \sum_{i=0}^{|ML|-1} bool2int(inst\_belong(\overrightarrow{ML}[i,0]) \neq inst\_belong(\overrightarrow{ML}[i,1]))$$

$$incumplidas\_CL(C) = \sum_{i=0}^{|CL|-1} bool2int(inst\_belong(\overrightarrow{CL}[i,0]) = inst\_belong(\overrightarrow{CL}[i,1]))$$

Siendo $bool2int(expresion\_booleana)$ una función que te devuelve 1 si $expresion\_booleana$ es verdad y 0 en otro caso. Y siendo $inst\_belong$ una función que dada una instancia te devuelve el clúster al que pertenece.

Función objetivo o función fitness

Nuestra función objetivo, como se explicó al inicio de la memoria es:

$$fitness(C) = desviacionParticion(C) + \lambda * infeasibility(C)$$

donde $\lambda = [D]/|R|$, con $[D]$ la distancia mayor que existe entre dos puntos y $|R|$ el número de restricciones.

```
1: ARGS:
2:
3: flotante λ = mayorDistancia/(size(ML) + size(CL))
4: funcion_objetivo = desviacionParticion() + λ * infeasibility()
5: return funcion_objetivo
```

Función que nos devuelve el valor de la función objetivo en el estado actual de la clase

Hay que notar que el valor de $mayorDistancia$ se calcula nada más inicializar el problema y se guarda en una variable que podremos consultar cuando se quiera. De la misma forma $funcion\_objetivo$ es un atributo de la clase que guarda el valor actual de la función objetivo.

2.4. Funciones auxiliares

En este subapartado hablo de métodos que han servido para mantener la coherencia dentro del problema. Estos métodos son aquellos destinados a la actualización de clústers y centroides y la generación de restricciones. No me voy a detener a desarrollarlos en pseudocódigo puesto que la actualización de centroides se ha explicado teóricamente al inicio en la descripción del problema (apartado **1**) y la generación de restricciones consiste en recorrer la diagonal superior de la matriz de restricciones como se ha dicho al inicio y en función del valor (i, j) añadir el par (i, j) a ML si hay un 1 o a CL si hay un -1.

2.5. Medición de tiempos

Se ha utilizado la librería `chrono`. El esquema de a continuación marca cómo se miden los tiempos en milisegundos de cualquier algoritmo en C++:

```
1: funcion algoritmo(){
2:         auto begin = chrono::high_resolution _clock::now();
3:                 Ejecutamos el algoritmo
4:         auto end = chrono::high_resolution _clock::now();
5:         auto elapsed=chrono::duration_cast<std::chrono::milliseconds>(end-begin);
6:         cout << "El tiempo ha sido" << elapsed << "ms" << endl;
7: }
```

Ejemplo de medición de tiempo de un algoritmo

3. Algoritmo de Búsqueda Local del primer mejor

En este apartado se describirá el algoritmo de Búsqueda Local empleado. Este algoritmo coge la filosofía del primer mejor. Esto es, coge el primer vecino que se encuentra que mejora la solución. Muchas veces esta estrategia ayuda a huir de óptimos locales y puede llegar a muy buenos resultados. La descripción básica del algoritmo en pasos es la siguiente:

ALGORITMO DE BÚSQUEDA LOCAL DESCRIPCIÓN CONCEPTUAL

1. Se inicializan las instancias aleatoriamente.
2. Se genera el vecindario.
3. Se recorre el vecindario de forma aleatoria hasta encontrar un vecino mejor.
4. Si se encuentra un vecino mejor lo cogemos como solución actual y volvemos al paso 2.
5. No se ha encontrado ningún vecino mejor luego nuestra solución es la actual.

Un vecino es mejor solución que la actual cuando su función objetivo es menor que la actual. Como criterio de parada, por lo tanto, tendremos que la solución no cambie, es decir, que no se encuentre ningún vecino mejor. Además, se añade como criterio de parada realizar un máximo de 100.000 consultas de la función objetivo (explorar 100.000 vecinos nuevos).

Como implementación hay que tener en cuenta que el código realizado es muy modular, luego el pseudocódigo lo será también, separando cada método/función en distintos códigos, que se llamarán entre ellos.

En el `Pseudocódigo 3.1` se hace uso de dos funciones de la clase **PAR** que hemos nombrado en el apartado **2** que es la clase encargada de representar el problema y de operar con las instancias y los clústers. Hay que destacar que la función `par.buscarPrimerVecinoMejor(iteraciones)` nos devuelve un booleano que nos indica si se ha encontrado algún vecino mejor o estamos ya en el mejor, algo que indica el final del algoritmo. A continuación se procede a describir ambas funciones.

Para los pseudocódigos posteriores recordar los atributos de `cluster` y `inst_belong` de la clase `PAR`. De la misma manera `funcion_objetivo` es un atributo de la clase (apartado **2.3**).

En el `Pseudocódigo 3.3` el formato del vecindario virtual es un vector de dimensión el número de vecinos factibles por 2, donde en la primera columna se tiene la instancia que se va a cambiar y en la segunda al clúster que se va a cambiar.

```
1: ALGORITMO DE BÚSQUEDA LOCAL
2: ARGS: 1: PAR par, 2: entero seed
3:
4:       Set_random(seed)       // Se fija la semilla
5:
6:       // Inicializamos aleatoriamente la asignación de clústers
7:       par.asignarInstanciasAleatoriamente()
8:
9:       // Buscamos mejores vecinos hasta llegar al mejor
10:      while(par.buscarPrimerVecinoMejor(100000))
11:
12: output par       // Contiene la solución encontrada
```

`Pseudocódigo 3.1`. Función que ejecuta el algoritmo con cierta semilla.

```
1: ALGORITMO DE BÚSQUEDA LOCAL: asignarInstanciasAleatoriamente()
2:     vacíamos los clústers ( tanto clusters como inst_belong)
3:
4:     // Asignamos a cada instancia un clúster aleatorio
5:     for i ∈ instancias do
6:         entero cluster_aleatorio = Randint(0,k − 1)
7:         inst_belong[i]=cluster_aleatorio
8:         clusters[cluster_aleatorio].insertarAlFinal(i)
9:     end for
10:
11:    // Arreglamos las restricciones que no se cumplan
12:    Barajamos los índices para recorrer de forma aleatoria las instancias
13:    for c ∈ {0, ..., k − 1} do        //Para cada clúster
14:        if size(clusters[c])==0 do        //Si está vacío
15:            for i ∈ instancias do        //Para cada instancia
16:                if size(clusters[inst_belong[i]])>1 do
17:                    Asignamos i al clúster c (en clusters y en inst_belong)
```

Pseudocódigo 3.2. Método que inicializa los clústers de forma aleatoria.

```
1: ALGORITMO DE B. LOCAL: buscarPrimerVecinoMejor(iteraciones)
2:     bool hay_cambio = falso        // Dice si se ha encontrado un vecino mejor
3:
4:     // Generamos vecindario virtual
5:     for i ∈ instancias do
6:         for c ∈ {0, ..., k − 1} do
7:             // Comprobamos que el clúster nuevo no es el actual de la
8:             // instancia y que no dejamos vacío el clúster antiguo
9:             if inst_belong[i]!=c and size(clusters[inst_belong[i]])>1 do
10:                vec_virtual.insertar([i, c])
11:
12:    Barajamos el vecindario
13:    for v ∈vec_virtual do if (not hay_cambio and iterations_BL<iteraciones) do
14:        hay_cambio = cambioClusterMejor(vec_virtual[v[0]], vec_virtual[v[1]])
15:    return hay_cambio and iterations_BL<iteraciones
```

Pseudocódigo 3.3. Método que busca el primer vecino mejor.

```
1: ALGORITMO DE BL: cambioClusterMejor(entero instancia, entero cluster)
2:     flotante fitnessold = funcion_objetivo
3:     Tomamos como nueva solución el vecino
4:
5:     bool mejora = verdadero        // Dice si se ha encontrado un vecino mejor
6:     if fitnessFunction() >= fitnessold do        // Si el vecino es peor
7:         Deshacemos el cambio
8:         mejora = falso
9:     return mejora
```

Pseudocódigo 3.4. Método que valora si el cambio al vecino mejora.

4. Algoritmo greedy de comparación COPKM

El algoritmo greedy COPKM genera un agrupamiento basado en el famoso algoritmo k-medias pero teniendo en cuenta la satisfacción de restricciones asociadas al conjunto de datos. El algoritmo COPKM hace también una interpretación débil de las restricciones. El algoritmo k-medias clásico minimiza únicamente la desviación general. Al modificarlo para tener en cuenta las restricciones el algoritmo acomete un problema más complejo y puede tardar más en encontrar una solución.

ALGORITMO GREEDY COPKM CONCEPTUAL

1. Se generan k centroides iniciales de forma aleatoria dentro del dominio asociado a cada dimensión del conjunto de datos.
2. Se barajan los índices de las instancias para recorrerlas de forma aleatoria sin repetición.
3. Se asigna cada instancia del conjunto de datos al grupo más cercano (aquel con centroide más cercano) que viole el menor número de restricciones ML y CL.
4. Se actualizan los centroides de cada grupo de acuerdo al promedio de los valores de las instancias asociadas a su grupo.
5. Se repiten los dos pasos anteriores mientras que al menos un grupo haya sufrido algún cambio.

En el `Pseudocódigo 4.1` hay que notar que la función a la que se hace referencia que se detalla en el `Pseudocódigo 4.2` te devuelve un bool: verdadero si los centroides han cambiado, y falso si no. Luego la condición de parada es que los centroides no cambien de una iteración a otra.

En el `Pseudocódigo 4.2` he acortado algunos pasos que se puede considerar triviales como la actualización de los clústers que solo consiste en cambiar, añadir y borrar elementos. Se hace uso de funciones como `distanciaEntreDosPuntos (=distancia)` que hacen referencia a la distancia euclídea. Otra parte del código que he omitido es cuando hay que buscar un mínimo: solo muestro como lo hago la primera vez, con la búsqueda de los clústers con los que menos incremento el infeasibility. El clúster más cercano y la instancia más cercana obvio el procedimiento.

En el `Pseudocódigo 4.3` al ver las restricciones, nuestro i está fijo, lo único que variamos es la j. Intuitivamente esto es mirar en la fila i de la matriz de restricciones para ver cuáles se incumplen de esa fila.

```
1:  ALGORITMO GREEDY COPKM
2:  ARGS: 1: PAR par, 2: entero seed
3:
4:      Set_random(seed)      // Se fija la semilla
5:
6:      Inicializamos los centroides con Rand() para cada coordenada de cada centroide.
7:      par.setCentroides(centroides)      // Asigna los centroides
8:
9:      // Buscamos mejores vecinos hasta llegar al mejor
10:     barajamos las instancias
11:     while(par.asignarInstanciasAClustersCercanos())
12:
13: output par      // Contiene la solución encontrada
```

`Pseudocódigo 4.1`. Función que ejecuta el algoritmo con cierta semilla.

```
1: ALGORITMO GREEDY COPKM: asignarInstanciasAClustersCercanos()
2:       vacíamos los clústers (solo la variable clusters)
3:       // PRIMER PASO Asignamos a cada instancia su clúster más cercano
4:       for i ∈ instancias do       // recorremos de forma aleatoria
5:             // Comprobamos las restricciones que se han violado
6:             menor_infeas = incrementoInfeasibility(i, inst_belong(i),0)
7:             restricciones_violadas[0]=menor_infeas
8:             for c ∈ clusters do       // Para cada clúster
9:                   infeas = incrementoInfeasibility(i, inst_belong(i),c)
10:                  restricciones_violadas[c]=infeas
11:                  if menor_infeas > infeas do
12:                        menor_infeas=infeas
13:
14:            // Cogemos los clústers de mínimo infeas
15:            candidatos={j : restricciones_violadas[j] == menor_infeas}
16:            // Cogemos el clúster más cercano de los candidatos
17:            cluster_cercano = j ∈ candidatos tal que j minimice al distancia al centroide
18:            // Guardamos la info
19:            inst_belong[i] = cluster_cercano
20:            clusters[cluster_cercano].insertar(i)
21:
22:      // SEGUNDO PASO Comprobamos las restricciones fuertes
23:      for c ∈clusters do
24:            if size(c) == 0 do // Si está vacío
25:                  candidatos={i ∈ instancias : size(clusters[ins_belong[i]]> 1}
26:
27:                  i_cerca=candidatos[i] tal que i minimice la distancia al centroide
28:                  Actualizamos el clúster (tanto en clusters como en inst_belong)
29:
30:      return updateCentroides() // TERCER PASO. Actualizamos los centroides
```

Pseudocódigo 4.2. Método que asigna a cada instancia el clúster más cercano.

```
1: ALGORITMO GREEDY COPKM: incrementoInfeasibility(i, c1, c2)
2:       return restriccionesConInstancia(i,c2) - restriccionesConInstancia(i,c1)
```

Pseudocódigo 4.3. Calcula el incremento de infeasibility al cambiar de clúster.

```
1: ALGORITMO GREEDY COPKM: restriccionesConInstancia(i, c)
2:       // Simulamos que la instancia pertenece al cluster
3:       anterior_cluster = inst_belong[i] and inst_belong[i] = c
4:
5:       infeas = size({(i, j) : j ≠ i and ((R[i, j] = 1 and inst_belong(i) ≠ inst_belong(j)) or
   (R[i, j] = −1 and inst_belong(i) = inst_belong(j)))})       // i es nuestro parámetro
6:
7:       inst_belong[i] = anterior_cluster       // Deshacemos la simulación
8:
9:       return infeas
```

Pseudocódigo 4.4. Calcula el infeasibility que produce una instancia al estar en un cluster.

5. Algoritmos Genéticos

Los algoritmos genétios son algoritmos de optimización, búsqueda y aprendizaje inspirados en los procesos de Evolución Natural y Evolución Genética. Estos algoritmos usan el concepto de población de soluciones. Esta población evolucionará en cada generación a través de los procesos de selección, cruce, mutación y reemplazamiento.

ALGORITMO GENÉTICO

1. Generación=0 $\longrightarrow$ Inicializar Población(Generación) $\longrightarrow$ Evaluar Población(Generación).
2. Generación = Generación + 1
3. Seleccionar Población' a partir de Población(Generación-1)
4. Recombinar Población'
5. Mutar Población'
6. Reemplazar Población(Generación) a partir de Población(Generación-1) y Población'
7. Evaluar Población(Generación)
8. Si no se ha llegado a una condición de parada volver al punto 2

Se han implementado 4 tipos distintos de algoritmos genéticos. Las variantes se basan en el tipo de modelo (Generacional y Estacionario) y en el tipo de cruce (Uniforme y Segmento Fijo). Así los 4 tipos surgen de las combinaciones entre el tipo de modelo y el tipo de cruce.

Se hablará primero de las partes comunes para posteriormente hablar de cada tipo. Para los pseudocódigos se recomienda recordar la estructura de la clase PAR. En concreto, es esencial entender que el atributo `poblacion` está SIEMPRE ordenado. Si se insertase un elemento en `poblacion`, gracias a un método de inserción ordenada implementado se metería de forma ordenada y en caso de superar el tamaño de la población se borraría el peor (el último).

5.1. Aspectos comunes genéticos

```
1: ALGORITMO GENÉTICO
2: ARGS: 1: PAR par, 2: entero tamaño, 3: entero tipo, 4: entero cruce, 5: entero seed
3:
4:      Set_random(seed)        // Se fija la semilla
5:      Asignamos el tamaño de la población
6:      par.crearPoblacionAleatoria()
7:
8:      while(Evaluaciones_función_objetivo < 100.000):
9:          // Cambio generacional P(t)←P(t+1)
10:         par.runEpoch(tipo,cruce))
11:
12:     par.finishEpochs()      // Nos quedamos con el mejor
13:
14: output par      // Contiene la solución encontrada
```

`Pseudocódigo 5.1`. Función que ejecuta el algoritmo Genético ([Generacional-Estacionario],[Cruce Uniforme-Segmento Fijo]) con cierta semilla.

```
1:  ALGORITMO GENÉTICO: crearPoblaconAleatoria()
2:      for(i=0; i<TAM_POBLACION; i++)
3:          par.asignarInstanciasAleatoriamente()      // Pseudocódigo 3.2
4:          // La solución aleatoria está en inst_belong
5:          cromosoma = inst_belong.copy()
6:          cromosoma.add(epoca_actual); cromosoma.add(funcion_objetivo);
7:
8:          // Insertamos de forma ordenada según la función objetivo
9:          poblacion.insertarOrdenado(cromosoma)
10:
11:     // Actualizamos el número de evaluaciones de la función objetivo
12:     iterations_ff+=50
13:
14: output poblacion aleatoria
```

Pseudocódigo 5.2. Método que inicializa aleatoriamente la población.

```
1:  ALGORITMO GENÉTICO: runEpoch(tipo,cruce)
2:      Si tipo==Estacionario ⇒ tam_pob=2, ncruces=2
3:      Si tipo==Generacional ⇒ tam_pob=TAM_POBLACION, ncruces=36
4:
5:      // Selección
6:      for(i=0; i<tam_pob; i++):
7:          rand1,rand2∈{0,1,...,TAM_POBLACION-1}
8:          random ← mín{rand1,rand2}
9:          poblacion_padres.add(poblacion[random])
10:
11:     // Cruce
12:     for(it1=0; it1<ncruces/2; it1++):
13:         for(it2=0; it2<2; it2++):
14:             Si cruce==UNIFORME:
15:                 operadorCruceUniforme(poblacion_padres[2*it1],
                                          poblacion_padres[2*it1+1],hijo)
16:             Si cruce==SEGMENTO FIJO:
17:                 operadorCruceSegmentoFijo(poblacion_padres[2*it1],
                                              poblacion_padres[2*it1+1],hijo)
18:             poblacion_hijos.add(hijo);
19:     Si Generacional ⇒ añadimos los padres que no se han cruzado a poblacion_hijos
20:
21:     // Mutación
22:     operadorMutacion(poblacion_hijos, 0.1*tam_pob)
23:
24:     // Reemplazamiento
25:     Si Estacionario ⇒ poblacion.insertarOrdenado(hijo) (∀ hijo ∈ poblacion_hijos)
26:     Si Generacional ⇒
27:         guardar el mejor Vaciar poblacion
28:         poblacion ← poblacion_hijos; eliminar el peor hijo y meter el mejor anterior.
29:     epoca++;
30: output poblacion(epoca)
```

Pseudocódigo 5.3. Método que avanza una generación (P(t)← $P(t+1)$).

```
1: ALGORITMO GENÉTICO: finishEpochs()
2:   //Cogemos la parte de la solucion del cromosoma 0 (n primeras componentes)
3:       inst_belong ← solucion(poblacion[0])
4: output solución algoritmo genético
```

`Pseudocódigo 5.4`. Escoge como solución el mejor de la población y termina la evolución.

```
1: ALGORITMO GENÉTICO: operadorMutacion(poblacion2,ngenes)
2:       // Si toca mutar esta época
3:       Si ngenes>1 or (0<ngnes<1 and epoca %(1/ngenes)==0):
4:             for(it=0; it<ngenes;it++):
5:                   gen ← Rand(0,TAM_POBLACION*Genes_x_Cromosoma-1)
6:                   i=gen/num_instancias; j= gen %num_instancias;
7:                   Hacer:
8:                         new_cluster ← Rand(0,num_clases-2)
9:                         Si new_cluster≥ old_cluster ⇒ new_cluster++;
10:                        pobacion2[i][j]=new_cluster;
11:                        Evaluamos la solución poblacion2[i] y comprobamos restricciones.
12:                        iterations_ff++  // Incrementamos evaluaciones fun. obj.
13:                  Mientras no haya cambiado el clúster antiguo
14:                  Actualizamos el valor de la fun. obj. de pob[i]
15: output Población_Hijos(t) tras proceso de mutación
```

`Pseudocódigo 5.5`. Implementa el operador de mutación.

En los `Pseudocódigos` de arriba aparece varias veces la función `insertarOrdenado(cromosoma)`. Esta función lo que añade es insertar de forma ordenada el cromosoma, teniendo una ordenación de menor a mayor del valor de la función objetivo. Si se superase el tamaño de la población permitida (que es 50), se añadiría el cromosoma en la posición correspondiente (tendríamos un tamaño de 51) y se eliminaría el peor de esos 51, es decir, el último.

En el `Pseudocódigo 5.3`, en el proceso de selección, nos quedamos con el menor aleatorio puesto que el vector población ya está ordenado.

5.2. Genético Generacional y Estacionario

Estas son las dos variantes de modelos Genéticos. El **Generacional** basa su proceso en reemplazar toda una generación por una nueva del mismo tamaño. El proceso sería el siguiente:

1. Selección con 50 torneos aleatorios de padres
2. Cruzamos los 36 primeros padres en parejas de 2, formando dos hijos cada pareja de padres. Los 14 padres restantes se añaden a la población de hijos.
3. De la nueva población de hijos, mutan 5 genes
4. Se reemplaza con elitismo: La población de hijos pasa a ser la población actual, donde el peor hijo es sustituido por el mejor de la población anterior.

El **Estacionario**, por su parte, basa su proceso en generar únicamente 2 hijos nuevos, los cuales tienen que competir con el resto de soluciones para entrar a la población. El proceso sería el siguiente:

1. Selección con dos torneos aleatorios de padres
2. Cruzamos los dos padres formando dos hijos.
3. De la nueva población de hijos, muta 1 gen cada 5 épocas, puesto que la probabilidad de mutación es de 0.2.
4. En el proceso de reemplazamiento los hijos compiten contra el resto de soluciones para entrar en $P(t+1)$.

Con el parámetro `tipo` podremos elegir el tipo de modelo genético que queremos en los `Pseudocódigos 5`.

