UNIVERSIDAD DE GRANADA

Grupo 2 – Miércoles 17.30 - 19.30

Metaheurísticas — Técnicas de Búsqueda basadas en Poblaciones para el Problema del Agrupamiento con Restricciones

Alejandro Manzanares Lemus - 77393031D
alexmnzlms@correo.ugr.es

Abril 28, 2020

Índice general

§0.1: Cambios Práctica 1

Después de la correción de la práctica anterior, se han implementado varios cambios con respecto a las siguientes partes:

- Se ha modificado el cálculo de la infactibilidad y se han redistribuido las funciones de cada algoritmo.
- Función Objetivo Greedy: Ahora se tiene en cuenta el parámetro λ para calcular la función objetivo.
- Búsqueda Local: Se ha modificado el operador de vecino, debido a los errores en el cálculo de la solución. Ya no se factoriza el cálculo de la infacibilidad, por lo que el tiempo de ejecución del algoritmo ha aumentado. Además, había un fallo a la hora de calcular la Tasa_C de la solución. Cuando se recalculaba la desviación general para obtener la función objetivo, no se estaban recalculando los centroides de los clusters. Por ello, la solución obtenida era incorrecta y presentaba una altísima infactibilidad.

Ambos cambios se encuentran reflejados en los apartados que explican los algoritmos greedy y búsqueda local. Debido a estos cambios, he repetido las pruebas de la práctica anterior con las mismas semillas y los mismos conjuntos de datos:

- 1584565171
- 1584764782
- 1584565259
- 1584564539
- 1522565615

Resultados globales ORIGINALES en el PAR con 10 % de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**
K-medias Restringido Débil	0,70	36	37,10	0,005	1.599,44	190	1.789,04	0,153	0,87	6	6,87	0,007
Búsqueda Local	1,03	261	6,88	0,034	671,73	472	1.627,00	1,350	2,37	255	9,79	0,036

NUEVOS Resultados globales en el PAR con 10 % de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**
K-medias Restringido Débil	0,70	36	1,52	0,006	1.599,44	190	1.983,49	0,161	0,87	6	1,04	0,006
Búsqueda Local	0,60	0	0,60	0,570	1.718,06	188	2.099,27	37,253	0,85	0	0,85	0,607

Resultados globales ORIGINALES en el PAR con 20 % de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**
K-medias Restringido Débil	0,60	31	31,60	0,005	1.659,37	212	1.870,97	0,351	0,85	4	4,45	0,004
Búsqueda Local	1,37	511	7,11	0,036	715,42	1.015	1.705,59	1,403	3,50	509	10,89	0,044

NUEVOS Resultados globales en el PAR con 20 % de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**
K-medias Restringido Débil	0,60	31	0,95	0,007	1.659,37	212	1.873,68	0,386	0,85	4	0,90	0,006
Búsqueda Local	0,60	0	0,60	0,673	1.667,17	394	2.066,21	40,36	0,85	0	0,85	0,676

Apartado 1:
Descripción del problema

El problema elegido es el **Problema del Agrupamiento con Restricciones**, a partir de ahora **PAR**, es una variante del problema de agrupamiento clásico. El problema de agrupamiento clásico consiste en que dado un conjunto X de datos con n características, hay que encontrar una partición C de tal manera que se minimice la desviación general de cada $c_i \in C$.
En la variante **PAR**, se introduce el concepto de restricción. Nosotros utilizamos las restricciones de instancia que pueden ser dos tipos:

- Restricciones **ML**(*Must-link*): Dos elementos $x_i \in X$ que posean una restricción ML, deben pertenecer al mismo $c_i \in C$.
- Restricciones **CL**(*Cannot-link*): Dos elementos $x_i \in X$ que posean una restricción CL, deben pertenecer a $c_i \in C$ distintos.

Además, estas restricciones serán débiles, es decir, el objetivo es minimizar tanto el número de restricciones incumplidas — una solución factible puede incumplir restricciones — como la desviación general de cada $c_i \in C$.