5.3. Operadores de cruce Uniforme(UN) y Segmento Fijo(SF)

El operador de cruce uniforme y segmento fijo se pueden ver implementados en los `Pseudocódigo 5.6` y `Pseudocódigo 5.7` respectivamente. El Uniforme asigna la mitad de los genes de un padre y la mitad del otro, donde estos genes guardan su posición de cromosoma en el cromosoma hijo. Por otro lado, en el Segmento Fijo asignamos todo un segmento del padre1 al hijo, quedando el resto del cromosoma hijo asignado uniformemente (como se hacía con el cruce uniforme) entre los dos padres.

```
1: ALGORITMO GENÉTICO: operadorCruceUniforme(padre1,padre2,hijo)
2:       indx = Shuffle([0,1,...,num_instancias-1])
3:       for(i=0; i<num_instancias;i++):
4:             Si i<num_instancias ⇒ hijo[ indx[i] ] = padre1[ indx[i] ]
5:             En otro caso ⇒ hijo[ indx[i] ] = padre2[ indx[i] ]
6:
7:       Evaluamos la solución hijo
8:       hijo.add(epoca); hijo.add(funcion_objetivo)
9:
10:      iterations_ff++  // Actualizamos las evaluaciones de la fun.obj.
11: output hijo cruzado
```

`Pseudocódigo 5.6`. Implementa el operador de Cruce Uniforme.

```
1: ALGORITMO GENÉTICO: operadorCruceSegmentoFijo(padre1,padre2,hijo)
2:       tam ← aleatorio ∈ {1,..,n_instancias}
3:       pos_ini ← aleatorio ∈ {0,...,num_instancias-1}
4:       hijo[(i+pos_ini) %n_instancias]=padre1[(i+pos_ini) %n_instancias] i ∈ {0, .., tam − 1}
5:
6:       Asignamos uniformemente entre padre1 y padre2 el resto.
7:
8:       Evaluamos la solución hijo
9:       hijo.add(epoca); hijo.add(funcion_objetivo)
10:
11:      iterations_ff++  // Actualizamos las evaluaciones de la fun.obj.
12: output hijo cruzado
```

`Pseudocódigo 5.6`. Implementa el operador de Cruce Segmento Fijo.

En todos los casos los parámetros son pasados por referencia, tanto para aumentar la velocidad de cómputo como para que se hagan los cambios directamente.

6. Algoritmos Meméticos

Los algoritmos meméticos son algoritmos basados en la evolución de poblaciones que para realizar búsqueda heurística intenta utilizar todo el conocimiento sobre el problema (usualmente conocimiento en términos de algoritmos específicos de búsqueda local para el problema). Esta hibridación es interesante puesto que los algoritmos evolutivos son buenos exploradores pero malos explotadores. Por otro lado los algoritmos de búsqueda local son malos exploradores pero buenos explotadores. Así, se intenta coger lo mejor de ambos intentando que los algoritmos meméticos sean tanto buenos exploradores como buenos explotadores.

ALGORITMO MEMÉTICO

1. Generación=0 $\longrightarrow$ Inicializar Población(Generación) $\longrightarrow$ Evaluar Población(Generación).
2. Generación = Generación + 1
3. Seleccionar Población' a partir de Población(Generación-1)
4. Recombinar Población'
5. Mutar Población'
6. Realizamos búsqueda local
7. Reemplazar Población(Generación) a partir de Población(Generación-1) y Población'
8. Evaluar Población(Generación)
9. Si no se ha llegado a una condición de parada volver al punto 2

Se proponen 3 tipos distintos de meméticos

- AM-(10,1.0): Cada 10 generaciones, se aplica la BLS sobre todos los cromosomas de P.
- AM-(10,0.1): Cada 10 generaciones, se aplica la BLS sobre un subconjunto de cromosomas de P seleccionado aleatoriamente con probabilidad igual a 0.1 para cada cromosoma.
- AM-(10,0.1mej): Cada 10 generaciones, aplicar la BLS sobre los 0.1N mejores cromosomas de la P actual (N es el tamaño de ésta).

```
1:  ALGORITMO MEMÉTICO
2:  ARGS: 1: PAR par, 2: entero tamaño, 3: entero bls, 4: real prob, 5: booleano best,
          6: entero seed
3:
4:        Set_random(seed)        // Se fija la semilla
5:        Asignamos el tamaño de la población
6:        par.crearPoblacionAleatoria()  // Pseudocódigo 5.2
7:
8:        while(Evaluaciones_función_objetivo < 100.000):
9:              // Cambio generacional P(t)←P(t+1)
10:             par.runEpoch(2,2,bls,prob,best))
11:
12:       par.finishEpochs()      // Nos quedamos con el mejor, Pseudocódigo 5.4
13:
14: output par       // Contiene la solución encontrada
```

Pseudocódigo 6.1. Función que ejecuta el algoritmo Memético con cierta semilla.

En nuestro caso el algoritmo de búsqueda local será un algoritmo de búsqueda local suave. Este viene descrito en el `Pseudocódigo 6.3`. En el `Pseudocódigo 6.1` tenemos nuevos parámetros: bls (la búsqueda local suave se ejcutará cada `bls` épocas), prob (probabilidad de que se le aplique BLS a un cromosoma), best (si está a true, cogemos los mejores para hacer BLS). El `Pseudocódigo 6.2` es una modificación del `Pseudocódigo 5.3`.

```
1: ALGORITMO MEMÉTICO: runEpoch(tipo,cruce,bls,prob,best)
2:       // Proceso de Selección...
3:
4:       // Proceso de Cruce...
5:
6:       // Proceso de Mutación...
7:
8:       // Búsqueda Local Suave
9:       Si bls>0 and epoca%bls==0:
10:           // Si hay que coger los mejores
11:           Si best and prob<1:
12:                 ordenamos poblacion_hijos según la función objetivo
13:
14:           Para cada hijo que esté en una posición menor a prob*tam_pob
15:                 busquedaLocalSuave(hijo,0.1*n_instancias)
16:
17:      // Proceso de Reemplazamiento..
18:
19:      epoca++;
20: output poblacion(epoca)
```

`Pseudocódigo 6.2`. Método que avanza una generación (P(t)$\leftarrow P(t+1)$). Los procesos comentados pueden verse desarrollados en el `Pseudocódigo 5.3`.

```
1: ALGORITMO MEMÉTICO: busquedaLocalSuave(solucion,fallos)
2:       Evaluamos la solución actual y la guardamos
3:       f ← 0, mejora ← true, i ← 0, mejor_solucion← solucion
4:       Barajamos indices
5:       Mientras (mejora or f<fallos) and i< n_instancias:
6:             clust_old ← solucion[i],mejor_clust ← clust_old,f_obj_mejor ← f_obj_actual
7:             Si hay más de un elemento en el clúster de la instancia i
8:                   Para cada clúster c ≠ de clust_old
9:                         Simulamos el cambio por c
10:                        iterations_ff++
11:                        Si f_obj_mejor > f_obj_actual  // Mejora
12:                              actualizamos mejor_clust=c, f_obj_mejor=f_obj_actual
13:                              mejor_solucion=solucion
14:            solucion=mejor_solucion // Nos quedamos con la mejor solución
15:            Si mejor_clust==clust_old ⇒ hemos fallado, f++
16:            En otro caso mejora=true
17:            i++;
18:
19: output explotación en poblacion_hijos
```

`Pseudocódigo 6.3`. Método que realiza una búsqueda local suave sobre una población.

7. Algoritmo de Enfriamiento Simulado

El algoritmo de Enfriamiento o Recocido Simulado es un algoritmo de búsqueda por entornos con un criterio probabilístico de aceptación de soluciones basado en Termodinámica. Son un modo de evitar que la búsqueda local finalice en óptimos locales, hecho que suele ocurrir con los algoritmos tradicionales de búsqueda local, es permitir que algunos movimientos sean hacia soluciones peores.

En el caso del Enfriamiento Simulado (ES), esto se realiza controlando la frecuencia de los movimientos de escape mediante una función de probabilidad que hará disminuir la probabilidad de estos movimientos hacia soluciones peores conforme avanza la búsqueda (y por tanto estamos más cerca, previsiblemente, del óptimo local). Se aplica la filosofía de diversificar al inicio e intensificar al final.

ALGORITMO ES

1. $T = T_0$, $S = SolucionAleatoria$, $MejorSolucion = S$
2. Mientras que se hayan generado menos de $maxvecinos$, halla menos éxitos que $maxexitos$, no se halla llegado a la temperatura final y las evaluaciones de la función objetivo sean menos que 100K
   - *a*) Cogemos un vecino de la solución $S' = Vecino$ y calculamos su incremento de la función objetivo
   - *b*) Si Mejora con respecto S o Si entra dentro de nuestra probabilidad de aceptación
     1) Actualizamos $S = S'$
     2) Si ha mejorado a la mejor solución actualizamos $MejorSolucion = S$
   - *c*) Actualizamos la temperatura $T = g(T)$

En este algoritmo tenemos bastantes parámetros de los que depende el desempeño del mismo. Estos parámetros son: temperaturas (valores iniciales y finales; y función de asignación $T_{i+1} = g(T_i)$, el número de maximos vecinos y de máximos éxitos. En la temperatura inicial calculada como $T_0 = \frac{\mu C(S_0)}{-ln(\phi)}$, luego la temperatura inicial va a depender de los parámetros μ(tanto por uno de empeoramiento permitido) y ϕ(probabilidad de aceptar las soluciones admitidas).

```
1: ALGORITMO ENFRIAMIENTO SIMULADO
2: ARGS: 1: PAR par, 2: entero seed
3:
4:       Set_random(seed)
5:       max_vecinos=10*NumInstancias, max_exitos=0.1*max_vecinos
6:       par.crearSolucionAleatoria()
7:       mu=phi=0.3, Ti=(mu*par.fitnessFunction())/(-log(phi)), Tf=0.001, M=10000
8:       temperaturas=generarTemperaturas(Ti,Tf,M)
9:       it=0, para=false
10:
11:      While val_fun_obj≤100000 and it< M + 1 and para=false:
12:            para = par.enfriamientoSimulado(temperaturas[it],max_vecinos,max_exitos)
13:      it++
14:      par.simularMejorSolucion() //Nos quedamos con la mejor solución
15: output par
```

`Pseudocódigo 7.1`. Función que ejecuta el algoritmo de Enfriamiento Simulado.

```
1: ALGORITMO ES: generarTemperaturas(Ti,Tf,M)
2:
3:       beta = (Ti-Tf)/(M*Ti*Tf)
4:
5:       //Se generan los M enfriamientos
6:       vector temperaturas(M+1); temperaturas[0]=Ti
7:       temperaturas[i+1]=temperaturas[i]/(1+beta*temperaturas[i]) Para todo i<M
8:
9: output temperaturas
```

`Pseudocódigo 7.2`. Función que genera el vector de temperaturas.

```
1: ALGORITMO ES: enfriamientoSimulado(T,max_vec,max_ex)
2:
3:       solucion ← mejor solución encontrada hasta ahora
4:       mejor_fitnes ← fitness de la mejor solución encontrada hasta ahora
5:       exitos=0, f_actual=mejor_fitness
6:
7:       For(i=0; i<max_vec and exitos<max_exitos and iterations_ff< 100000;i++):
8:             instancia_random = instancia elegida aleatoriamente perteneciente
9:                                a un clúster que tenga más de una instancia
10:            cluster_anterior = inst_belong[instancia_random]
11:            cluster_random = cluster aleatorio distinto a cluser_anterior
12:            solucion[instancia_random]=cluster_random
13:            # Calculamos la función objetivo con la nueva solución
14:
15:            inc_fit=funcion_objetivo - f_actual
16:            Si inc_fit>0 and Rand()> e^(-inc_fit/T)
17:                  // Deshacemos el cambio
18:                  solucion[inst_rando]=cluster_anterior;
19:            en otro caso:
20:                  exitos++, f_actual=funcion_objetivo
21:                  Si funcion_objetivo < mejor_fitness
22:                        // Actualizamos la mejor solución
23:                        mejor_fitness = funcion_objetivo
24:                        mejor_solucion = inst_belong
25:
26: output exitos≤0
```

`Pseudocódigo 7.3`. Método que ejecuta una iteración de enfriamiento.

```
1: ALGORITMO ES: simularMejorSolucion()
2:
3: //Cogemos mejor_solucion y la establecemos como solucion actual de la clase
4:
```

`Pseudocódigo 7.3`. Establece la mejor solución como solución actual de la clase

En el `Pseudocódigo 7.3`, hay que recordar que `funcion_objetivo` es un atributo de la clase PAR que guarda el valor de la función objetivo de la solución actual. Hay que recordar también que el vector `inst_belong` contiene en la posición i el clúster al que pertenece la instancia i.

8. Algoritmo de Búsqueda Multiarranque Básica

El algoritmo de Búsqueda Multiarranque Básica consiste en generar un determinado número de soluciones aleatorias iniciales y optimizar cada una de ellas con el algoritmo de Búsqueda Local considerado, devolviendo la mejor solución encontrada.

Las soluciones iniciales se generan al azar en la región factible del problema, y la etapa de búsqueda se realiza mediante nuestro algoritmo de búsqueda local. Este método converge al óptimo global del problema si y sólo si el número de soluciones iniciales generadas tiende a infinito. Este procedimiento es muy ineficiente puesto que se pueden generar muchos puntos cercanos entre sí, de modo que al aplicarles la búsqueda local se obtendrán los mismos óptimos locales.

ALGORITMO BMB

1. S ← Generamos solución aleatoria
2. S' ← buscamos óptimo local a través de BL
3. Actualizamos la mejor solución obtenida hasta el momento
4. Volver al paso si no se cumple la condición de parada

Cabe notar que en el `Pseudocódigo 8.1`, la propia clase `PAR` almacena la mejor solución visitada hasta el momento. Es por ello que en todo momento, en cada una de las 10 iteraciones, tiene almacenada la mejor solución que ha visitado. Lo que hacemos en la línea 9 es una operación que pone esa mejor solución como la solución actual del objeto `PAR`.

Por otro lado los métodos utilizados ya se han explicado con anterioridad, es por ello por lo que están en color **verde**.

```
1: ALGORITMO BÚSQUEDA MULTIARRANQUE BÁSICA
2: ARGS: 1: PAR par, 2: entero seed
3:
4:       Set_random(seed)
5:       for(i=0; i<10; i++):
6:             par.crearSolucionAleatoria()
7:             while(par.buscarPrimerVecinoMejor(10.000)
8:
9:       # Nos quedamos con la mejor solución de todas las que hemos visitado
10:
11: output par
```

`Pseudocódigo 8.1`. Función que ejecuta el algoritmo de Búsqueda Multiarranque Básica con cierta semilla.

9. Algoritmo de Búsqueda Local Reiterada

El algoritmo de búsqueda local reiterada (ILS) consiste en generar una solución inicial aleatoria y aplicar el algoritmo de búsqueda local sobre ella. Una vez obtenida la solución optimizada, se estudiará si es mejor que la solución encontrada hasta el momento y se realizará una mutación sobre la mejor de estas dos. A continuación se vuelve a ejecutar BL con la solución mutada. La solución optimizada se volverá a comparar con la mejor y empezaría de nuevo el ciclo. El operador de mutación consiste, en cierto modo, en hacer un cruce de segmento fijo consigo mismo y mutando el resto.

Así, la aplicación de la ILS necesita de la definición de cuatro componentes:

1. Una solución inicial
2. Un procedimiento de mutación que aplica un cambio brusco sobre la solución actual para obtener una solución intermedia
3. Un procedimiento de búsqueda local
4. Un criterio de aceptación que decide a qué solución se aplica el procedimiento de modificación

```
1: ALGORITMO BÚSQUEDA LOCAL REITERADA
2: ARGS: 1: PAR par, 2: entero seed
3:
4:       Set_random(seed)
5:       size = 0.1*num_instancias
6:       par.crearSolucionAleatoria()
7:       while(par.buscarPrimerVecinoMejor(10.000)
8:
9:       for(i=0; i<10; i++):
10:            par.mutacionILS(size)
11:            while(par.buscarPrimerVecinoMejor(10.000)
12:
13:      # Nos quedamos con la mejor solución de todas las que hemos visitado
14:
15: output par
```

Pseudocódigo 9.1. Función que ejecuta el algoritmo de Búsqueda Local Reiterada.

```
1: ALGORITMO BÚSQUEDA LOCAL REITERADA: mutacionILS(size)
2:
3:       inicio = aleatorio∈ {0, ..., Num_instancias − 1}
4:       for(t=0; t<size; t++):
5:            i=(t+inicio) %num_instancias
6:            solucion[i]= aleatorio∈ {0, ..., Num_clases − 1}
7:
8:       # Comprobamos que la solución respeta las restricciones fuertes y sino
   las arreglamos
9:
10: output par
```

Pseudocódigo 9.2. Operador de mutación para ILS.

9.1. Algoritmo Híbrido ILS-ES

Este algoritmo tendrá la misma composición que el ILS que acabamos de ver, con la única diferencia de que el algoritmo de búsqueda por trayectorias simples considerado para refinar las soluciones iniciales será el ES visto en la sección 7.

```
1: ALGORITMO BÚSQUEDA LOCAL REITERADA CON ES
2: ARGS: 1: PAR par, 2: entero seed
3:
4:       Set_random(seed)
5:       size = 0.1*num_instancias
6:       max_vec=10*NumInstancias, max_ex=0.1*max_vec
7:       par.crearSolucionAleatoria()
8:       mu=phi=0.3, Ti=(mu*par.fitnessFunction())/(-log(phi)), Tf=0.001, M=10000
9:       temperaturas=generarTemperaturas(Ti,Tf,M)
10:
11:      it=0, para=false
12:      While val_fun_obj≤10000 and it< M + 1 and para=false:
13:            para = par.enfriamientoSimulado(temperaturas[it],max_vec,max_ex)
14:            it++
15:
16:      for(i=1; i<10; i++):
17:            par.mutacionILS(size)
18:            it=0, para=false
19:            While val_fun_obj≤10000 and it< M + 1 and para=false:
20:                  para = par.enfriamientoSimulado(temperaturas[it],max_vec,max_ex)
21:                  it++
22:
23:      # Nos quedamos con la mejor solución de todas las que hemos visitado
24:
25: output par
```

`Pseudocódigo 9.1.1`. Función que ejecuta el algoritmo de Búsqueda Local Reiterada con ES.

10. Procedimiento considerado para el desarrollo de la práctica

En este apartado se hablará de los aspectos relacionados con el entorno de programación usado, fuentes externas de código usadas y temas por el estilo. Se aportará también un manual de usuario que permitirá saber cómo funciona el programa para que cualquier persona que lea el manual sea capaz de utilizar el programa desde su propio ordenador.

El lenguaje de programación utilizado ha sido **C++**. La elección de este ha sido porque es un lenguaje muy versátil que permite una modularización adecuada al problema PAR. El uso de clases para representar el problema facilita a mi parecer mucho la tarea. Se ha usado para programar **Visual Studio Code**: entorno de mucho reconocimiento en la industria que permite la compilación del programa así como ayuda en la programación.

Para la **medición de tiempo** se ha usado la librería `chrono`. Se ha usado también librerías como `vector` y `math.h`. Se han usado los ficheros de `random.h` y `random.cpp` propuestos en PRADO para la generación de números aleatorios. El resto del programa se ha hecho todo desde cero, siendo el sustento teórico el seminario dado en clase y el enunciado de la práctica.

Para poder computar las **tablas** aprovechando la salida del programa sin necesidad que hacer un traspaso manual de los datos a cualquier otro tipo de programa, se ha implementado una función en la que directamente nos saca en formato LaTeX las tablas de los datos de las ejecuciones. Esta función se encuentra implementada en los ficheros `totex.h` y `totex.cpp`.

El **desarrollo** ha sido largo: montar toda la estructura del problema PAR para que sea coherente, consistente, modular y sencilla ha llevado mucho tiempo. Se ha considerado que ese tiempo invertido facilitará el desarrollo para posteriores trabajos con este mismo problema. La estructuración del problema ha seguido el hilo de preferir sacrificar memoria para ganar velocidad: almacenamos muchos valores que pueden calcularse en cualquier momento y tenemos distintas representaciones para mismos conceptos como los clústers o las restricciones. También habría que dotar de lógica a los algoritmos, el cómo llamarlos y desde dónde. Así se ha considerado que un algoritmo simplemente es una función que con un par de líneas con instrucciones sobre un objeto de la clase PAR es capaz de llevar a cabo el algoritmo que representa.

Como se ha dicho, se ha apostado por la velocidad. Esto ha hecho que se hayan tenido que tomar decisiones de optimización de tiempo. Un ejemplo que evidencia esto está dentro de la implementación que han conllevado los algoritmos genéticos y meméticos: el tratado de la población para escoger el mejor requería una **ordenación**. Se decidió por lo tanto que lo más eficiente era, en vez de ordenar el vector en cada generación, vaciar el vector e insertar los nuevos cromosomas de forma ordenada en función del valor de su función objetivo. Así se ha implementado un método `insertPoblacion` que consigue insertar de forma ordenada en un tiempo de orden $\mathcal{O}(log_2 n)$.

Se ha incluído el fichero `random.h` y `random.cpp` sugerido en PRADO para la generación de **números aleatorios**. A estos archivos se les ha añadido una nueva función que te desordena un vector de índices, algo que será útil para recorrer vectores de forma aleatoria:

```
1: vector<int> ShuffleIndices(int size);
```

donde `size` define el tamaño del vector de salida. Es decir, si queremos recorrer de forma aleatoria un vector de tamaño n tendremos que hacer:

```
#import <vector>
#import "random.h"

using namespace std;

main(){
        vector<int> vector_a_recorrer(n);
        vector<int> indices_shuffled = ShuffleIndices(n);
        for(int i; i < n; i++){
                vector_a_recorrer[ indices_shuffled[i] ];
        }
}
```

Una vez montado todo el código viendo que arroja resultados coherentes se ha procedido a la captación de datos y experimentos.

10.1. Manual de usuario

Llegados a este punto toca hablar de la distribución de ficheros. Tenemos las carpetas:

carpeta	bin	data	include	obj	src
	ejecutable	bupa_set.dat	busqlocal.h	archivos .obj	busqlocal.cpp
		bupa_set_const_10.const	genetico.h	generados	genetico.cpp
A		bupa_set_const_20.const	greedyCOPKM.h	en el makefile	greedyCOPKM.cpp
R		glass_set.dat	memetico.h		main.cpp
C		glass_set_const_10.const	par.h		memetico.cpp
H		glass_set_const_20.const	random.h		par.cpp
I		zoo_set.dat	readfiles.h		random.cpp
V		zoo_set_const_10.const	totex.h		readfiles.cpp
O		zoo_set_const_20.const	enfriamientosimulado.h		totex.cpp
S			busqmulbas.h		enfriamientosimulado.cpp
			busqlocreit.h		busqmulbas.cpp
			busqlocreitES.h		busqlocareit.cpp
					busqlocreitES.cpp

Todo dentro del mismo directorio donde se encuentra el makefile. El makefile se ejecuta haciendo `make` desde la terminal. El fichero ejecutable se ejecuta haciendo:

```
./bin/ejecutable [<nombre_del_fichero>|--tex] [semilla]
```

donde `nombre_del_fichero` $\in$ {`bupa_set, glass_set, zoo_set`} y la semilla es opcional. La opción `--tex` hará que se ejecuten todas las ejecuciones con todos los conjuntos de datos y que se nos muestre por pantalla la generación de las tablas LaTeX. Se harán por defecto 55 ejecuciones con cada archivo de datos (10 por cada algoritmo, de las cuales 5 para cada tipo de restricciones).

Para modificar aspectos generales del código, como el número de pruebas o los algoritmos que usar se tendrá que modificar el fichero `main.cpp`. Para el número de iteraciones modificaremos `NUM_PRUEBAS` y para los algoritmos que usar, bien puede modificar los booleanos: `greedy_run, bl_run, geneticos_run, memeticos_run, trayectorias_run.` O bien simplemente se tiene que escribir después de la inicialización del problema PAR:

```
algoritmo(#parámetros*);
```

Para ver qué parámetros usar en cada algoritmo se puede consultar en los ficheros tanto `.h` como `.cpp` donde viene comentado la función del algoritmo y de cada parámetro.

11. Experimentos y análisis de resultados

Recordando los datos que se nos han dado, tenemos 3 bases de datos distintas y cada base de datos consta de dos ficheros distintos de restricciones, un fichero con un 10 % de restricciones y el otro con un 20 %. Así tenemos 6 problemas distintos. El procedimiento ha sido realizar 5 ejecuciones distintas inicializando con distintas semillas y hacer estadísticos con los resultados arrojados, pudiendo con ello sacar conclusiones comparando los distintos algoritmos y la distinta complejidad de las bases de datos.

Las semillas utilizadas en las cinco ejecuciones de cada problema han sido $\{1, 2, 3, 4, 5\}$. A continuación analizamos y comparamos los distintos problemas, empezando con el greedy y el de búsqueda local y continuando con los genéticos y meméticos.

Antes de empezar el análisis de los algoritmos hay que recordar la naturaleza de cada problema sabiendo que la *Tasa_Inf* en el problema de Bupa no se puede comparar directamente con la de Zoo debido a la diferencia en el número de datos. Habrá que hacer un análisis relativo de cada conjunto sacando conclusiones comunes.

Independiente de cada algoritmo, lo esperado era que la función fitness más alta sea la de Zoo, después Glass y por último Bupa. Esto es así puesto que cuantos más clústers, se consigue que estos clústers sean "más pequeños" minimizando la distancia intra-clústes. Por otro lado cuantas más instancias, mejor se coge un centroide cercano a las instancias representativas minimizando la distancia intra-clústers.

Por otro lado los tiempos se esperan que lleven un orden inverso a la función fitness, llevando más tiempo aquellos problemas con más instancias y mayor número de clústers. Bupa es el que gana con 16 clústers y 345 instancias, teniendo en cuenta que tenemos una restricción fuerte que implica que ningún clúster puede estar vacío implica muchas comprobaciones.

11.1. GreedyCOPKM y Búsqueda Local

Las tablas de resultados tras las ejecuciones son las siguientes:

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	2	1.03667	1.05182	3	4	0.389345	0.393281	14	30	0.252988	0.261028	149
Ejecución 2	0	0.92574	0.92574	4	1	0.390753	0.391737	14	12	0.245451	0.248667	152
Ejecución 3	1	1.00701	1.01459	3	3	0.395552	0.398504	14	13	0.247788	0.251272	160
Ejecución 4	6	0.994104	1.03957	4	7	0.400947	0.407836	14	13	0.241399	0.244882	185
Ejecución 5	2	1.05624	1.0714	3	0	0.3185	0.3185	16	45	0.230778	0.242837	112
Media	2.2	1.00395	1.02062	3.4	3	0.379019	0.381972	14.4	22.6	0.243681	0.249737	151.6

Tabla 1: Resultados obtenidos por el algoritmo Greedy COPKM con 10 % de restricciones

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	0	0.9048	0.9048	4	0	0.355634	0.355634	20	65	0.257117	0.266181	188
Ejecución 2	1	0.967536	0.971574	3	0	0.311382	0.311382	11	0	0.230513	0.230513	118
Ejecución 3	0	0.9048	0.9048	3	3	0.3151	0.316662	14	0	0.219861	0.219861	144
Ejecución 4	2	0.96121	0.969286	5	19	0.312456	0.322348	14	0	0.246182	0.246182	101
Ejecución 5	5	1.03427	1.05446	4	0	0.349455	0.349455	11	0	0.233939	0.233939	87
Media	1.6	0.954523	0.960984	3.8	4.4	0.328806	0.331096	14	13	0.237523	0.239335	127.6

Tabla 2: Resultados obtenidos por el algoritmo Greedy COPKM con 20 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	11	0.610733	0.694087	110	58	0.18783	0.244904	472	114	0.106857	0.137408	5199
Ejecución 2	13	0.614392	0.712902	153	33	0.212138	0.244611	446	114	0.116114	0.146664	5748
Ejecución 3	15	0.622082	0.735747	94	46	0.195533	0.240799	486	72	0.120588	0.139883	4554
Ejecución 4	13	0.622082	0.720591	81	15	0.247637	0.262398	457	113	0.108473	0.138755	5462
Ejecución 5	14	0.622495	0.728582	98	56	0.190911	0.246017	406	118	0.108912	0.140535	5945
Media	13.2	0.618357	0.718382	107.2	41.6	0.20681	0.247746	453.4	106.2	0.112189	0.140649	5381.6

Tabla 3: Resultados obtenidos por el algoritmo BL con 10 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	23	0.71821	0.811086	90	41	0.250176	0.271521	448	246	0.111261	0.145564	5294
Ejecución 2	12	0.745351	0.793808	100	131	0.208785	0.276985	449	193	0.119201	0.146114	5296
Ejecución 3	33	0.714332	0.847589	89	36	0.246058	0.2648	454	191	0.118132	0.144765	5942
Ejecución 4	30	0.718036	0.839179	110	41	0.250143	0.271488	524	197	0.117886	0.145356	7220
Ejecución 5	14	0.733343	0.789876	137	41	0.250086	0.271431	429	206	0.116568	0.145292	6572
Media	22.4	0.725855	0.816308	105.2	58	0.24105	0.271245	460.8	206.6	0.11661	0.145418	6064.8

Tabla 4: Resultados obtenidos por el algoritmo BL con 20 % de restricciones

	Zoo		Glass		Bupa	
% Restricciones	10 %	20 %	10 %	20 %	10 %	20 %
Ejecución 1	5242	4052	11360	9667	71126	57669
Ejecución 2	8034	4331	11833	10257	79438	56699
Ejecución 3	4670	3790	12278	10184	57575	64039
Ejecución 4	3761	5105	12785	12751	79194	86854
Ejecución 5	4721	6714	10469	9492	88640	75947

Tabla 5: Evaluaciones de la función objetivo con BL

11.1.1. Análisis del algoritmo Greedy COPKM

Si hay algo que se puede decir sin que tiemble el pulso es la velocidad de este algoritmo. El tiempo mayor ha sido de 196ms con los datos *Bupa* con un 20 % de restricciones en la primera ejecución. Por lo demás, se puede observar como en casi cualquier ejecución de cualquiera de los problemas la tasa de infeasibility es relativamente baja.

Destacan mucho los datos de la cuarta ejecución de Glass con 20 % de restricciones y la primera de Bupa con 20 % de restricciones, donde los valores que toman distan mucho del resto de ejecuciones. A la vista está comparándolos con la media en cada uno como se diferencian. Estos valores son sin duda culpa de una mala selección de centroides iniciales, que condicionan el problema desde el inicio, siendo dependientes de la semilla que se toma para el problema. Pero por otra parte vemos como este algoritmo no nos asegura una tasa de infeasibility perfecta.

Teniendo en cuenta justo lo que se dijo antes de comenzar el análisis de los algoritmos, con el algoritmo greedy COPKM se puede ver una tendencia a minimizar al máximo el infeasibility.

Precisamente por tomar un infeasibility tan bajo en la mayoría de los casos, su distancia media es casi mínimo, puesto que los valores de referencia se han tomado asegurando una infeasibility de 0. Sin embargo, se puede ver como la función fitness da valores relativamente elevados para tener un infeasibility tan bajo y esto se debe a como gestiona el control de las distancias el algoritmo que simplemente reúne cada instancia con su centroide más cercano con lo cual no optimiza mucho esta función objetivo.

Comparando los resultados del 10 % con los del 20 % podemos ver que siguen una misma línea común: infeasibility medios parecidos, función fitnes similar y tiempos casi idénticos. Se

han obtenido incluso mejores resultados con el 20 % de restricciones con lo que se puede concluir la independencia de este algoritmo con la tasa de restricciones que se le ponga siempre y cuando el problema siga siendo soluble.

11.1.2. Análisis del algoritmo de Búsqueda Local

Para el algoritmo de búsqueda local cabe resaltar la importancia que tiene en sus soluciones finales la guía a través de la función fitness: se observa como en ninguna ejecución de ningún problema se han conseguido respetar todas las restricciones pero, por otro lado, las distancias intra-clústers han sido menores que la de referencia, creando una media de Dist-Intra bastante alta (mayor del 0.1) y que en todos los casos es porque mejora esa distancia. En la tabla 5 se ve las veces que se ha llamado a la función fitness, (se exigía que la solución se encontrase en menos de 100000 llamadas), que, en cierto modo, nos podemos hacer una idea de las alternativas que ha rechazado porque la función fitness aumentaba. Seguramente muchos de estos aumentos tenían que ver con un aumento de la distancia intra-cluster, aunque ello conllevase una disminución en el infeasibility.

Cabe destacar también su gran coste computacional que viene reflejado con sus tiempos, llegando a tardar más de 7 segundos en algunas ejecuciones del problema Bupa.

Si comparamos su desempeño entre las restricciones del 10 % y del 20 % vemos como conforme aumentan las restricciones, la calidad de la solución sí que empeora:

- Zoo: la tasa de infeasibility pasa de 13.2 a 22.4 y la función objetivo aumenta hasta un 0.1.
- Glass: la tasa de infeasibility pasa de 41.6 a 58 y la función objetivo aumenta.
- Bupa: la tasa de infeasibility pasa de 106.2 a 206.6 y la función objetivo aumenta hasta un 0.15.

Los tiempos se mantienen más o menos estables pero se puede ver como el resto de datos que hablan de la calidad del algoritmo, empeoran. Se puede intuir que el infeasibility aumenta conforme aumentan las restricciones por la importancia que le damos a la función objetivo y la importancia que tiene el infeasibility dentro de la función objetivo. Por otro lado, el aumento de la función objetivo se debe tanto al incremento de infeasibility como al aumento de la distancia intra-cluster, que aumenta por la misma razón que el infeasibility. El algoritmo quiere llevar un claro control intermedio de ambos datos (infeasibility y distancia intra-clúster) para que ninguno se dispare y haga que nuestra función objetivo se dispare.

11.1.3. Análisis entre greedyCOPKM y Búsqueda Local

En vista a los resultados el algoritmo COPKM consigue unos resultados mejores debido a que la tasa de infeasibility es muchísimo menor en todos los casos y con ello se tiene una solución mucho más respetuosa con las restricciones que es lo que se quiere primordialmente. Esto ocurre porque el algoritmo COPKM se centra en cada iteración en que el incremento del número de restricciones incumplidas sea lo menor posible (cosa que no hace BL).

La búsqueda local se centra en cada iteración en el valor de la función objetivo y aquí sí que sale vencedora la búsqueda local, pero porque lo que intenta minimizar en todo momento es el valor de nuestra función fitness.

Por otro lado los tiempos son muy distintos, saliendo claramente perdedor el algoritmo de búsqueda local, donde los tiempos mayores llegan a tardar varios segundos mientras que COPKM el máximo tiempo son 196ms.

Gráficas

En las gráficas de la Figura 1 podemos ver una comparativa gráfica a lo largo del tiempo de como se comporta cada algoritmo con cada problema en cuanto al infeasibility. Se ha utilizado la semilla **1**.

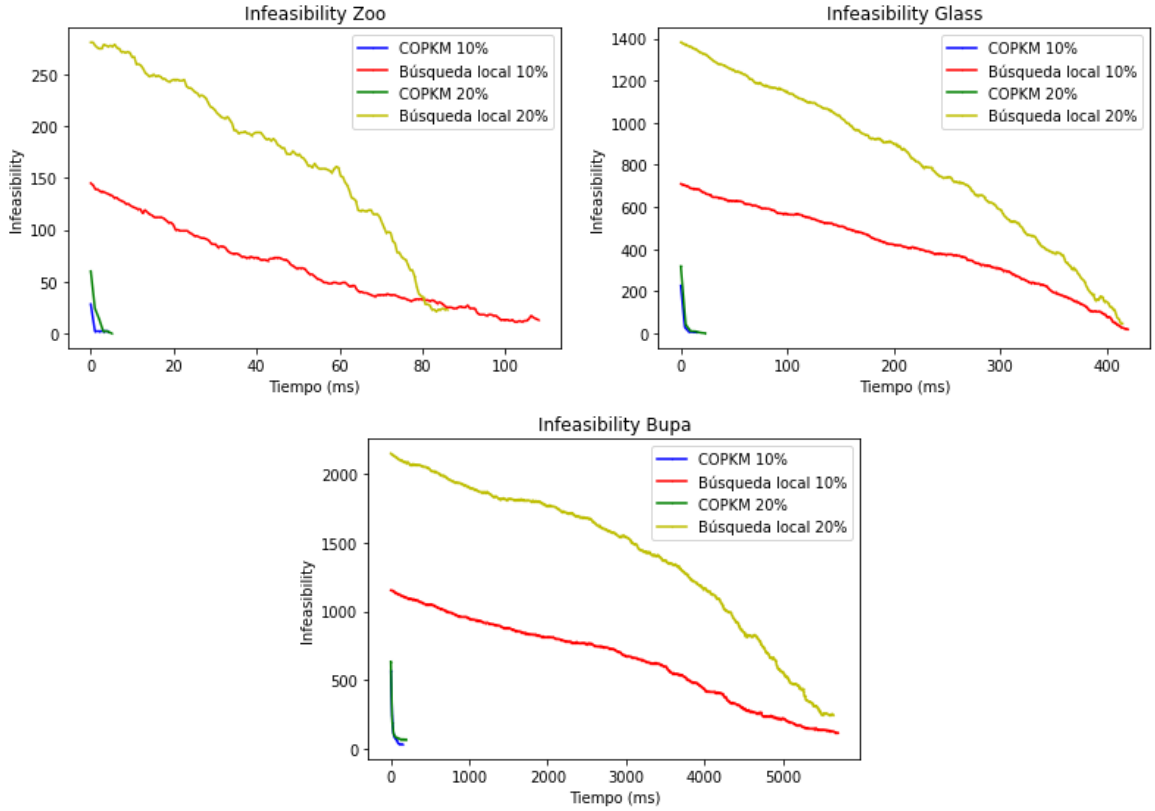

Figura 1: Gráficas de convergencia de infeasibility de cada algoritmo con cada base de datos

La misma comparativa está en las gráficas de la Figura 2 pero con respecto a la función objetivo. Se puede visualizar en estas gráficas tanto los tiempos: comparando cada problema (siendo el bupa el más costoso) y como el algoritmo COPKM es mucho más rápido. Sin embargo, se puede ver como el algoritmo BL, como hemos dicho antes, consigue un mejor desempeño en la función objetivo. Aunque en lo que nos importa primordialmente que son las restricciones, se puede ver en la Figura 1 como el algoritmo COPKM es claramente mejor.

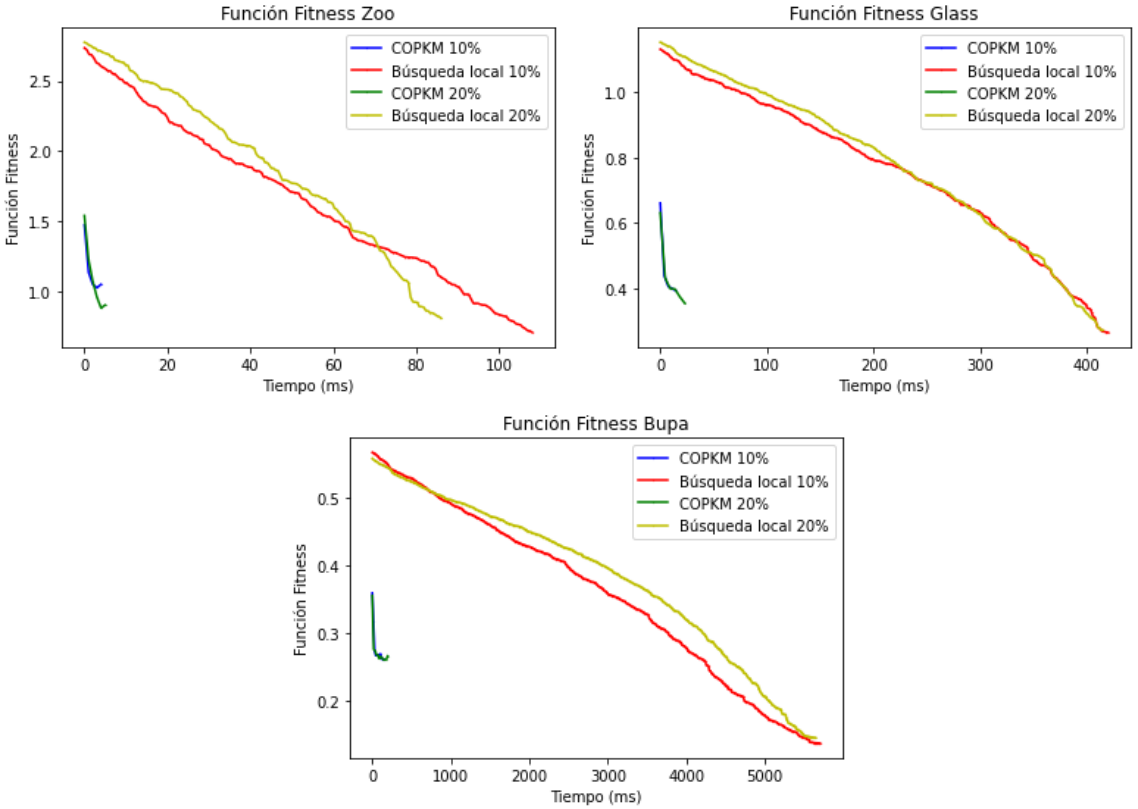

Figura 2: Gráficas de convergencia de la función objetivo de cada algoritmo con cada base de datos

11.1.4. Análisis extra: modificación de la función fitness para BL

Visto los resultados arrojados, la búsqueda local incumple muchas restricciones en comparación al algoritmo greedy, su infeasibility es muy alto. Es por ello que surge la duda de modificar la función fitness para que el peso que ejerce el infeasibility sobre la función fitness sea mayor. La modificación ha sido multiplicar hasta por cinco su peso, siendo la fórmula:

$$fitness(C) = desviacion(C) + 5\lambda infeasibility(C)$$

De esta forma se pretende que se atienda mucho más al infeasibility durante el proceso del algoritmo. Se ha considerado que quintuplicar el peso de las restricciones violadas sobre la función fitness es suficiente para que tenga algo de efecto sobre los resultados.

Este experimento tiene sentido puesto que si estamos valorando la calidad del algoritmo precisamente en primer lugar por el número de restricciones violadas, estas restricciones deberían de tenerse más en cuenta que el resto de parámetros como puede ser la desviación.

En las gráficas de la Figura 3 se puede ver como la búsqueda local modificada busca, sobre todo al inicio de las iteraciones, disminuir el infeasibility. Pero conforme avanzan los vecindarios se ve como la búsqueda local acaba cogiendo a esta modificada pero sin llegar a los mismos mínimos. En la Tabla 6 se pueden ver estos mínimos. Vemos como el infeasibility disminuye lo suficiente, ¿pero habremos perdido calidad en la distancia intra-clúster?

	Zoo		Glass		Bupa	
% Restricciones	10 %	20 %	10 %	20 %	10 %	20 %
BL	13	19	23	46	114	246
BL modificado	4	3	4	27	7	51

Tabla 6: Comparación de infeasibility entre Búsquedas Locales

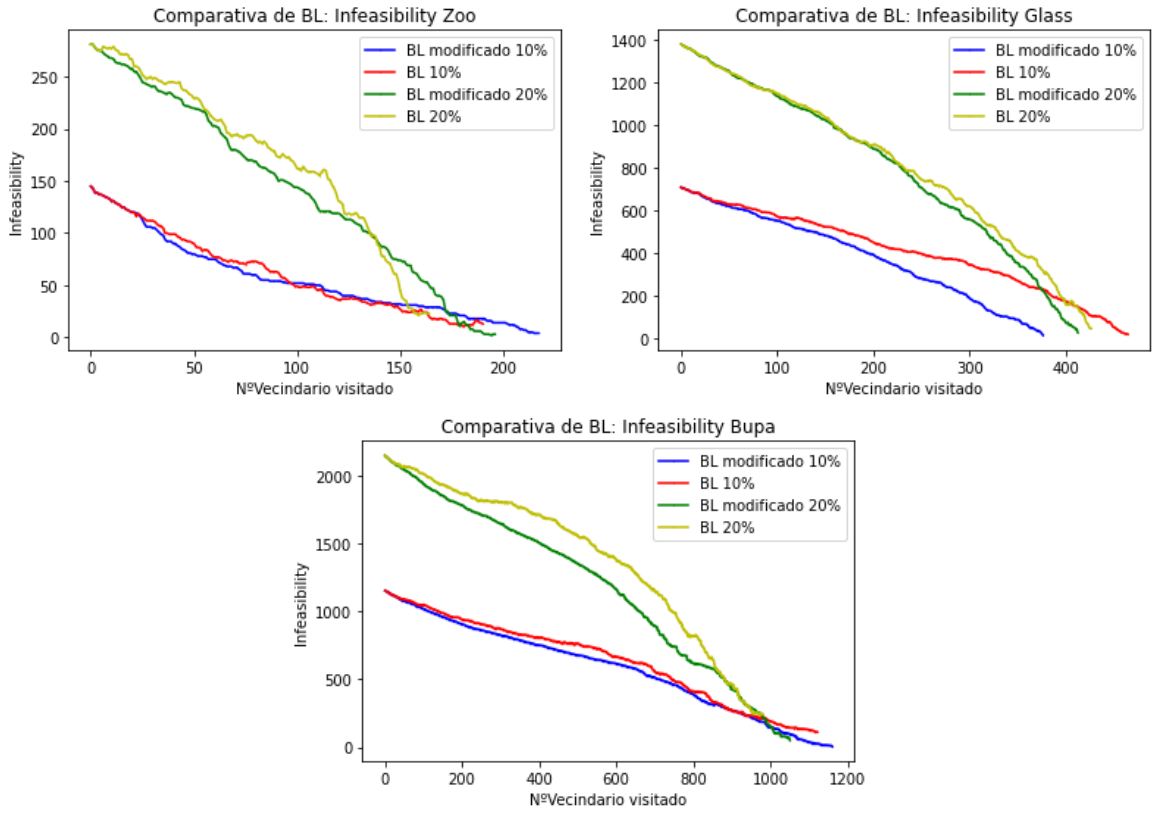

Figura 3: Gráficas comparativas entre las dos versiones de búsqueda local dependiendo de su función fitness.

Pasamos a ver las comparativas con respecto a la desviación. Podemos ver en las gráficas de la Figura 4 como evolucionan y en la Tabla 7 los valores finales obtenidos. Se ve en las gráficas como en todos los casos la modificación de la búsqueda local hace, como cabía esperar, que las soluciones sean peores con respecto a la desviación, se ve cómo las lineas azul y verde (que corresponden a la modificación) están siempre por encima de la amarilla y roja (la BL sin modificar). Por otra parte en la tabla podemos hacernos mejor una idea del resultado final viendo los valores obtenidos al final del proceso y pudiendo compararlo con los valores de referencia. Efectivamente, la desviación aumenta considerablemente, pero vemos que sigue siendo una desviación que mejora a la de referencia lo cual nos quiere decir que nuestra función objetivo a podido guiar bien nuestras soluciones.

	Zoo		Glass		Bupa	
% Restricciones	10 %	20 %	10 %	20 %	10 %	20 %
BL	0.611483	0.71821	0.247807	0.24962	0.106857	0.111261
BL modificado	0.875504	0.828259	0.312065	0.278855	0.164665	0.161748
Valores Referencia	0.904799856193481		0.364290281975566		0.229248049533093	

Tabla 7: Comparación de desviación entre Búsquedas Locales

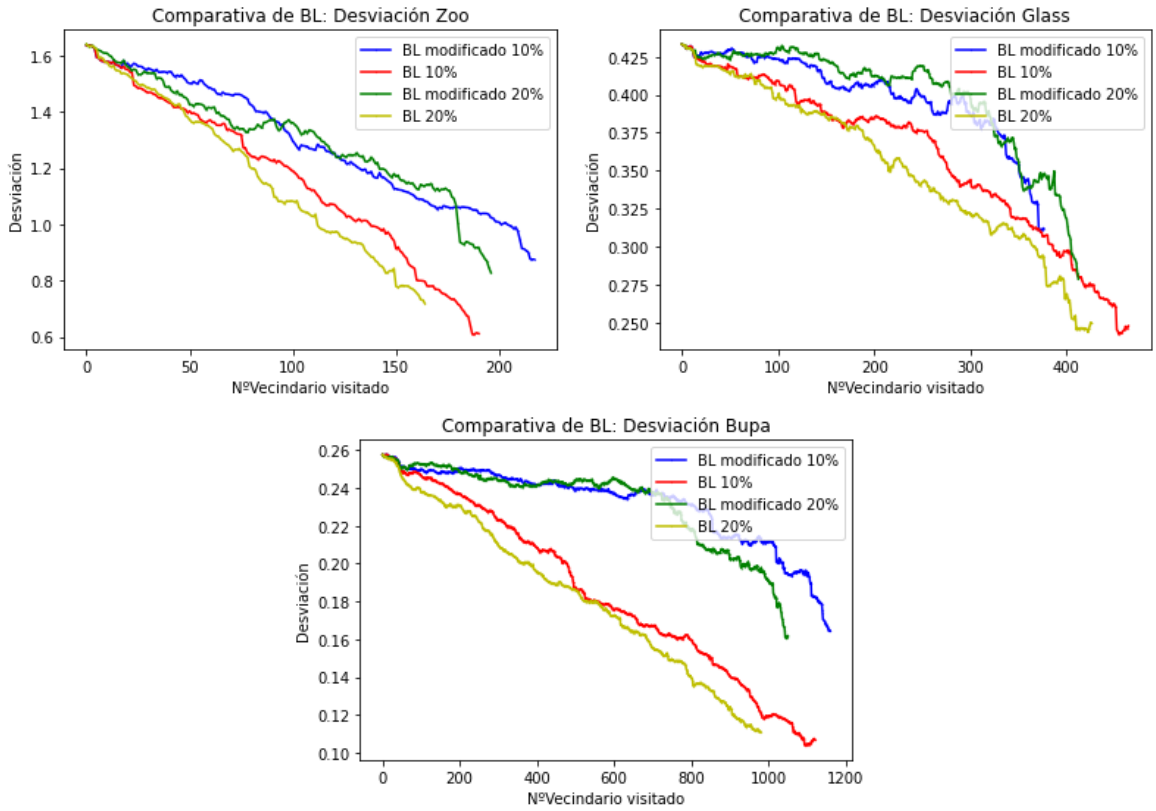

Figura 4: Gráficas comparativas entre las dos versiones de búsqueda local dependiendo de su función fitness.

Haciendo un resultado conjunto entre infeasibility y desviación obtenidos por esta nueva modificación podemos afirmar que las soluciones obtenidas mejoran a las que teníamos anteriormente con la búsqueda local sin modificar su función fitness. Es más, vemos como las soluciones se empiezan a asemejar a las soluciones del greedy COPKM en el sentido de que han disminuido las restricciones violadas y ha aumentado la desviación. Este aumento de la desviación viene provocado precisamente por la disminución del infeasibility puesto que al cumplir las restricciones la desviación de la partición se restringe más haciendo que sea mayor.

11.2. Géneticos y Meméticos

Al haber tantas combinaciones distintas para estos tipos de algoritmos, hay muchas tablas para analizar. Es por ello que se va a ir analizando poco a poco. En primer lugar se van a poner las 8 tablas correspondientes a los 4 algoritmos Genéticos con el 10 % y 20 % de restricciones. A continuación se hará un análisis entre estos resultados. Posteriormente, aparecerán las 6 tablas correspondientes a los 3 tipos de Meméticos con el 10 % y 20 % de restricciones, se hará un análisis de estos y para terminar se hará un análisis conjunto de Genéticos y Meméticos.

11.2.1. Genéticos

Todas las ejecuciones han sido el resultado de evaluar 100.000 veces la función objetivo. Para analizar los algoritmos genéticos vamos a comparar los comportamientos de los 4 tipos distintos que hay, para posteriormente dar un análisis acerca de como de bien funcionan estos algoritmos en nuestro problema.

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	12	0.631091	0.722022	2112	50	0.19822	0.247422	4224	522	0.159134	0.299024	7424
Ejecución 2	30	0.699123	0.926452	2107	48	0.210044	0.257278	4260	563	0.161696	0.312573	7546
Ejecución 3	12	0.615149	0.70608	2107	56	0.188551	0.243657	4203	481	0.158837	0.28774	7532
Ejecución 4	14	0.598474	0.704561	2093	79	0.260234	0.337973	4201	445	0.15738	0.276635	7418
Ejecución 5	10	0.59997	0.675746	2094	30	0.248963	0.278485	4240	544	0.152633	0.298419	7454
Media	15.6	0.628761	0.746972	2102.6	52.6	0.221202	0.272963	4225.6	511	0.157936	0.294878	7474.8

Tabla 8: Resultados obtenidos por el algoritmo AGG-Uniforme con 10 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	16	0.767231	0.83184	2218	89	0.238748	0.285082	4825	1184	0.162087	0.327185	9297
Ejecución 2	25	0.71611	0.817063	2208	169	0.194399	0.282382	4834	1047	0.149059	0.295054	9343
Ejecución 3	19	0.733622	0.810346	2203	51	0.247812	0.274363	4834	944	0.146539	0.278172	9195
Ejecución 4	32	0.714506	0.843725	2204	62	0.246278	0.278556	4772	954	0.154114	0.28714	9604
Ejecución 5	18	0.696072	0.768758	2208	34	0.245765	0.263466	4812	1009	0.150157	0.290853	9295
Media	22	0.725509	0.814346	2208.2	81	0.234601	0.27677	4815.4	1027.6	0.152391	0.295681	9346.8

Tabla 9: Resultados obtenidos por el algoritmo AGG-Uniforme con 20 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	11	0.620167	0.703521	2100	15	0.24766	0.262421	3947	600	0.15864	0.319433	6921
Ejecución 2	10	0.619472	0.695249	1976	35	0.238409	0.27285	3919	564	0.147225	0.298371	7074
Ejecución 3	10	0.6274	0.703176	2005	66	0.200839	0.265786	3942	653	0.155503	0.330499	6958
Ejecución 4	10	0.616167	0.691944	1999	108	0.231707	0.337984	3931	631	0.159665	0.328766	6940
Ejecución 5	20	0.773729	0.925282	1993	18	0.24577	0.263483	3936	632	0.159803	0.329171	6881
Media	12.2	0.651387	0.743834	2014.6	48.4	0.232877	0.280505	3935	616	0.156167	0.321248	6954.8

Tabla 10: Resultados obtenidos por el algoritmo AGG-Segmento Fijo con 10 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	10	0.733622	0.774003	2143	37	0.250358	0.269621	4516	1199	0.159097	0.326286	8733
Ejecución 2	26	0.712829	0.817819	2120	67	0.245506	0.280387	4539	781	0.158548	0.267452	9769
Ejecución 3	20	0.707524	0.788286	2110	75	0.254305	0.293351	4546	859	0.149985	0.269765	8693
Ejecución 4	17	0.712827	0.781474	2084	110	0.207066	0.264333	4516	1067	0.149513	0.298297	8700
Ejecución 5	24	0.707654	0.804567	2102	57	0.248077	0.277751	4536	1076	0.152142	0.30218	8697
Media	19.4	0.714891	0.79323	2111.8	69.2	0.241062	0.277089	4530.6	996.4	0.153857	0.292796	8918.4

Tabla 11: Resultados obtenidos por el algoritmo AGG-Segmento Fijo con 20 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	6	0.765171	0.810637	2104	50	0.2127	0.261902	4352	692	0.153851	0.339299	7439
Ejecución 2	9	0.614028	0.682227	2095	101	0.254053	0.353441	4213	611	0.160835	0.324576	7524
Ejecución 3	18	0.627571	0.763968	2080	108	0.260193	0.366469	4198	642	0.15597	0.328018	7472
Ejecución 4	12	0.621508	0.71244	2096	116	0.266497	0.380645	4211	477	0.160527	0.288358	7437
Ejecución 5	7	0.725634	0.778678	2062	58	0.216854	0.273928	4248	624	0.165148	0.332372	7418
Media	10.4	0.670782	0.74959	2087.4	86.6	0.242059	0.327277	4244.4	609.2	0.159266	0.322525	7458

Tabla 12: Resultados obtenidos por el algoritmo AGE-Uniforme con 10 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	17	0.718965	0.787613	2207	39	0.246378	0.266682	4970	957	0.146	0.279445	9291
Ejecución 2	28	0.714409	0.827475	2196	53	0.24752	0.275112	4775	1223	0.152128	0.322664	9201
Ejecución 3	18	0.710859	0.783544	2180	151	0.20157	0.280182	4872	1005	0.155988	0.296126	9180
Ejecución 4	29	0.664317	0.781422	2179	35	0.247706	0.265927	4760	1047	0.158007	0.304002	9263
Ejecución 5	12	0.733253	0.78171	2185	65	0.245187	0.279027	4773	1106	0.159034	0.313256	9298
Media	20.8	0.708361	0.792353	2189.4	68.6	0.237672	0.273386	4830	1067.6	0.154231	0.303099	9246.6

Tabla 13: Resultados obtenidos por el algoritmo AGE-Uniforme con 20 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	12	0.623931	0.714863	1983	43	0.251173	0.293487	4005	475	0.160689	0.287983	6925
Ejecución 2	21	0.751238	0.910368	1985	26	0.245132	0.270718	3940	493	0.157968	0.290086	7187
Ejecución 3	14	0.596779	0.702866	1969	56	0.223614	0.278721	3937	546	0.150863	0.297185	6998
Ejecución 4	13	0.713528	0.812037	1976	57	0.190531	0.246621	3906	555	0.163047	0.311781	6946
Ejecución 5	13	0.609071	0.70758	1960	65	0.284017	0.34798	3951	699	0.173347	0.360671	6854
Media	14.6	0.658909	0.769543	1974.6	49.4	0.238893	0.287505	3947.8	553.6	0.161183	0.309541	6982

Tabla 14: Resultados obtenidos por el algoritmo AGE-Segmento Fijo con 10 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	17	0.720538	0.789185	2077	56	0.247567	0.276721	4547	1214	0.170364	0.339646	8679
Ejecución 2	30	0.750522	0.871664	2086	48	0.246178	0.271167	4485	1101	0.152629	0.306154	8759
Ejecución 3	15	0.728162	0.788733	2068	55	0.247486	0.276119	4512	1068	0.1551	0.304023	8656
Ejecución 4	36	0.711596	0.856967	2077	97	0.210893	0.261392	4495	1117	0.155973	0.311729	8673
Ejecución 5	31	0.718933	0.844114	2083	48	0.248389	0.273379	4507	1104	0.154323	0.308266	8728
Media	25.8	0.72595	0.830133	2078.2	60.8	0.240102	0.271756	4509.2	1120.8	0.157678	0.313964	8699

Tabla 15: Resultados obtenidos por el algoritmo AGE-Segmento Fijo con 20 % de restricciones

Para comparar los 4 tipos, vamos a hacer un estudio a partir de estas tablas sobre cómo influye el modelo (Generacional o Estacionario) y cómo influye el tipo de cruce (Uniforme y Segmento Fijo). Para ello cogeremos las medias de las tablas de arriba y haremos un estudio en donde filtraremos por tipo de modelo y tipo de cruce.

Tipo de modelo: Generacional y Estacionario

En las tablas comprendidas entre la Tabla 16 y la Tabla 19 se encuentran las correspondientes al análisis del modelo. Estas tablas se han obtenido cogiendo la fila de medias de las tablas completas de las ejecuciones, haciendo la media entre los dos tipos de cruce distintos. Estas nuevas medias las tomaremos como valores de referencia para comparar ambos modelos.

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Media AGGUN	15.6	0.628761	0.746972	2102.6	52.6	0.221202	0.272963	4225.6	511	0.157936	0.294878	7474.8
Media AGGSF	12.2	0.651387	0.743834	2014.6	48.4	0.232877	0.280505	3935	616	0.156167	0.321248	6954.8
Media	13.9	0.640074	0.745403	2058.6	50.5	0.22704	0.2767	4080.3	563.5	0.15705	0.3081	7214.8

Tabla 16: Resultados obtenidos por el algoritmo AG Generacional con 10 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Media AGEUN	10.4	0.670782	0.74959	2087.4	86.6	0.242059	0.327277	4244.4	609.2	0.159266	0.322525	7458
Media AGESF	14.6	0.658909	0.769543	1974.6	49.4	0.238893	0.287505	3947.8	553.6	0.161183	0.309541	6982
Media	12.5	0.66485	0.7596	2031	68	0.240476	0.3074	4096.1	581.4	0.1602245	0.316	7220

Tabla 17: Resultados obtenidos por el algoritmo AG Estacionario con 10 % de restricciones

Comparando la Tabla 16 con la Tabla 17, las tablas del 10 % de restricciones, vemos que en términos de infeasibility no obtenemos unos mejores resultados claros con un modelo u otro. En cuanto a la distancia intra-clúster obtenemos ligeramente mejores resultados con el Genético Generacional, siendo menor en él la distancia en todos los problemas. Esto implica que cuando hablemos de la función objetivo, el algoritmo Generacional también va a obtener unos, ligeramente, mejores resultados. El tiempo de ejecución en ambos es similar.

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Media AGGUN	22	0.725509	0.814346	2208.2	81	0.234601	0.27677	4815.4	1027.6	0.152391	0.295681	9346.8
Media AGGSF	19.4	0.714891	0.79323	2111.8	69.2	0.241062	0.277089	4530.6	996.4	0.153857	0.292796	8918.4
Media	21.2	0.7202	0.8038	2160	75.1	0.2378	0.2769	4673	1012	0.1531	0.2942385	9132.6

Tabla 18: Resultados obtenidos por el algoritmo AG Generacional con 20 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Media AGEUN	20.8	0.708361	0.792353	2189.4	68.6	0.237672	0.273386	4830	1067.6	0.154231	0.303099	9246.6
Media AGESF	25.8	0.72595	0.830133	2078.2	60.8	0.240102	0.271756	4509.2	1120.8	0.157678	0.313964	8699
Media	23.3	0.71716	0.81124	2133.8	64.7	0.239	0.272571	4669.6	1094.2	0.156	0.3085	8972.8

Tabla 19: Resultados obtenidos por el algoritmo AG Estacionario con 20 % de restricciones

Pasamos a los problemas con el 20 % de restricciones, cuyas tablas correspondientes son: la Tabla 18 y la Tabla 19. Empezamos de nuevo fijándonos en el infeasibility, en donde pasa como con el 10 %, no llega a haber un modelo que mejore al otro en todos los problemas. Pasamos pues, a comentar la distancia intra-clúster: como en el infeasibility, tampoco hay un modelo que mejore al otro en todos los problemas. Consecuentemente, nuestra función objetivo tampoco podrá determinar que modelo tiene mejor comportamiento. Nuestra última baza para comparar es el tiempo, en donde sale ganando por la mínima el modelo Estacionario.

Habiendo hecho un desglose de los resultados de los problemas con el 10 % y 20 % de restricciones, hemos obtenido que para el 10 % el modelo Generacional mejoraba ligeramente al Estacionario puesto que conseguía una distancia intra-clúster algo menor, lo que repercutía directamente en que la función objetivo, en media también fuese mejor. Por otro lado, para el 20 % de restricciones se ha obtenido que el modelo Estacionario es mejor. Sin embargo, este último resultado se ha obtenido recurriendo al tiempo de ejecución (última medida a la que observar para determinar que algoritmo es mejor en caso de que no se pueda decidir ni por el infeasibility ni por la función objetivo). Así, por haberse elegido en el problema del 10 % el modelo Generacional por la medida de la función objetivo, concluiremos que el modelo Generacional se comporta ligeramente mejor que el modelo Estacionario para estos problemas.

Evidentemente, el modelo Estacionario necesita de más generaciones para llegar a estos resultados, pero esto es de esperar puesto que se hacen muchas menos evaluaciones de la función objetivo por generación que en el modelo Generacional. Como conclusión analítica de estos resultados, basta decir que la exploración de ambos modelos es prácticamente igual de buena, ambas filosofías consiguen llegar a soluciones similares.

Tipo de cruce: Uniforme y Segmento Fijo

En las tablas comprendidas entre la Tabla 20 y la Tabla 23 se encuentran las correspondientes al análisis del tipo de cruce. Estas tablas se han obtenido cogiendo la fila de medias de las tablas completas de las ejecuciones, haciendo la media entre los dos tipos de modelos distintos. Estas nuevas medias las tomaremos como valores de referencia para comparar ambos modelos.

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Media AGGUN	15.6	0.628761	0.746972	2102.6	52.6	0.221202	0.272963	4225.6	511	0.157936	0.294878	7474.8
Media AGEUN	10.4	0.670782	0.74959	2087.4	86.6	0.242059	0.327277	4244.4	609.2	0.159266	0.322525	7458
Media	13	0.64798	0.7483	2095	69.6	0.2313	0.3	4235	560.1	0.1586	0.3087	7466.4

Tabla 20: Resultados obtenidos con el cruce Uniforme con 10 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Media AGGSF	12.2	0.651387	0.743834	2014.6	48.4	0.232877	0.280505	3935	616	0.156167	0.321248	6954.8
Media AGESF	14.6	0.658909	0.769543	1974.6	49.4	0.238893	0.287505	3947.8	553.6	0.161183	0.309541	6982
Media	13.4	0.655	0.7567	1994.6	48.9	0.235885	0.284	3941.4	584.8	0.1587	0.3154	6968.4

Tabla 21: Resultados obtenidos con el cruce Segmento Fijo con 10 % de restricciones

Empezaremos la comparación fijándonos en los problemas del 10 % de restricciones. Como de costumbre nos fijaremos en primer lugar en la tasa de infactibilidad: el cruce uniforme sale bien parado en Zoo y Bupa, mientras que el cruce de Segmento Fijo sale triunfante en el Glass. En la distancia intra-clúster sale vencedor el cruce Uniforme. Sin embargo, no es lo suficientemente notoria la mejora en la distancia intra-clúster ya que no provoca que los valores de la función objetivo sea siempre mejor que la del cruce de Segmento Fijo. El Segmento Fijo, por otra parte, es en media más rápido. En resumen, el cruce Uniforme mejora (aunque no lo suficiente) en la distancia intra-clúster y el cruce Segmento Fijo mejora en tiempo. Viendo que son mejoras poco notorias, es difícil concluir que una opción es mejor que otra, ya que las diferencias son mínimas.

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Media AGGUN	22	0.725509	0.814346	2208.2	81	0.234601	0.27677	4815.4	1027.6	0.152391	0.295681	9346.8
Media AGEUN	20.8	0.708361	0.792353	2189.4	68.6	0.237672	0.273386	4830	1067.6	0.154231	0.303099	9246.6
Media	21.4	0.7169	0.8033	2198.8	74.8	0.2361	0.275	4822.7	1047.6	0.153311	0.2994	9296.7

Tabla 22: Resultados obtenidos con el cruce Uniforme con 20 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Media AGGSF	19.4	0.714891	0.79323	2111.8	69.2	0.241062	0.277089	4530.6	996.4	0.153857	0.292796	8918.4
Media AGESF	25.8	0.72595	0.830133	2078.2	60.8	0.240102	0.271756	4509.2	1120.8	0.157678	0.313964	8699
Media	22.6	0.7204	0.8117	2095	65	0.24102	0.27442	4519.9	1058.6	0.1558	0.3034	8808.7

Tabla 23: Resultados obtenidos con el cruce Segmento Fijo con 20 % de restricciones

Comparando ahora con el problema del 20 % tenemos exactamente el mismo escenario que en el 10 %.

Podemos concluir que si el tiempo es importante, nos podríamos decantar por usar el cruce de Segmento Fijo, también podríamos decantarnos por el cruce Uniforme sin obtener mucha diferencia. A pesar de ello, ambas opciones son opciones factibles para el problema, arrojando soluciones similares.

Conclusión de los 4 tipos de Genético

En estos dos últimos apartados se ha concluido que el modelo Generacional mejora (aunque sea ligeramente) al modelo Estacionario. Por otro lado, en cuanto al cruce, se ha dicho que es indiferente el tipo de cruce que hagamos, no afectará de forma sustancial a los resultados. Estas dos conclusiones nos hablan de que los 4 algoritmos distintos tienen un desempeño similar, que era lo que se podía intuir mirando las tablas y viendo que había valores muy parecidos en todas.

Como para los Meméticos se ha tenido que escoger uno de los 4, se ha decidido que el modelo Generacional tiene que ser el que se use y en cuanto al tipo de cruce, como hay que hacer varios experimentos, se ha elegido el Segmento Fijo ya que el tiempo de ejecución es ligeramente menor.

En resumen, se ha decidido que el mejor modelo es el Generacional con Segmento Fijo.

11.2.2. Meméticos

Como pasaba con los genéticos, todas las ejecuciones han sido el resultado de evaluar 100.000 veces la función objetivo. Para analizar los algoritmos Meméticos vamos a comparar directamente los 3 tipos que se nos proponen. Los resultados de las ejecuciones están recogidos en las tablas que van desde la Tabla 24 a la Tabla 29.

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	29	1.01595	1.2357	1794	334	0.274395	0.603065	3885	987	0.19381	0.458314	8257
Ejecución 2	21	0.786359	0.945489	1794	193	0.267737	0.457657	3551	983	0.205432	0.468864	6340
Ejecución 3	19	0.706224	0.850199	1838	192	0.335413	0.524348	3686	913	0.218742	0.463416	8160
Ejecución 4	16	0.625526	0.746768	1832	446	0.300099	0.738981	3642	966	0.204029	0.462905	8152
Ejecución 5	17	0.882565	1.01138	1806	244	0.320395	0.560501	3751	925	0.206869	0.454758	8387
Media	20.4	0.803324	0.957908	1812.8	281.8	0.299608	0.57691	3703	954.8	0.205776	0.461651	7859.2

Tabla 24: Resultados obtenidos por el algoritmo Memético(10,1.0) con AGG-SF con un 10 % de restricciones

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	23	0.74427	0.837146	1994	452	0.297665	0.53298	4684	1964	0.188602	0.462464	10813
Ejecución 2	44	0.704835	0.882511	1935	634	0.273904	0.603971	4460	1908	0.192483	0.458537	11422
Ejecución 3	51	0.628101	0.834043	1976	481	0.283036	0.533449	4379	1949	0.202219	0.473989	10784
Ejecución 4	18	0.755186	0.827872	1912	520	0.317415	0.588132	4531	1640	0.20352	0.432204	10721
Ejecución 5	44	0.634795	0.81247	2013	238	0.288607	0.412512	4335	1826	0.206337	0.460956	10595
Media	36	0.693437	0.838808	1966	465	0.292126	0.534209	4477.8	1857.4	0.198632	0.45763	10867

Tabla 25: Resultados obtenidos por el algoritmo Memético(10,1.0) con AGG-SF con un 20 % de restricciones

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	14	0.649109	0.755196	1891	205	0.245575	0.447303	4623	383	0.165598	0.268238	6340
Ejecución 2	9	0.611641	0.67984	1894	17	0.245144	0.261872	3631	386	0.170459	0.273902	6452
Ejecución 3	26	0.752247	0.949266	1907	68	0.212768	0.279683	3652	635	0.156888	0.32706	6345
Ejecución 4	25	0.775518	0.964959	1922	95	0.245359	0.338843	3670	414	0.149548	0.260495	6378
Ejecución 5	9	0.614797	0.682996	1898	118	0.261029	0.377146	3655	380	0.175658	0.277494	6312
Media	16.6	0.680662	0.806451	1902.4	100.6	0.241975	0.34097	3846.2	439.6	0.16363	0.281438	6365.4

Tabla 26: Resultados obtenidos por el algoritmo Memético(10,0.1) con AGG-SF con un 10 % de restricciones

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	31	0.714506	0.839687	2021	147	0.227001	0.303531	4368	675	0.14603	0.240153	8148
Ejecución 2	21	0.750378	0.835178	2001	86	0.25622	0.300992	4248	790	0.180215	0.290373	8166
Ejecución 3	28	0.702769	0.815835	2010	101	0.250002	0.302584	4353	1302	0.138546	0.320098	8498
Ejecución 4	18	0.708261	0.780946	2007	205	0.226161	0.332886	4324	819	0.172786	0.286988	8195
Ejecución 5	12	0.750452	0.798909	2016	71	0.24154	0.278503	4269	803	0.171723	0.283694	8154
Media	22	0.725273	0.814111	2011	122	0.240185	0.303699	4312.4	877.8	0.16186	0.284261	8232.2

Tabla 27: Resultados obtenidos por el algoritmo Memético(10,0.1) con AGG-SF con un 20 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	34	0.867007	1.12465	1919	52	0.188003	0.239173	3616	422	0.180311	0.293402	6311
Ejecución 2	10	0.614358	0.690134	1887	134	0.222597	0.354459	3726	368	0.173965	0.272584	6373
Ejecución 3	11	0.610131	0.693484	1908	54	0.252259	0.305397	3660	411	0.152346	0.262489	6452
Ejecución 4	11	0.638013	0.721367	1908	121	0.219243	0.338312	3662	390	0.169285	0.273801	6357
Ejecución 5	6	0.725128	0.770594	1888	249	0.222427	0.467453	3729	411	0.142744	0.252887	6360
Media	14.4	0.690927	0.800045	1902	122	0.220906	0.340959	3678.6	400.4	0.16373	0.271033	6370.6

Tabla 28: Resultados obtenidos por el algoritmo Memético(10,0.1mej) con AGG-SF con un 10 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	33	0.721275	0.854532	2014	134	0.211153	0.280915	4264	1314	0.151604	0.33483	8450
Ejecución 2	45	0.585136	0.76685	2025	85	0.241982	0.286234	4306	829	0.132107	0.247704	8284
Ejecución 3	15	0.733565	0.794137	1997	130	0.228398	0.296077	4295	844	0.149644	0.267332	8217
Ejecución 4	42	0.604234	0.773834	2013	315	0.250363	0.414355	4305	780	0.189888	0.298652	8127
Ejecución 5	21	0.730876	0.815676	2018	95	0.256586	0.306044	4271	763	0.183324	0.289717	8216
Media	31.2	0.675017	0.801005	2013.4	151.8	0.237696	0.316725	4288.2	906	0.161313	0.287647	8258.8

Tabla 29: Resultados obtenidos por el algoritmo Memético(10,0.1mej) con AGG-SF con un 20 % de restricciones

Empezando el análisis vemos de primeras como el algoritmo Memético(10,1.0) es claramente el perdedor en esta batalla: el infeasibility medio de los problemas, tanto para el 10 % de restricciones como para el 20 % de restricciones, es, de lejos, el peor de los 3 tipos de Memético. Al ser el infeasibility nuestra medida primordial de comparación, clasificamos este Memético como el peor de los 3. Podemos pensar que esto es así porque explota mucho y explora poco: aplica Búsqueda Local Suave a todos los cromosomas de la población, agotando así evaluaciones de la función objetivo.

Comparando ahora entre Memético(10,0.1) y Memético(10,0.1mej), en el infeasibility no predomina ninguno de los dos en el problema del 10 % de restricciones, pero sí en el del 20 %, quedando mejor el Memético(10,0.1). Sin embargo, la función objetivo toma valores similares en ambas, incluso ganando en la mayoría de veces el Memético(10,0.1mej). Esto tiene sentido ya que al violar más restricciones se permite tener una solución con una distancia intra-clúster menor permitiendo mejorar la función objetivo. Como la infactibilidad es el punto importante para ver el desempeño, vamos a concluir, que el algoritmo Memético(10,0.1) tiene un desempeño levemente mejor.

11.2.3. Comparación entre Genéticos y Meméticos

Llegados a este punto ya hemos visto como se desenvuelven los algoritmos Genéticos y Meméticos. Recordando, hemos visto que los algoritmos Genéticos tenían un desempeño similar, imponiéndose por la mínima el modelo Generacional e imponiéndose en términos de velocidad de ejecución el cruce Segmento Fijo. Por otro lado, de los algoritmos Meméticos, ha destacado el mal desempeño del Memético(10,1.0) quedando una pelea muy igualada entre Memético(10,0.1) y Memético(10,0.1mej), en donde ha salido victorioso, también por la mínima, Memético(10,0.1).

Procedemos ahora a hacer una comparación general entre los Meméticos y los Genéticos. Esta comparación no va a variar mucho dependiendo de qué tipo de Genético hablemos puesto que se ha visto que se arrojan resultados muy parecidos. Con los Meméticos no pasará igual, pues habrá que destacar el caso de Memético(10,1.0) por su mal desempeño.

Y empezamos comentando ya el Memético(10,0.1) (Tablas 24 y 25) es claramente peor que cualquier Genético, la infactibilidad arrojada por este algoritmo es realmente mala. Pasamos

ahora a comentar el resto, en donde, para que sea más fácil su estudio, se ha decidido centrarnos en un representante de cada debido a que no van a variar mucho los resultados. El representante será el que mejor desempeño ha tenido: AGG con cruce Segmento Fijo (Tablas 10 y 11) y Memético(10,0.1)(Tablas 26 y 27):

Vemos como el algoritmo Genético mejora en términos de infeasibility al Memético en Zoo y Glass. Sin embargo, en Bupa, el Memético consigue mejor infeasibility. Podemos intuir que esto es debido a que el espacio de soluciones de Bupa es mucho más complejo debido al número de instancias y sobre todo al número de clases. Esto es algo importante porque tenemos que recordar que los algoritmos Genéticos son buenos exploradores y malos explotadores, esto puede significar que nuestro algoritmo genético, en espacios más simples explore bien y consiga llegar entornos donde el mínimo local es suficientemente bueno. Por otro lado, en espacios complejos, es más complicado encontrar ese entorno, es aquí donde, si encuentras un entorno que es medianamente bueno, lo que puede hacer mejorar tu algoritmo es explotar ese entorno para hallar el mínimo y es por ello por lo que podemos intuir que nuestro algoritmo Memético obtiene mejores resultados en este problema (Bupa). El mismo razonamiento es aplicable a los valores de la función objetivo ya que las distancias intra-clúster son parecidas. En tiempo de ejecución, el Memético consigue ser un poco más rápido que el Genético.

En las Figuras 5, 6 y 7 podemos ver la evolución de los algoritmos AGGSF, Memético(10,1.0) y Memético(10,0.1) con el paso de las evaluaciones. Los valores representados es el valor de la función objetivo del mejor cromosoma de la población en cada generación. Es muy interesante observar la explotación de los algoritmos meméticos: mientras que el AGGSF mantiene una curvatura suave, los meméticos tienen "escalones" debido al proceso de explotación. Estos "escalones" son muy notorios sobre todo en el Memético(10,1.0) y en bupa como cabía esperar, puesto que el proceso de explotación es mayor (por el tipo de algoritmo) y se le dedican más iteraciones (por el tamaño del conjunto de datos). En estas gráficas se puede observar como, a medida que aumenta la complejidad el Memético(10,0.1) se va acercando al AGGSF hasta que en el conjunto Bupa consigue tener mejor desempeño (como habíamos comentado).

Como conclusión podemos decir que, entre el AGGSF y Memético(10,0.1) no podemos destacar a ningun algoritmo por encima del otro, podemos decir que tienen mejor desempeño uno sobre el otro dependiendo del problema. Así, se recomienda estudiar antes el problema al que nos enfrentamos y valorar como de necesaria es hacer una explotación en mitad de la exploración. Por lo general, suele venir bien hacer la explotación, pero hay que recordar que no estamos usando un algoritmo de búsqueda local como el que hemos analizado en la sección **3**, se trata un algoritmo de búsqueda local suave, que hace una explotación no muy buena pero que es muy rápida.

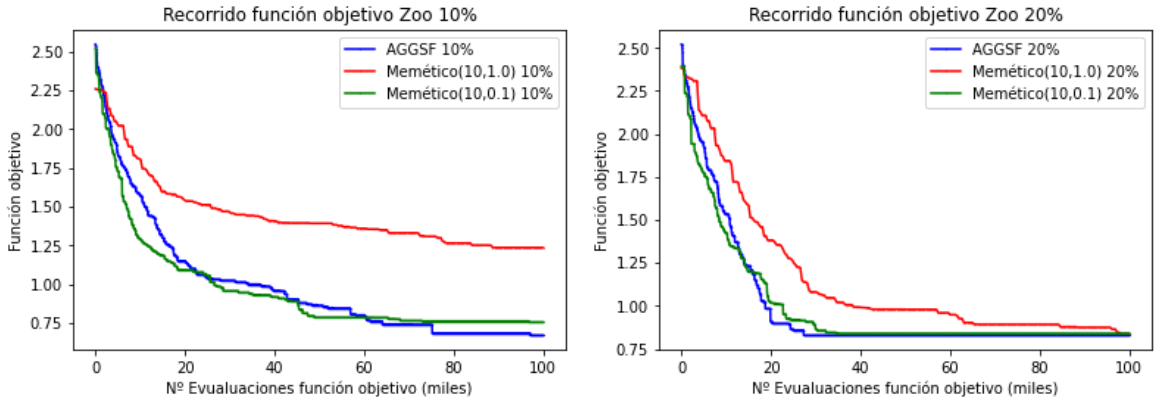

Figura 5: Gráficas de convergencia del valor de la función objetivo de algoritmos Genéticos y Meméticos en Zoo

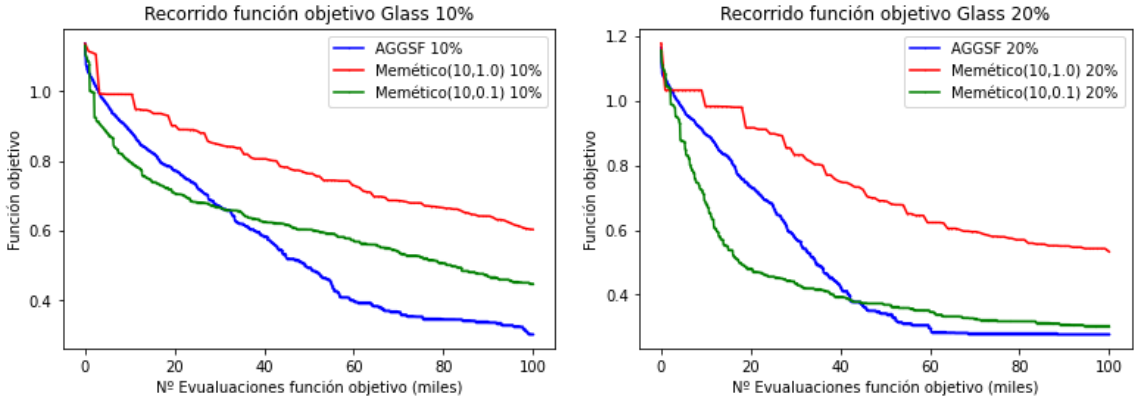

Figura 6: Gráficas de convergencia del valor de la función objetivo de algoritmos Genéticos y Meméticos en Glass

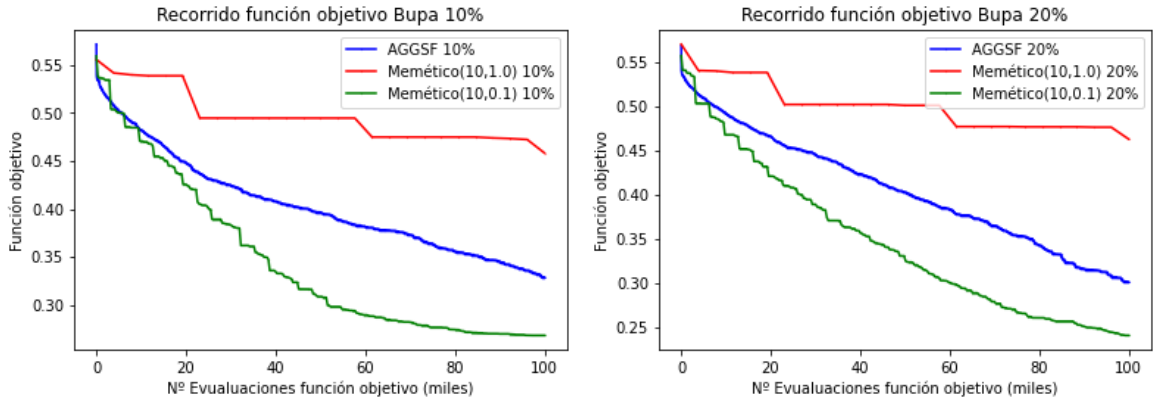

Figura 7: Gráficas de convergencia del valor de la función objetivo de algoritmos Genéticos y Meméticos en Bupa

11.2.4. Análisis Extra: Aumento de probabilidad de mutación

Vamos a probar en este experimento aumentar la probabilidad de mutación para ver como se comportan las soluciones. Se ha decidido multiplicar por 5 el número de genes a mutar, esto es: 25 genes en el Generacional y 1 gen cada generación en el Estacionario. En las Tablas que van desde la Tabla 30 a la Tabla 37 se ven los resultados de las ejecuciones con este aumento de mutación de genes

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	11	0.607856	0.69121	2065	34	0.236413	0.269871	3934	638	0.155373	0.326349	7032
Ejecución 2	6	0.61608	0.661546	1988	23	0.244292	0.266925	3972	396	0.156922	0.263045	7126
Ejecución 3	21	0.773671	0.932801	2060	48	0.194873	0.242106	3992	344	0.156399	0.248587	7031
Ejecución 4	20	0.753123	0.904675	2004	58	0.187803	0.244877	3970	574	0.154829	0.308654	7130
Ejecución 5	9	0.600582	0.668781	2266	32	0.223769	0.255258	3970	491	0.152626	0.284208	7021
Media	13.4	0.670262	0.771803	2076.6	39	0.21743	0.255808	3967.6	488.6	0.15523	0.286169	7068

Tabla 30: Ejecuciones con mayor mutación de AGGUN 10 %

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	22	0.710354	0.799192	2158	34	0.249216	0.266917	4541	644	0.144668	0.234468	8675
Ejecución 2	13	0.726902	0.779397	2095	112	0.209653	0.267961	4614	627	0.158596	0.246026	8701
Ejecución 3	16	0.718814	0.783423	2115	47	0.251407	0.275876	4597	608	0.148833	0.233613	8714
Ejecución 4	22	0.657246	0.746084	2109	60	0.244301	0.275538	4602	595	0.148984	0.231951	8700
Ejecución 5	23	0.716594	0.80947	2107	28	0.244371	0.258948	4528	561	0.139317	0.217543	8711
Media	19.2	0.705982	0.783513	2116.8	56.2	0.23979	0.269048	4576.4	607	0.148079	0.23272	8700.2

Tabla 31: Ejecuciones con mayor mutación de AGGUN 20 %

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	10	0.60952	0.685297	1921	45	0.189666	0.233948	4178	513	0.15686	0.294338	6672
Ejecución 2	7	0.616808	0.669852	1919	56	0.206531	0.261637	3813	312	0.144156	0.227768	6715
Ejecución 3	14	0.599882	0.705969	1920	38	0.249587	0.286981	3830	415	0.171744	0.282959	6716
Ejecución 4	10	0.609054	0.684831	1915	54	0.212163	0.265301	3779	356	0.161737	0.25714	6809
Ejecución 5	7	0.589884	0.642927	1933	108	0.237067	0.343343	3815	559	0.16146	0.311265	6735
Media	9.6	0.60503	0.677775	1921.6	60.2	0.219003	0.278242	3883	431	0.159191	0.274694	6729.4

Tabla 32: Ejecuciones con mayor mutación de AGGSF 10 %

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	28	0.663863	0.776929	2044	119	0.208796	0.270748	4385	595	0.141312	0.224279	8332
Ejecución 2	16	0.712238	0.776847	2025	126	0.20183	0.267427	4471	751	0.157178	0.261898	8611
Ejecución 3	25	0.707213	0.808165	2044	109	0.208794	0.26554	4433	901	0.159874	0.28551	8367
Ejecución 4	19	0.71881	0.795533	2029	78	0.240992	0.281599	4415	744	0.154528	0.258272	8758
Ejecución 5	10	0.724884	0.765264	2050	37	0.249315	0.268577	4375	952	0.15624	0.288988	8404
Media	19.6	0.705401	0.784548	2038.4	93.8	0.221945	0.270778	4415.8	788.6	0.153826	0.26379	8494.4

Tabla 33: Ejecuciones con mayor mutación de AGGSF 20 %

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	12	0.629164	0.720095	2170	70	0.187521	0.256404	3938	167	0.137077	0.181831	6979
Ejecución 2	13	0.712436	0.810945	2006	32	0.240856	0.272346	4158	247	0.113161	0.179354	7007
Ejecución 3	16	0.598158	0.7194	1988	58	0.188315	0.245389	4002	227	0.127075	0.187908	7036
Ejecución 4	12	0.606698	0.69763	1984	57	0.187532	0.243622	3964	153	0.122546	0.163548	7247
Ejecución 5	17	0.604516	0.733336	2002	16	0.248449	0.264193	4052	242	0.1338	0.198653	7015
Media	14	0.630194	0.736281	2030	46.6	0.210535	0.256391	4022.8	207.2	0.126732	0.182259	7056.8

Tabla 34: Ejecuciones con mayor mutación de AGEUN 10 %

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	19	0.730199	0.806923	2217	98	0.210224	0.261244	4520	525	0.120473	0.19368	8617
Ejecución 2	29	0.714506	0.831611	2096	112	0.209973	0.268281	4665	465	0.123435	0.188275	9131
Ejecución 3	23	0.757446	0.850322	2091	131	0.201993	0.270193	4587	318	0.126505	0.170847	8711
Ejecución 4	26	0.716533	0.821523	2081	48	0.249415	0.274404	4538	286	0.131347	0.171227	8679
Ejecución 5	21	0.730196	0.814996	2061	50	0.249599	0.275629	4594	269	0.126432	0.163942	8630
Media	23.6	0.729776	0.825075	2109.2	87.8	0.224241	0.26995	4580.8	372.6	0.125638	0.177594	8753.6

Tabla 35: Ejecuciones con mayor mutación de AGEUN 20 %

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	17	0.618976	0.747796	1911	17	0.244259	0.260987	3736	318	0.147961	0.233182	7156
Ejecución 2	9	0.610733	0.678931	1883	16	0.248484	0.264228	3761	268	0.142436	0.214257	6653
Ejecución 3	8	0.62502	0.685641	1889	67	0.187642	0.253573	3796	241	0.140449	0.205034	6646
Ejecución 4	15	0.605306	0.71897	1916	53	0.188023	0.240177	3898	204	0.135299	0.189969	6710
Ejecución 5	11	0.613041	0.696395	1904	27	0.248421	0.27499	3761	262	0.138019	0.208232	6649
Media	12	0.614615	0.705547	1900.6	36	0.223366	0.258791	3790.4	258.6	0.140833	0.210135	6762.8

Tabla 36: Ejecuciones con mayor mutación de AGESF 10 %

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	11	0.770426	0.814845	2072	132	0.208612	0.277333	4357	487	0.130591	0.198499	8255
Ejecución 2	40	0.71281	0.874333	2000	39	0.250199	0.270503	4401	663	0.131389	0.223838	8331
Ejecución 3	10	0.733622	0.774003	2467	138	0.196004	0.267848	4369	372	0.125699	0.177571	8763
Ejecución 4	15	0.742016	0.802588	2004	34	0.247528	0.265229	4359	496	0.123739	0.192902	8337
Ejecución 5	32	0.721201	0.850419	2000	42	0.246927	0.268793	4395	438	0.129991	0.191066	8410
Media	21.6	0.736015	0.823238	2108.6	77	0.229854	0.269941	4376.2	491.2	0.128282	0.196775	8419.2

Tabla 37: Ejecuciones con mayor mutación de AGESF 20 %

Vamos a analizar los resultados. En primer lugar vamos a fijarnos en el infeasibility, y vemos que ocurre algo y es que se mejora siempre con esta nueva mutación excepto en el Glass del 20 % de restricciones, en donde siempre sale un resultado peor. Por otro lado, en Bupa por ejemplo se obtienen resultados mucho mejores. A pesar de ello, el valor final medio de la función objetivo es en todos los casos del Glass mejor esta nueva mutación, en Zoo salen resultados muy parecidos y en Bupa se consiguen mejorar mucho los resultados.

Los resultados con Bupa quizás chocan: en el análisis de la comparativa entre Meméticos y Genéticos dedujimos que el Memético funcionaba mejor en Bupa por la complejidad del mismo y por la capacidad explotadora del Memético. Ahora, sin embargo, estamos aumentando la mutación lo cual aumenta la capacidad exploradora del algoritmo, ¡y también estamos consiguiendo mejores resultados!

La respuesta puede estar en que la complejidad del problema hace que encontrar mínimos sea algo más complicado de lo normal, recordamos que antes, independientemente del número de instancias, mutábamos 5 genes, eso en conjuntos de 101 instancias o 214 instancias y 7 clases (Zoo y Glass) puede funcionar más o menos bien, porque hay menos combinaciones de soluciones posibles, luego hay más probabilidad de que la mutación vaya encaminada a un mínimo. En el caso del bupa tenemos ¡345 instancias y 16 clases!, esto hace que una mutación de 5 genes sea un cambio muy sútil en la población y haga que evolucionen las poblaciones muy lentamente. Aumentando el número de genes a mutar conseguimos acelerar la evolución. ¡OJO! el efecto contrario se tiene en problemas de pocas instancias, en donde mutar mucho tiene como consecuencia tener soluciones aleatorias. Es por ello que se ha visto que a veces en Glass y Zoo se consiguen resultados un poquito peores. Por lo tanto, no es incompatible el razonamiento hecho en la comparativa de Genéticos y Meméticos, la capacidad explotadora del Memético sigue siendo algo importante, pero también es importante acelerar la evolución en conjuntos grandes como en Bupa a través de una mayor mutación.

Concluimos que la probabilidad de mutación podría tener más sentido que dependiese del número de cromosomas y del número de genes por cromosoma, Es decir, que hubiese una probabilidad fija de mutación por gen, en vez de por cromosoma.

11.3. Búsquedas por Trayectorias

En esta sección analizaremos las 4 metaheurísticas relacionadas con la búsqueda basadas en trayectorias. Analizaremos en primer lugar Enfriamiento Simulado, seguiremos con la Búsqueda Local Reiterada junto con la Búsqueda Local Reiterada con Enfriamiento Simulado. Posteriormente se analizará la Búsqueda Multiarranque Básica frente a las dos anteriores. Y por último se hará un análisis conjunto de las 4 metaheurísticas.

11.3.1. Análisis Enfriamiento Simulado

Las tablas son el resultado de las ejecuciones tras 100.000 evaluaciones de la función objetivo. El algoritmo se ha ejecutado con todos los valores por defecto que se indican en el enunciado de la práctica utilizando el esquema de Cauchy para el enfriamiento.

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	10	0.624169	0.699945	1732	50	0.200596	0.249798	1652	973	0.242012	0.502765	807
Ejecución 2	9	0.631308	0.699506	1680	59	0.190511	0.24857	1743	979	0.242741	0.505101	802
Ejecución 3	2	0.763437	0.778593	1812	48	0.203836	0.25107	1723	965	0.244252	0.50286	812
Ejecución 4	8	0.623561	0.684182	1807	55	0.193565	0.247687	1560	976	0.241264	0.50282	829
Ejecución 5	9	0.616344	0.684543	1648	56	0.188344	0.243451	1788	998	0.245365	0.512817	812
Media	7.6	0.651764	0.709354	1735.8	53.6	0.195371	0.248115	1693.2	978.2	0.243127	0.505273	812.4

Tabla 38: Resultados obtenidos por el algoritmo ES con un 10 % de restricciones

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	41	0.613923	0.779484	1877	60	0.243357	0.274593	1763	1974	0.242144	0.5174	986
Ejecución 2	16	0.727228	0.791837	2029	41	0.248621	0.269966	1911	2011	0.236464	0.51688	979
Ejecución 3	46	0.620608	0.80636	1798	130	0.202758	0.270437	1867	1989	0.239222	0.51657	995
Ejecución 4	34	0.716554	0.853849	2006	42	0.248128	0.269994	1964	1972	0.24228	0.517258	988
Ejecución 5	29	0.716594	0.833698	1978	116	0.210162	0.270553	2163	1952	0.247049	0.519237	994
Media	33.2	0.678981	0.813046	1937.6	77.8	0.230605	0.271109	1933.6	1979.6	0.241432	0.517469	988.4

Tabla 39: Resultados obtenidos por el algoritmo ES con un 20 % de restricciones

Con las Tablas 38 y 39, habiendo analizado ya todo lo anterior, podemos ver que deja resultados decentes exceptuando en el conjunto BUPA, en donde deja una infactibilidad altísima junto con una función objetivo muy elevada también. Esto puede ser debido al número de instancias que posee el conjunto Bupa: al tener 345, se consigue que haya únicamente 30 enfriamientos, dejando poca versatilidad a la exploración de otros óptimos. También puede deberse a que aceptamos soluciones mucho peores de las que deberíamos, o que aceptamos muchas más soluciones de las que deberíamos: al final este conjunto tiene tantos clústers que obtener una solución peor a la que tenemos por mínima que sea, es algo muy probable, y por lo tanto se gastan también muchas evaluaciones de la función objetivo en explorar soluciones que tienen poco sentido. Al final de la sección se hará un estudio enfocado en esto para intentar mejorar los resultados y comprender mejor la fuente del problema.

En las gráficas 8 podemos ver como evoluciona el algoritmo en los 3 conjuntos de datos. El área sombreada indica el rango de valores que han tomado las funciones objetivos en la exploración de soluciones, siendo la linea resaltada la mejor solución hasta el momento.

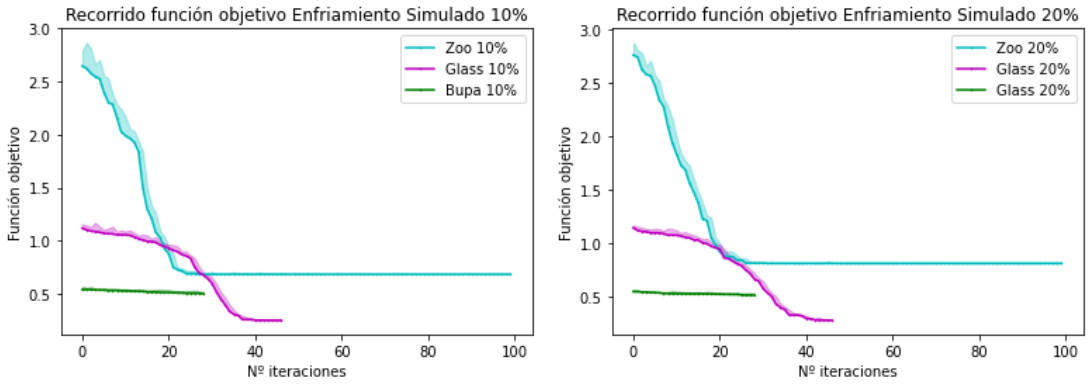

Figura 8: Evolución del valor de la función objetivo a lo largo de las iteraciones en los tres conjuntos de datos. La estala transparente indica el rango de valores que han tenido las soluciones consideradas por el algoritmo en cada iteración

11.3.2. Búsqueda Local Reiterada con BL y ES

Procedemos ahora a analizar los resultados obtenidos con las búsquedas locales reiteradas, tanto con búsqueda local como con enfriamiento simulado. Las ejecuciones se han hecho con 10.000 evaluaciones de la función objetivo en cada optimización (ya sea por BL o por ES). En el caso de ES se han dejado los valores por defecto que vienen en el enunciado.

En las Tablas 40, 41, 42 y 43 podemos ver los resultados de las ejecuciones. El análisis que podemos hacer entre estas dos metaheurísticas es sobre todo acerca de la diferencia entre BL y ES. Sabemos ya que BL se desenvuelve bastante bien dando buenos resultados, y efectivamente es lo que obtenemos al aplicarlo con ILS: obtenemos infeasibilitys razonablemente buenos junto con un valor de la función objetivo bastante bueno. Sin embargo, al aplicar ES obtenemos peores resultados, siendo terriblemente malos en el Bupa, algo que cabía esperar después de haber analizado ES en el apartado anterior. Por lo tanto, en este "duelo" ILS sale triunfadora, dejando buenos resultados, ante un "rival", ILS-ES que no ha sido nada competente en esta "lucha", dejando resultados muy malos.

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	8	0.605365	0.665986	536	45	0.195418	0.2397	1782	124	0.11822	0.15145	6551
Ejecución 2	11	0.604204	0.687558	693	48	0.187893	0.235127	2094	251	0.117842	0.185107	9605
Ejecución 3	15	0.509463	0.623128	539	47	0.187561	0.23381	1903	144	0.116887	0.155478	7183
Ejecución 4	9	0.581597	0.649796	762	41	0.228348	0.268693	1791	111	0.11896	0.148707	6650
Ejecución 5	5	0.602203	0.640091	590	56	0.185801	0.240907	1938	184	0.123019	0.172328	7864
Media	9.6	0.580566	0.653312	624	47.4	0.197004	0.243648	1901.6	162.8	0.118986	0.162614	7570.6

Tabla 40: Resultados obtenidos por el algoritmo ILS con un 10 % de restricciones

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	9	0.730534	0.766876	434	101	0.209042	0.261624	2739	205	0.119734	0.148319	7978
Ejecución 2	24	0.668127	0.765041	435	35	0.248623	0.266845	2220	269	0.113656	0.151166	9055
Ejecución 3	36	0.586597	0.731968	588	111	0.209288	0.267076	2533	267	0.127081	0.164311	8920
Ejecución 4	27	0.629814	0.738843	480	32	0.247604	0.264263	2074	227	0.126989	0.158642	8219
Ejecución 5	31	0.582324	0.707505	436	30	0.250854	0.266472	2167	277	0.119717	0.158342	8104
Media	25.4	0.639479	0.742047	474.6	61.8	0.233082	0.265256	2346.6	249	0.121435	0.156156	8455.2

Tabla 41: Resultados obtenidos por el algoritmo ILS con un 20 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	18	0.620407	0.756805	1611	61	0.188763	0.248789	2177	978	0.238247	0.50034	847
Ejecución 2	10	0.618613	0.694389	1919	49	0.191277	0.239495	2320	972	0.239797	0.500281	846
Ejecución 3	11	0.60675	0.690104	1760	51	0.188932	0.239118	2199	954	0.248567	0.504228	850
Ejecución 4	24	0.74183	0.923694	1630	51	0.188942	0.239128	2323	955	0.243392	0.49932	848
Ejecución 5	12	0.617626	0.708558	2042	61	0.189322	0.249349	2197	956	0.243844	0.500041	849
Media	15	0.641045	0.75471	1792.4	54.6	0.189447	0.243176	2243.2	963	0.242769	0.500842	848

Tabla 42: Resultados obtenidos por el algoritmo ILS-ES con un 10 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	22	0.709919	0.798757	2014	50	0.244	0.270031	2661	1986	0.232912	0.509842	1029
Ejecución 2	45	0.626912	0.808626	1757	114	0.211378	0.270727	2763	1952	0.241267	0.513456	1031
Ejecución 3	36	0.600842	0.746213	1807	76	0.239854	0.27942	2621	1952	0.232676	0.504865	1044
Ejecución 4	10	0.735948	0.776328	2137	39	0.25018	0.270484	2761	1949	0.239995	0.511766	1016
Ejecución 5	40	0.609028	0.770552	2437	40	0.248513	0.269338	3086	1985	0.235864	0.512654	1044
Media	30.6	0.65653	0.780095	2030.4	63.8	0.238785	0.272	2778.4	1964.8	0.236543	0.510516	1032.8

Tabla 43: Resultados obtenidos por el algoritmo ILS-ES con un 20 % de restricciones

11.3.3. Búsqueda Multiarranque Básica

En este apartado analizaremos los resultados de BMB comparándolo con los 3 anteriores de búsqueda por trayectorias.

En las Tablas 44 y 45 podemos ver los resultados de las ejecuciones. El desempeño de la metaheurística no es el mejor pero tampoco el peor. Las conclusiones que se pueden sacar acerca de esta metaheurística es que es difícil valorar su rendimiento con solo 10 iteraciones. Como se ha comentado en la parte del algoritmo BMB, esta metaheurística llega al óptimo global con probabilidad $\to 1$ cuando el número de iteraciones $\to \infty$. 10 iteraciones se nos quedan quizás cortas para ver la potencia de este algoritmo.

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	9	0.604377	0.672575	1016	54	0.187189	0.240328	3579	407	0.152699	0.261771	10569
Ejecución 2	10	0.598026	0.673802	1025	50	0.188962	0.238164	3577	405	0.148809	0.257345	10638
Ejecución 3	11	0.614512	0.697866	1008	61	0.186791	0.246818	3691	387	0.145532	0.249244	10616
Ejecución 4	8	0.612242	0.672863	1165	46	0.196003	0.241269	3622	393	0.146028	0.251347	10748
Ejecución 5	11	0.618613	0.701967	1088	46	0.195533	0.240799	3640	448	0.150678	0.270736	10698
Media	9.8	0.609554	0.683815	1060.4	51.4	0.190896	0.241476	3621.8	408	0.148749	0.258088	10653.8

Tabla 44: Resultados obtenidos por el algoritmo BMB con un 10 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	17	0.712827	0.781474	791	102	0.20978	0.262882	3933	688	0.14895	0.244886	13369
Ejecución 2	10	0.734551	0.774932	960	112	0.209298	0.267607	4088	813	0.140065	0.253431	13226
Ejecución 3	16	0.713206	0.777816	860	26	0.249461	0.262997	3973	823	0.13867	0.25343	12826
Ejecución 4	10	0.733622	0.774003	1051	33	0.248395	0.265575	4070	741	0.146359	0.249685	13005
Ejecución 5	14	0.733343	0.789876	892	35	0.245665	0.263887	4099	658	0.135316	0.227068	12852
Media	13.4	0.72551	0.77962	910.8	61.6	0.23252	0.26459	4032.6	744.6	0.141872	0.2457	13055.6

Tabla 45: Resultados obtenidos por el algoritmo BMB con un 20 % de restricciones

11.3.4. Comparación entre las búsquedas por trayectorias

Vamos a ver ahora conjuntamente los cuatro algoritmos en las tablas poniendo la media de los resultados de las ejecuciones.

Destacar los algoritmos ILS-ES y ES, que tienen un desempeño realmente malo, especialmente en el conjunto de Bupa. Esto puede ser debido a como hemos hablado antes, al tipo de

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
ES	7.6	0.651764	0.709354	1735.8	53.6	0.195371	0.248115	1693.2	978.2	0.243127	0.505273	812.4
ILS	9.6	0.580566	0.653312	624	47.4	0.197004	0.243648	1901.6	162.8	0.118986	0.162614	7570.6
ILS-ES	15	0.641045	0.75471	1792.4	54.6	0.189447	0.243176	2243.2	963	0.242769	0.500842	848
BMB	9.8	0.609554	0.683815	1060.4	51.4	0.190896	0.241476	3621.8	408	0.148749	0.258088	10653.8

Tabla 46: Media de las ejecuciones de las búsquedas por trayectorias con 10 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
ES	33.2	0.678981	0.813046	1937.6	77.8	0.230605	0.271109	1933.6	1979.6	0.241432	0.517469	988.4
ILS	25.4	0.639479	0.742047	474.6	61.8	0.233082	0.265256	2346.6	249	0.121435	0.156156	8455.2
ILS-ES	30.6	0.65653	0.780095	2030.4	63.8	0.238785	0.272	2778.4	1964.8	0.236543	0.510516	1032.8
BMB	13.4	0.72551	0.77962	910.8	61.6	0.23252	0.26459	4032.6	744.6	0.141872	0.2457	13055.6

Tabla 47: Media de las ejecuciones de las búsquedas por trayectorias con 10 % de restricciones

aceptación de soluciones que tenemos, tanto en función de lo peores que son, como la probabilidad de ser aceptadas.

Centrándonos ahora en ILS y BMB podemos decir que los dos son superiores a los que hemos visto de ES. Entre ellos dos, destacamos ILS puesto que deja unos números en infactibilidad y función objetivo mucho mejores que los de BMB. Ya hemos dicho que la potencialidad de BMB está en dejar que el número de iteraciones tienda a infinito. Así, ILS sale como clara vencedora entre los 4 algoritmos de esta sección. La variación de ILS frente a BMB en no empezar desde una solución completamente aleatoria hace, para un número bajo de iteraciones como es nuestro caso, sea un algoritmo mucho mejor que BMB, permitiendo encontrar mínimos mucho más rápido, aunque estos no sean los mejores. Con esto último me explico. Es probable que cuando tengamos un número de iteraciones lo suficientemente alto, BMB tenga mejor rendimiento que ILS puesto que ILS tiene una búsqueda mucho más local. BMB en realidad busca "vecinos" que están más alejados de él que un vecino normal: modificamos el cluster del 10 % de las instancias, en vez de 1 sola, esto hace que el 90 % de las instancias sigan en el mismo clúster, de aquí que sea un algoritmo que busca más localmente. Evidentemente BMB con soluciones aleatorias iniciales, no tiene nada de búsqueda local (quitando la optimización, evidentemente), esto hace que busque por todo el espacio de forma arbitraria.

Conclusión: para el número de iteraciones que tenemos, 10, ILS tiene un mejor desempeño porque es un mejor buscador local que BMB. Si tuvieramos un número de iteraciones lo suficientemente alto, BMB podría tener mejor desempeño que ILS por su capacidad de búsqueda local al generar soluciones aleatorias iniciales.

Graficado

En las gráficas de la Figura 9 podemos ver el comportamiento de los tres algoritmos ILS, ILS-ES y BMB a lo largo de las iteraciones en cada conjunto de datos. Es destacable el gran sombreado rosa, que hace referencia al algoritmo BMB, que evidentemente tiene soluciones iniciales peores puesto que son aleatorias. Nos damos cuenta como de repente en algunos datos, el algoritmo ILS-ES (en Zoo 10 %) y el algoritmo ILS (en Glass 20 %) se tienen picos de malos resultados. Esto se debe a que no dejan de ser algoritmos con un cierto factor de exploración, y en estos casos, los picos indican una exploración hacia una solución peor.

En las gráficas de Bupa, se puede ver claramente la diferencia de rendimiento entre los tres algoritmos. Algo de lo que ya hemos hablado.

De especial relevancia de estos gráficos es un información que no se observa en las tablas, esta información responde a la pregunta: *¿A qué altura del proceso hemos obtenido la que ha resultado ser la mejor solución?.*

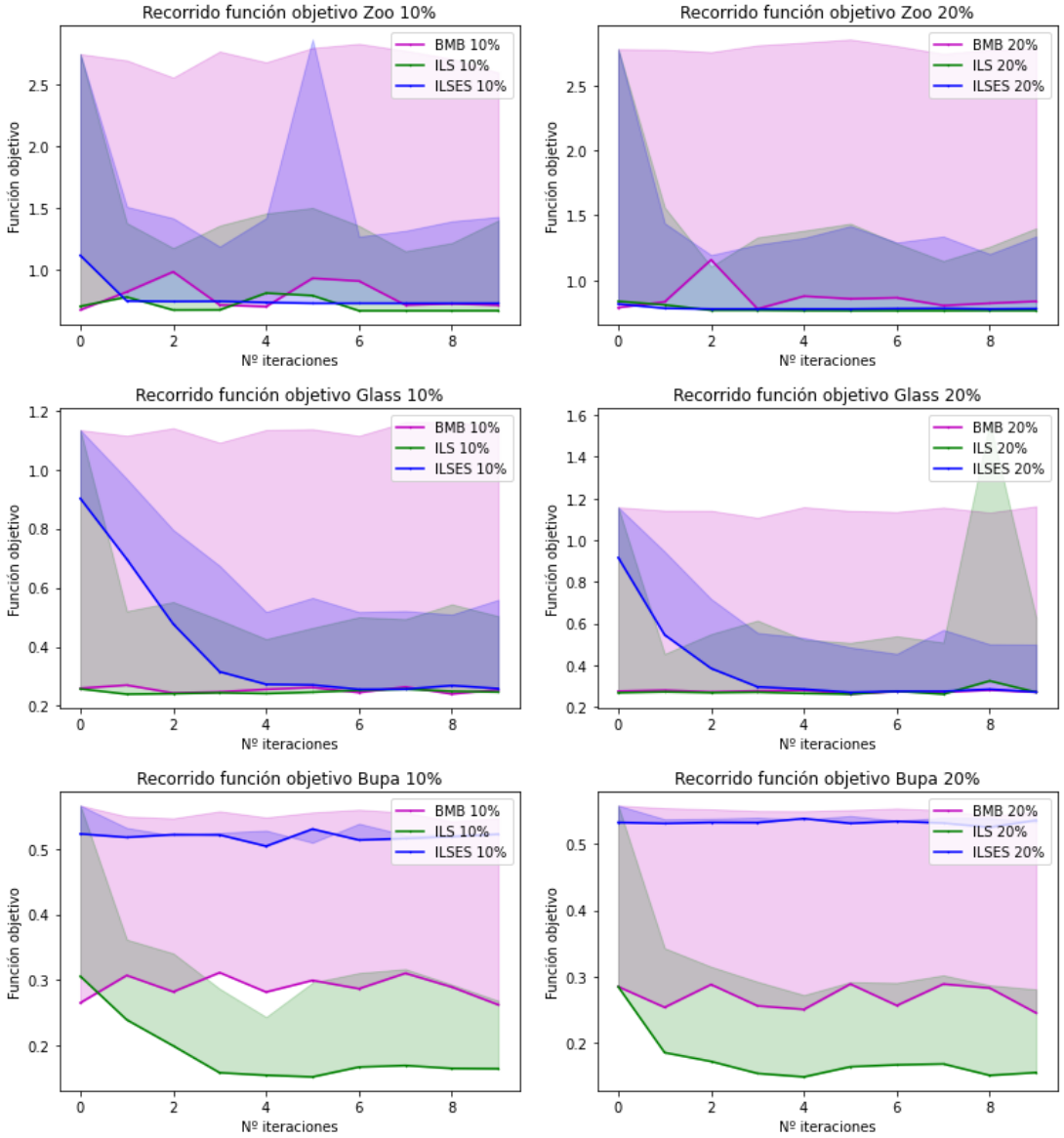

Figura 9: Evolución del valor de la función objetivo a lo largo de las iteraciones en los tres algoritmos de ILS, ILS-ES y BMB. La estala transparente indica el rango de valores que han tenido las soluciones consideradas por el algoritmo en cada iteración

La respuesta a la pregunta anteriormente realizada es tarea "sencilla" de responder si sabemos interpretar cada algoritmo en cada gráfico:

1. Empezando por el algoritmo BMB. De este no podemos afirmar nada, vemos en general un comportamiento muy uniforme. Aunque se haya conseguido con antelación un mínimo de la función objetivo, esto no quiere decir que para otra ejecución con otra semilla baste hacer menos. Sabemos que la virtud de este algoritmo está en realizar muchas iteraciones, puesto que tiene convergencia asegurada al óptimo global cuando las iteraciones tienden a infinito.

2. Yendo ahora a por el algoritmo ILS, tenemos que es el que antes encuentra una mejor

solución, que no quiere decir, que sea la mejor solución al final del proceso. Su capacidad de explotación con su BL y su inicio en soluciones mutadas encontradas con anterioridad, hace que esto sea así, consiguiendo disminuir lo más rápido posible el valor de la función objetivo. Su capacidad de exploración gracias a la mutación hace que por lo general se consiga llegar a mínimos mejores conforme avance el algoritmo.

3. Por último ILS-ES. Es el que encuentra peor posicionado. La capacidad de exploración que tiene entre la mutación y el enfriamiento simulado, hace que le se dificultoso explotar buenas zonas. Mejora también conforme avanzan las iteraciones. En el bupa ya sabemos el mal rendimiento que tiene.

11.3.5. Análisis extra: Mejora de ES

Recordamos que tenemos $\mu = \phi = 0{,}3$, quizás aceptar soluciones un 30 % peores con un 0.3 de probabilidad sea un problema para el desempeño de los modelos. Nos damos cuenta que si bajamos la probabilidad de aceptar las soluciones peores a 0, entonces tendríamos un algoritmo similar al de BL. Por otro lado, si acercamos esta probabilidad a 1 tendríamos un algoritmo muy cercano a lo que sería un método aleatorio (sin serlo, porque al final acotamos el conjunto de las soluciones con su porcentaje de empeoramiento). Por otro lado, la forma de actualizar el enfriamiento también va a ser clave para saber en qué momento nos centramos en buscar el óptimo local y no en explorar otros óptimos cercanos. En la gráfica 10 vemos una comparativa de como se comportan los enfriamientos de Cauchy y Proporcionales.

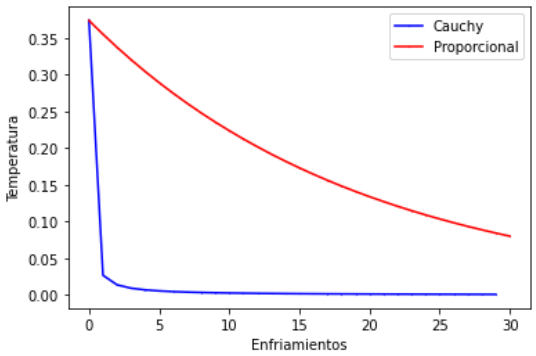

Figura 10: Enfriamiento de Cauchy Vs Enframiento proporcional ($T_{k+1} = \alpha T_k$, $\alpha = 0{,}95$). Datos con el conjunto Bupa.

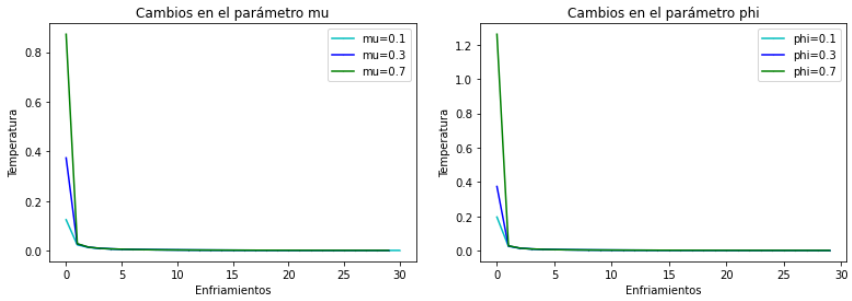

Figura 11: Enfriamiento de Cauchy cambiando los parámetros. Datos Bupa.

Fijándonos en la gráfica de los enfriamientos (Gráfica 10), vemos que la proporcional coge casi forma de recta. Sin embargo la de Cauchy tiene claramente una curva muy pronunciada, obteniendo prácticamente una esquina. Vamos a ver como evolucionan los enfriamientos cambiando los parámetros μ, ϕ. Esta comparativa podemos verla en la gráfica 11, en donde vemos

que implican un comportamiento mucho más suave conforme se bajan ambos parámetros, creando mayores cambios el parámetro ϕ para valores grandes puesto que está en el denominador.

Probando distintas combinaciones se han obtenido unos buenos resultados con los parámetros $\mu = 0{,}1$ y $\phi = 0{,}4$ y poniendo una temperatura final un poco más baja, a 10^{-4}. La gráfica de la evolución de temperaturas con enfriamientos se puede ver en la Figura 12. Conseguimos una esquina muy pronunciada, esto hace que haya un periodo de búsqueda de óptimos más alejados, pero rápidamente buscamos muy localmente. En cierto modo estamos diciendo que se aceptan con un 0.4 de probabilidad (que he considerado una probabilidad alta) soluciones un 10 % peores.

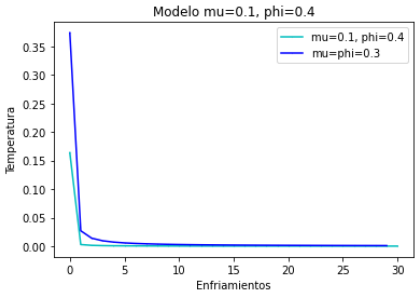

Figura 12: Parámetros $\mu = 0{,}1, \phi = 0{,}4, T_f = 10^{-4}$. Datos Bupa.

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
ES	7.6	0.651764	0.709354	1735.8	53.6	0.195371	0.248115	1693.2	978.2	0.243127	0.505273	812.4
ESmod	14.6	0.603137	0.713771	2410.6	54.2	0.197083	0.250418	4008	109.2	0.11839	0.147655	7108.6
ILS-ES	15	0.641045	0.75471	1792.4	54.6	0.189447	0.243176	2243.2	963	0.242769	0.500842	848
ILS-ESmod	11.8	0.656377	0.745794	2222.8	37.6	0.213347	0.250347	3818.8	119.8	0.116558	0.148663	7032
Búsqueda L.	13.2	0.618357	0.718382	107.2	41.6	0.20681	0.247746	453.4	106.2	0.112189	0.140649	5381.6

Tabla 48: Comparación con las modificaciones. (Medias-10 %)

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
ES	33.2	0.678981	0.813046	1937.6	77.8	0.230605	0.271109	1933.6	1979.6	0.241432	0.517469	988.4
ES-mod	31.6	0.681343	0.808946	2188.6	80.2	0.226045	0.267798	4584.2	244.8	0.115283	0.149419	8443.4
ILS-ES	30.6	0.65653	0.780095	2030.4	63.8	0.238785	0.272	2778.4	1964.8	0.236543	0.510516	1032.8
ILS-ESmod	24.6	0.735914	0.835251	2215	70.6	0.232996	0.269751	4577	247.2	0.114918	0.149387	9031.8
Búsqueda L.	22.4	0.725855	0.816308	105.2	58	0.24105	0.271245	460.8	206.6	0.11661	0.145418	6064.8

Tabla 49: Comparación con las modificaciones. (Medias-20 %)

Los resultados obtenidos con estos nuevos parámetros se pueden ver en las tablas 50,51, 52 y 53. Las comparaciones con los anteriores las tenemos en las tablas 48 y 49. Vemos como hay un cambio sustancial en el conjunto bupa, donde ahora se consiguen buenos resultados. Casualmente, en bupa se consiguen resultados muy parecidos a los de búsqueda local normal, por eso se ha incluído también en las tablas. Esto es porque en realidad estamos dejando muy poca cobertura para explorar otros óptimos, y en realidad a partir de una iteración muy temprana prácticamente se impide aceptar soluciones peores, haciendo que se haga una especie de búsqueda local. Los demás conjuntos no se han visto demasiado afectados.

La conclusión que sacamos es que para conjuntos grandes como Bupa es más rentable una búsqueda local a una global, debido a la infinidad de posibilidades que tiene el espacio de un conjunto tan grande: concluimos que centrarnos en subespacio local del problema y buscar ahí el mínimo es la mejor alternativa entre estas dos disyuntivas

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	23	0.551804	0.726089	1934	55	0.209952	0.264075	4597	93	0.120239	0.145162	7520
Ejecución 2	10	0.624708	0.700484	3197	59	0.188198	0.246257	3935	121	0.120695	0.153122	7523
Ejecución 3	14	0.610816	0.716903	2370	37	0.21136	0.247769	4043	101	0.115673	0.142739	7210
Ejecución 4	17	0.620049	0.748869	2184	60	0.187726	0.246768	3999	135	0.116841	0.153019	6755
Ejecución 5	9	0.608311	0.676509	2368	60	0.188179	0.247222	3466	96	0.118505	0.144231	6535
Media	14.6	0.603137	0.713771	2410.6	54.2	0.197083	0.250418	4008	109.2	0.11839	0.147655	7108.6

Tabla 50: Resultados obtenidos por el algoritmo ESmod con un 10 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	23	0.71821	0.811086	2260	108	0.210272	0.266498	4804	246	0.116995	0.151297	8591
Ejecución 2	47	0.591054	0.780844	2409	60	0.249902	0.281138	4219	274	0.110857	0.149064	8623
Ejecución 3	19	0.715152	0.791876	2067	101	0.21035	0.262931	4269	197	0.119063	0.146532	9244
Ejecución 4	26	0.758949	0.863939	2129	32	0.249927	0.266587	5272	312	0.109558	0.153064	8606
Ejecución 5	43	0.623347	0.796984	2078	100	0.209774	0.261835	4357	195	0.119944	0.147135	7153
Media	31.6	0.681343	0.808946	2188.6	80.2	0.226045	0.267798	4584.2	244.8	0.115283	0.149419	8443.4

Tabla 51: Resultados obtenidos por el algoritmo ESmod con un 20 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	12	0.665119	0.756051	2306	52	0.187647	0.238817	3974	115	0.116938	0.147756	7063
Ejecución 2	10	0.78983	0.865606	2141	23	0.242832	0.265465	4418	124	0.10693	0.140161	7564
Ejecución 3	12	0.618294	0.709226	2215	46	0.188094	0.23336	3881	103	0.119392	0.146995	6780
Ejecución 4	12	0.587676	0.678607	2448	43	0.211002	0.253315	3433	150	0.116988	0.157186	6992
Ejecución 5	13	0.620969	0.719478	2004	24	0.237159	0.260776	3388	107	0.122541	0.151216	6761
Media	11.8	0.656377	0.745794	2222.8	37.6	0.213347	0.250347	3818.8	119.8	0.116558	0.148663	7032

Tabla 52: Resultados obtenidos por el algoritmo ILS-ESmod con un 10 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	20	0.721107	0.801868	2111	31	0.250875	0.267013	4480	285	0.114141	0.153882	9791
Ejecución 2	23	0.71821	0.811086	2341	124	0.208695	0.27325	5381	224	0.112702	0.143937	10132
Ejecución 3	39	0.710964	0.868449	2164	46	0.245812	0.26976	5436	234	0.115833	0.148462	8330
Ejecución 4	13	0.799149	0.851644	2371	39	0.250113	0.270416	3746	244	0.115174	0.149197	8529
Ejecución 5	28	0.730141	0.843207	2088	113	0.209487	0.268316	3842	249	0.116738	0.151459	8377
Media	24.6	0.735914	0.835251	2215	70.6	0.232996	0.269751	4577	247.2	0.114918	0.149387	9031.8

Tabla 53: Resultados obtenidos por el algoritmo ILS-ESmod con un 20 % de restricciones

11.4. Análisis conjunto de todos los algoritmos

En las tablas 54 y 55 se han recogido los valores medios de las ejecuciones de los algoritmos más destacables. Los algoritmos que no aparecen en estas tablas son aquellos que guardan una similitud muy grande en sus resultados con alguno de los algoritmos que sí aparecen. En concreto, cualquier genético se ha visto que es muy similar a cualquier otro genético y por ello el representante es AGG-SF. Por otro lado el Memético(10,0.1mej) es similar a Memético(10,0.1) que es el representante.

	Zoo				**Glass**				**Bupa**			
Medias	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Greedy COPKM	2.2	1.00395	1.02062	3.4	3	0.379019	0.381972	14.4	22.6	0.243681	0.249737	151.6
Búsqueda L.	13.2	0.618357	0.718382	107.2	41.6	0.20681	0.247746	453.4	106.2	0.112189	0.140649	5381.6
AGG Segmento F.	12.2	0.651387	0.743834	2014.6	48.4	0.232877	0.280505	3935	616	0.156167	0.321248	6954.8
Memético(10,0.1)	16.6	0.680662	0.806451	1902.4	100.6	0.241975	0.34097	3846.2	439.6	0.16363	0.281438	6365.4
Memético(10,1.0)	20.4	0.803324	0.957908	1812.8	281.8	0.299608	0.57691	3703	954.8	0.205776	0.461651	7859.2
ES	7.6	0.651764	0.709354	1735.8	53.6	0.195371	0.248115	1693.2	978.2	0.243127	0.505273	812.4
ILS	9.6	0.580566	0.653312	624	47.4	0.197004	0.243648	1901.6	162.8	0.118986	0.162614	7570.6
ILS-ES	15	0.641045	0.75471	1792.4	54.6	0.189447	0.243176	2243.2	963	0.242769	0.500842	848
BMB	9.8	0.609554	0.683815	1060.4	51.4	0.190896	0.241476	3621.8	408	0.148749	0.258088	10653.8

Tabla 54: Medias de los resultados obtenidos con los algoritmos más destacables con el 10 %const

	Zoo				**Glass**				**Bupa**			
Medias	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Greedy COPKM	1.6	0.954523	0.960984	3.8	4.4	0.328806	0.331096	14	13	0.237523	0.239335	127.6
Búsqueda L.	22.4	0.725855	0.816308	105.2	58	0.24105	0.271245	460.8	206.6	0.11661	0.145418	6064.8
AGG Segmento F.	19.4	0.714891	0.79323	2111.8	69.2	0.241062	0.277089	4530.6	996.4	0.153857	0.292796	8918.4
Memético(10,0.1)	22	0.725273	0.814111	2011	122	0.240185	0.303699	4312.4	877.8	0.16186	0.284261	8232.2
Memético(10,1.0)	36	0.693437	0.838808	1966	465	0.292126	0.534209	4477.8	1857.4	0.198632	0.45763	10867
ES	33.2	0.678981	0.813046	1937.6	77.8	0.230605	0.271109	1933.6	1979.6	0.241432	0.517469	988.4
ILS	25.4	0.639479	0.742047	474.6	61.8	0.233082	0.265256	2346.6	249	0.121435	0.156156	8455.2
ILS-ES	30.6	0.65653	0.780095	2030.4	63.8	0.238785	0.272	2778.4	1964.8	0.236543	0.510516	1032.8
BMB	13.4	0.72551	0.77962	910.8	61.6	0.23252	0.26459	4032.6	744.6	0.141872	0.2457	13055.6

Tabla 55: Medias de los resultados obtenidos con los algoritmos más destacables con el 20 %const

En las tablas destaca la gran superioridad del algoritmo Greedy COPKM en términos de infactibilidad. Es de largo el mejor. Recuperamos el argumento que se dio en su comparación con el algoritmo de Búsqueda Local acerca de que el algoritmo Greedy COPKM quiere minimizar únicamente la infactibilidad consiguiendo en cada iteración minimizar la distancia intra-clúster asignando instancias al clúster más cercano. Al otro lado del muro tenemos el Memético(10,1.0) que tiene resultados terriblemente malos que hemos achacado quizás a la falta de exploración debido a que se centra demasiado en la explotación agotando evaluaciones de la función objetivo. Como algoritmos malos también destacar el mal desempeño del algoritmo ES e ILS-ES en el conjunto Bupa, ya en su estudio dijimos que una búsqueda global de este tipo en conjuntos tan grandes no es conveniente.

Búsqueda Local con Genético y Memético

Comparando ahora Búsqueda Local con Genético y Memético, tenemos unos resultados interesantes puesto que vemos como, conforme aumenta la complejidad del problema en término de número de instancias, el algoritmo de búsqueda local consigue llegar a soluciones cada vez mejores mientras que el Genético y el Memético se ven claramente perjudicados. Esto reafirma la idea que teníamos en la comparación entre Genético y Memético de la subsección anterior, en donde se hablaba de la necesidad de explotación en problemas donde el espacio de soluciones es más complejo. La comparación entre estos tres tipos de algoritmos es muy interesante: en Zoo Gana AGG-SF , en Glass gana BL pero no por mucho y en Bupa gana BL por muchísimo. Desde el punto de vista del Memético, nunca gana, pero es el más estable, dando resultados no los mejores, pero tampoco disparatados: sin distanciarse mucho de los otros dos. Es en esta última

afirmación donde nos damos cuenta empíricamente de como se hace notar que el Memético se basa en usar parte de la teoría de algoritmos Genéticos con parte de Búsqueda Local.

Como conclusión, entre estos tres algoritmos podríamos destacar la Búsqueda Local frente a Genético y Memético simplemente porque se comporta muy bien en comparación a los otros dos en conjuntos de datos más complicados y se comporta de forma muy parecida a la competencia en los conjuntos de datos más simples.

Trayectorias vs BL y Greedy

Pasamos ahora la comparación de las últimas metaheurísticas estudiadas, las de búsqueda por trayectorias,, con la búsqueda local y el greedy. El Greedy como ya se ha comentado a lo largo de todo el documento, es el gran ganador por su baja infeasibility, el motivo es el que ya se comentó, su objetivo es minimizar precisamente la infactibilidad, mientras que el resto buscan minimizar la función objetivo.

Comparando ahora con Búsqueda Local, los algoritmos ILS-ES y ES son claros perdedores en el conjunto Bupa. Sin embargo, en los conjuntos Zoo y Glass salen victoriosos estos dos últimos (sobre todo en el conjunto de 10 % de restricciones). La comparación con sentido es comparar Búsqueda Local frente a Enfriamiento Simulado: la búsqueda local se centra en una zona concreta del espacio de soluciones e intenta explotarla buscando el óptimo local. Por otro lado, Enfriamiento Simulado mete estocasticidad con un operador de mutación con el que se amplía la búsqueda a óptimos más lejanos. Esto es importante porque como bien hemos explicado en el análisis de los malos resultados del Bupa, estos resultados venían de la mano de la complejidad del conjunto Bupa (muchas instancias con muchos clústers) y por ello se obtenían malos resultados.

La explicación era que al tener un conjunto tan grande, se hacían pocas iteraciones y, si se aceptaban instancias que fueran un tanto por ciento grande peores, pues además implicaba un gasto de evaluaciones de la función objetivo muy grandes. Ahora pasa a la inversa, y vemos como en el conjunto Zoo con 7 clases y 214 instancias es el más pequeño y esto implica una mejora sustancial incluso respecto de Búsqueda Local, permitiendo hacer muchas más exploraciones por el espacio de soluciones con mayor probabilidad de éxito. El termino intermedio lo tenemos en Glass: 7 clases y 214 instancias, en donde ambas partes tienen un desempeño similar.

Si nos fijamos ahora en BMB, pasa parecido: conseguimos en conjuntos pequeños muy buenos resultados debido a que es un algoritmo de exploración máxima, en donde cada iteración empieza en una solución aleatoria. En Zoo es mejor que BL, en Glass más o menos mismo rendimiento y en Bupa sale perdedor de largo.

Con ILS no iba a pasar lo mismo. ILS mejora prácticamente a BL: en los conjuntos de Glass y Bupa tienen un desempeño similar, pero en Zoo es claramente ganador. El hecho de que en Zoo sea el ganador es debido a que el espacio de soluciones es el más pequeño y al tener una capacidad de exploración que no tiene BL hace que su búsqueda sea mucho más enriquecedora, pudiendo encontrar óptimos locales más próximos al óptimo global. Conforme crece la complejidad del problema, decrece la superioridad de ILS frente a BL, pero no llega a ser BL claramente mejor en ningún conjunto. Al final, ILS posee tanto un componente de 'exploración' como un componente de búsqueda local, que es precisamente la Búsqueda Local con la que la estamos comparando, lo que hace que el algoritmo de Búsqueda Local tenga pocas posibilidades de ser superior a ILS.

En las tablas 56 y 57 Se pueden ver las medias de todos las metaheurísticas estudiadas.

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Greedy COPKM	2.2	1.00395	1.02062	4	3	0.379019	0.381972	14.4	22.6	0.243681	0.249737	151.2
Búsqueda L.	11.2	0.698878	0.783747	104.2	35.8	0.220812	0.25604	485	115.8	0.115316	0.146349	4900
AGG-UN	11.8	0.67729	0.766706	2228.6	59.4	0.215896	0.274348	4369.4	535.4	0.157644	0.301125	7568
AGG-SF	10.6	0.719239	0.799562	2047.8	72.6	0.232004	0.303445	4043.8	555	0.158619	0.307353	7057
AGE-UN	14.8	0.660433	0.772582	2248.8	51.6	0.22632	0.277096	4358.2	575.2	0.158268	0.312415	7567.6
AGE-SF	11.2	0.707089	0.791958	2048.6	73.8	0.226015	0.298637	4059.8	595.6	0.162037	0.321651	7016.4
MEM-10	22	0.733671	0.900379	1856.8	318	0.300247	0.613172	3709.2	947.6	0.211939	0.465884	7923.2
MEM-01	13.2	0.64638	0.746405	1931.4	110.2	0.246251	0.354692	3741	393.8	0.164785	0.270319	6470
MEM-01best	12.4	0.614136	0.708098	1961.6	107.4	0.219041	0.324727	3750.4	464.8	0.163558	0.288118	6443.6
ES	11	0.607004	0.690358	1767.4	57	0.192754	0.248844	1691.8	982.2	0.242533	0.505752	829
BMB	13.2	0.581101	0.681126	1085.2	51.4	0.189912	0.240492	3757.2	438	0.148331	0.26571	11307
ILS	8.6	0.588852	0.65402	602.4	48	0.192754	0.239988	2046.8	145.6	0.11769	0.156709	7760.2
ILS-ES	12	0.617069	0.708001	1755.2	51.6	0.192463	0.24324	2317.6	958.6	0.241229	0.498122	873.4

Tabla 56: Medias de todas las metaheurísticas con el 10 % de restricciones

	Zoo				Glass				Bupa			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Greedy COPKM	1.6	0.954523	0.960984	3.6	4.4	0.328806	0.331096	14.2	13	0.237523	0.239335	128
Búsqueda L.	21	0.734545	0.819345	105.8	72.8	0.231721	0.269621	412.2	232.4	0.118093	0.150499	5587.8
AGG-UN	26.6	0.724781	0.832194	2301.8	62.8	0.240251	0.272945	4964	904.2	0.151822	0.277905	9299.8
AGG-SF	25.4	0.738694	0.841261	2148.2	59.6	0.245709	0.276737	4567.4	1033.2	0.154442	0.298513	8955.8
AGE-UN	26	0.722982	0.827972	2307.2	105.8	0.217246	0.272327	4901.6	1034	0.151814	0.295996	9320.6
AGE-SF	24.4	0.711422	0.809951	2174.2	53.2	0.246937	0.274633	4645.6	1035.8	0.157426	0.301859	8716.6
MEM-10	47.8	0.735365	0.928385	1994.8	493.2	0.288409	0.545174	4344	1923.6	0.208949	0.477178	9820.8
MEM-01	33.8	0.684034	0.820521	2050	164.2	0.234416	0.3199	4299.4	1183	0.163436	0.328395	8396.6
MEM-01best	42	0.688242	0.857841	2041.4	156.8	0.229873	0.311504	4318.2	980.6	0.159222	0.295958	8205.4
ES	28.4	0.677227	0.791908	1921.6	58	0.239135	0.269331	1964	1969.6	0.23852	0.513163	996.8
BMB	28.6	0.66451	0.779999	918.2	80.4	0.222843	0.2647	4073.4	777.4	0.144542	0.252943	13457.4
ILS	21.6	0.671371	0.758594	494.6	34.2	0.247176	0.264981	2116.2	281.4	0.120078	0.159316	9105.2
ILS-ES	27.8	0.717378	0.829637	1848	57.6	0.250318	0.280305	2592	1936.4	0.23776	0.507773	1047

Tabla 57: Medias de todas las metaheurísticas con el 20 % de restricciones

11.4.1. Análisis extra: uso de Búsqueda Local (Sección 3) en lugar de BLS

En este apartado se ha decidido probar sustituir la búsqueda local suave por la búsqueda local utilizada en la Sección **3**. Para ello, se ejecutará la búsqueda local #(número de instancias) veces cada 10 épocas. Que aproximadamente, es lo que se ejecuta la búsqueda local suave, salvando condiciones de parada.

En las Figuras 13, 14 y 15 podemos ver una comparativa gráfica sobre la evolución de la función objetivo de los algoritmos Meméticos con BLS, Meméticos con BL y BL.

En estas gráficas observamos visualmente como mejora mucho la ejecución: en Zoo no se aprecia mucho ya que se conseguían buenos resultados con la búsqueda Local suave, pero en Glass y Bupa se nota como mejora mucho la inclusión de la Búsqueda Local en lugar de la Suave. Cabe notar, los escalones tan grandes que se forman: esto es porque nuestra Búsqueda Local consume muchas evaluaciones de la función objetivo. También cabe notar que hay alguna gráfica en la que únicamente vemos dos escalones, esto se debe a que como los datos de la función objetivo de la gráfica están sobre el mejor de la población en cada generación, es posible que tras el proceso de explotación no se haya conseguido mejorar al mejor cromosoma de la población, quedando la mejor solución igual.

Comparando la Búsqueda Local con la Búsqueda Local Suave, vemos como es mucho mejor explotadora , consiguiendo pendientes muy inclinadas que implica una gran mejora de la solución.

Se puede concluir en esta comparativa entre Meméticos, que la opción de usar Búsqueda Local es mejor que la de Búsqueda Local Suave sin mucha discusión, aunque, evidentemente algo tendremos que perder con esta nueva implementación, ¿no?

Efectivamente perdemos en velocidad de ejecución. En las Tablas 58 y 59 podemos ver las ejecuciones del Memético(10,1.0) con Búsqueda Local, y en las Tablas 60 y 61 las ejecuciones del Memético(10,0.1) con Búsqueda Local. Estas 4 tablas las podemos comparar con las Tablas 24, 25 y 26, 27 respectivamente.

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	10	0.609071	0.684847	2656	62	0.190784	0.251794	8437	567	0.154013	0.305963	67810
Ejecución 2	10	0.609221	0.684997	2698	14	0.248834	0.262611	8523	637	0.163606	0.334315	67831
Ejecución 3	15	0.611189	0.724854	2638	63	0.187684	0.249679	8516	572	0.150667	0.303957	68690
Ejecución 4	14	0.60805	0.714137	2644	16	0.244027	0.259772	8542	624	0.179428	0.346653	68340
Ejecución 5	9	0.738985	0.807184	2684	56	0.190874	0.24598	8557	672	0.158898	0.338986	67627
Media	11.6	0.635303	0.723204	2664	42.2	0.212441	0.253967	8515	614.4	0.161323	0.325975	68059.6

Tabla 58: Ejecuciones con Búsqueda Local Normal del Memético(10,1.0) con el 10 %

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	24	0.713874	0.810788	2773	106	0.210093	0.265278	9075	987	0.151788	0.289416	69786
Ejecución 2	16	0.733622	0.798232	2722	50	0.24962	0.275651	9084	1258	0.163854	0.339271	69565
Ejecución 3	10	0.733622	0.774003	3277	130	0.208843	0.276522	9670	1150	0.167049	0.327407	69928
Ejecución 4	31	0.597283	0.722464	3115	51	0.249487	0.276038	9254	934	0.152624	0.282862	69899
Ejecución 5	13	0.731853	0.784348	2773	55	0.246359	0.274993	9150	1096	0.146443	0.29927	69924
Media	18.8	0.702051	0.777967	2932	78.4	0.232881	0.273696	9246.6	1085	0.156352	0.307645	69820.4

Tabla 59: Ejecuciones con Búsqueda Local Normal del Memético(10,1.0) con el 20 %

Yendo al grano sobre en qué sale perdiendo la implementación con Búsqueda Local, está claro que salen perdiendo en el tiempo de ejecución. Esto es muy notorio en el caso de Memético(10,1.0) en bupa, en donde la ejecuciciones llegan a tardar más de 1 minuto, esto es sobre una 7 veces más lento que la implementación con la Búsqueda Local Suave. En el caso de Memético(10,0.1) el cambio no es tan fuerte, pero sigue siendo el doble de lento, lo cual es bastante decir. El caso del Memético(10,0.1mej) es similar al de Memético(10,0.1) y se ha decidido no meter la tabla y seguir hablando únicamente del Memético(10,0.1) como "representante" de ambos con tal de no meter mucha información redundante y que sea la memoria más legible.

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	6	0.663591	0.709056	1702	25	0.243103	0.267704	3261	93	0.116127	0.14105	13939
Ejecución 2	10	0.608205	0.683981	1710	15	0.248273	0.263033	3264	149	0.118095	0.158025	13960
Ejecución 3	12	0.585393	0.676325	1652	20	0.248086	0.267767	3611	98	0.12025	0.146513	14391
Ejecución 4	11	0.614392	0.697746	1633	56	0.194595	0.249701	3411	179	0.120712	0.168681	14006
Ejecución 5	12	0.605868	0.696799	1651	48	0.194483	0.241717	3329	117	0.107998	0.139353	13995
Media	10.2	0.61549	0.692782	1669.6	32.8	0.225708	0.257984	3375.2	127.2	0.116636	0.150724	14058.2

Tabla 60: Ejecuciones con Búsqueda Local Normal del Memético(10,0.1) con el 10 %

	Zoo				**Glass**				**Bupa**			
	Tasa_Inf	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*	*Tasa_Inf*	*Dist-Intra*	*Agr*	*T(ms)*
Ejecución 1	15	0.755254	0.815826	1751	34	0.250057	0.267758	3785	180	0.121571	0.14667	15574
Ejecución 2	34	0.668458	0.805752	1742	35	0.250023	0.268245	3818	213	0.120006	0.149707	16492
Ejecución 3	10	0.733622	0.774003	1729	43	0.249253	0.271639	3842	211	0.116426	0.145848	15618
Ejecución 4	9	0.718037	0.754379	1791	44	0.247638	0.270544	4155	322	0.121032	0.165932	15612
Ejecución 5	29	0.716594	0.833698	1763	114	0.204724	0.264074	3847	217	0.116365	0.146624	15626
Media	19.4	0.718393	0.796732	1755.2	54	0.240339	0.268452	3889.4	228.6	0.11908	0.150956	15784.4

Tabla 61: Ejecuciones con Búsqueda Local Normal del Memético(10,0.1) con el 20 %

Nos fijamos Ahora en el resto de medidas de las tablas para poder así analizar mejor el comportamiento en las gráficas de las Figuras 13, 14 y 15. Evidentemente, como tal muestran las gráficas el valor final medio de la función objetivo mejora en todos los casos. Vamos a analizar si esto ocurre porque disminuye la distancia-intra clúster o por la tasa de infactibilidad.

Infeasibility: En el Memético(10,1.0) mejora una auténtica barbaridad en todos los casos, el más notorio es el caso del conjunto de datos Glass, en donde se llega a reducir hasta en 5 veces. En el resto de conjuntos también se consigue una mejora bastante buena. En el Memético(10,0.1) se consiguen resultados buenísmos, mejora en todos los casos la nueva implementación con Búsqueda Local Normal. Los resultados obtenidos son muy similares a los conseguidos en el algoritmo de (solo) Búsqueda Local (Tablas 3 y 4).

Distancia Intra-Clúster: En el Memético(10,1.0) aquí se consigue mejorar en casi todos los casos, menos en el Zoo 20 %, aún así, las mejoras no son tan notorias como ocurría con el infeasibility. En el Memético(10,0.1) se mejora en todos los casos pero de forma muy poco sustancial.

Viendo el análisis de estos dos datos queda claro que la disminución del valor de la función objetivo ha venido de la mano del infeasibility, nuestra medida primera para valorar los resultados de los algoritmos. Por ello, podemos concluir que esta mejora con Búsqueda Local Normal mejora (y por mucho) a la implementada con Búsqueda Local Suave. Como último comentario, recordar que los tiempos son mucho mayores, así el algoritmo de Búsqueda Local Normal, quedaría en una posición por encima del Memético con Búsqueda Local Normal ya que las soluciones son por lo general de los Meméticos son peores y son muchísimo más lentas.

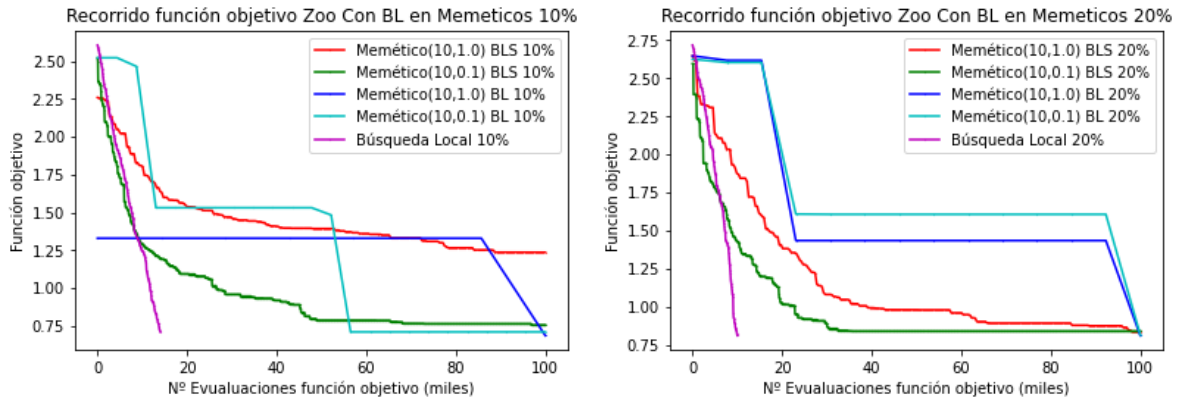

Figura 13: Gráficas de convergencia del valor de la función objetivo de algoritmos Meméticos con Búsqueda Local en Zoo

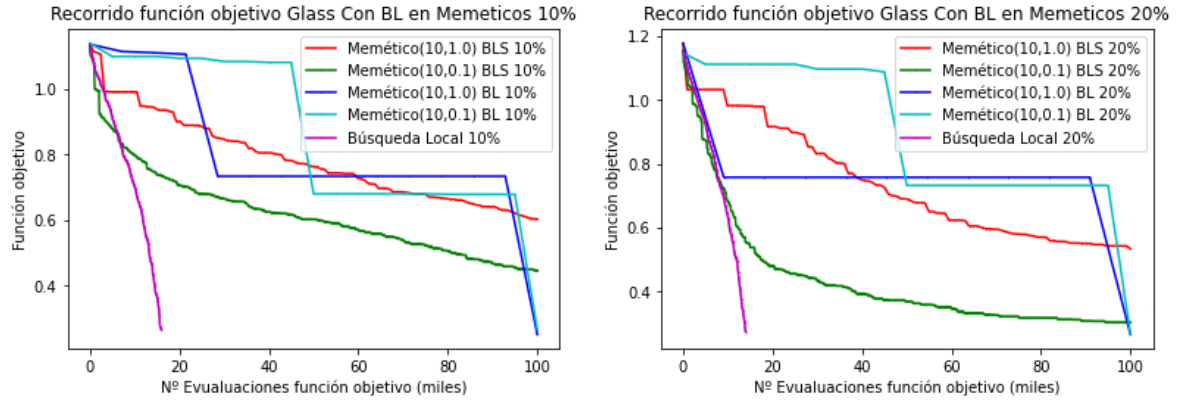

Figura 14: Gráficas de convergencia del valor de la función objetivo de algoritmos Meméticos con Búsqueda Local en Glass

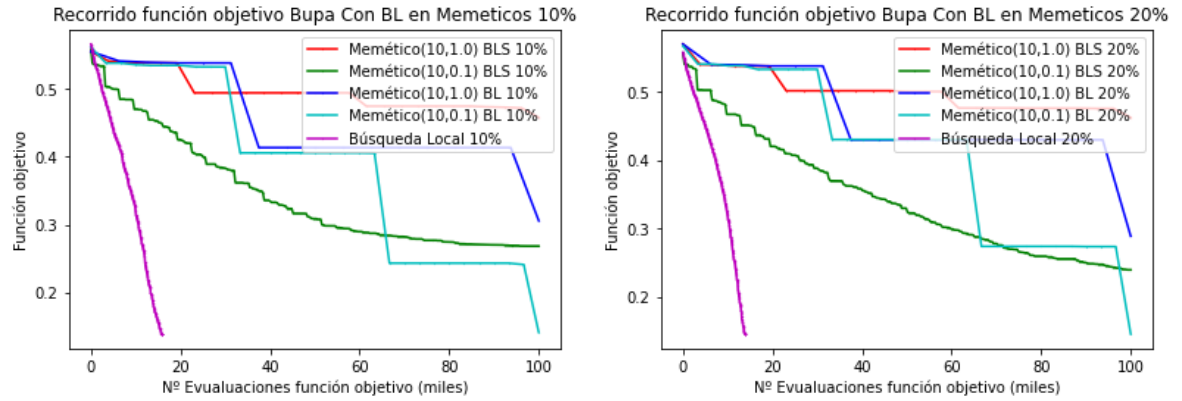

Figura 15: Gráficas de convergencia del valor de la función objetivo de algoritmos Meméticos con Búsqueda Local en Bupa

12. Bibliografía

No se ha utilizado ninguna fuente externa, utilizando únicamente la información proporcionada por la asignatura de Metaheurísticas de la Universidad de Granada.