§1.1: Formalización de los datos

- Los **datos** se representan en una matriz $i \times n$ siendo i el número de datos que tenemos y n el número de características que tiene cada $x_i \in X$
  $\vec{x_i} = \{x_{i0}\ ...\ x_{in}\}$ donde cada $x_{ij} \in \mathbb{R}$
- Una **partición** C consiste en un conjunto de k clusters. $C = \{c_0\ ...\ c_k\}$. Cada c_i contiene un conjunto de elementos x_i. El número de elementos de c_i es $|c_i|$ y normalmente un cluster c_i tiene asociada una etiqueta l_i — Esto no lo utilizaremos en la implementación del problema —.
- Para cada cluster c_i se puede calcular su **centroide** $\vec{\mu_i}$ como el promedio de los elementos $x_i \in c_i$.
  $\vec{\mu_i} = \frac{1}{|c_i|} \sum_{\vec{x_j} \in c_i} \vec{x_j}$.
- La **distancia media intra-cluster** $\bar{c_i}$ se define como la media de las distancias de cada $x_i \in c_i$ a su centroide μ_i.
  $\bar{c_i} = \frac{1}{|c_i|} \sum_{\vec{x_j} \in c_i} \|\vec{x_j} - \vec{\mu_i}\|_2$
- La **desviación general** de la partición C se calcula como la media de las distancias medias intra-cluster $\bar{c_i}$.
  $\bar{C} = \frac{1}{k} \sum_{c_i \in C} \bar{c_i}$.
- El **conjunto de restricciones totales** R se calcula como la unión entre el conjunto de restricciones ML y el conjunto de restricciones CL.
  $|R|$ es el número de restricciones total $|R| = |ML| + |Cl|$.
- La **infactibilidad** — *infeasibility* — se calcula como el número de restricciones que incumple una partición C del conjunto X dado un conjunto de restricciones R. Se define $V(\vec{x_i}, \vec{x_j})$ como una función que devuelve 1 si la pareja $(\vec{x_i}, \vec{x_j})$ incumple alguna restricción.
  $infeasibility = \sum_{i=0}^{n} \sum_{j=i+1}^{n} V(\vec{x_i}, \vec{x_j})$

Apartado 2:
Descripción de los algoritmos empleados

§2.1: Función objetivo

Todos los algoritmos implementados comparten la misma función objetivo: $F_{objetivo} = \bar{C} + (infeasibility \cdot \lambda)$

El valor esta función se calcula como la suma de la **desviación general** ($\bar{C}$) y la infactibilidad de la partición (*infeasibility*) escalada por un parámetro (λ). Este parámetro tiene la función de dar relevancia al término *infeasibility*. Si se establece correctamente, primero da prioridad a reducir las restricciones incumplidas por encima de la desviación general.
Este parámetro se calcula como: $\lambda = \frac{\lceil D \rceil}{|R|}$. Siendo D la distancia entre los dos puntos más lejanos y R el número total de restricciones.
El objetivo de los algoritmos descritos a continuación es minimizar el valor de esta función objetivo.

- **K-medias Restringido Débil**: Algoritmo de heurística greedy que se basa en asignar un elemento al cluster de cuyo centroide se encuentre más cercano de manera iterativa, hasta obtener el mejor ajuste posible.
- **Búsqueda Local**: Algoritmo de búsqueda basado en trayectorias, se vale de la exploración de vecindarios para obtener la mejor solución posible.
- **Algoritmos Genéticos**: Algoritmos de búsqueda basado en poblaciones. En esta práctica implementamos un esquema generacional y uno estacionario, y los combinamos con dos operadores de cruce, uno uniforme y otro de segmento fijo, obteniendo así 4 algoritmos distintos.
- **Algoritmos Meméticos**: Algoritmos de búsqueda basado en poblaciones. Se basa en la implementación de un algortimo genético con un esquema generacional, con la diferencia de que cada determinado número de generaciones, se aplica una búsqueda local a un determinado número de cromosomas. En esta práctica se implementan 3 versiones que se desarrollarán más adelante.

§2.2: Representación de los datos

Los datos comunes a ambos algoritmos se representan de la siguiente manera:

- Para los **datos** utilizo una matriz *posiciones* de números reales de dimensión $i \times n$.

```
double: matriz[i][n] posiciones
```

- Para los **centroides** de cada cluster utilizo una matriz *centroides* de dimensión $k \times n$.

```
double: matriz[k][n] centroides
```

- Para las **restricciones**, he elegido no utilizar las representación en forma de matriz, porque es muy costoso recorrerla secuencialmente, y la representación en forma de lista no te permite acceder a un elemento en concreto, por eso he decidido utilizar un map *restricciones*. Este tipo de estructura se puede recorrer secuencialmete de forma eficiente y ademas, existe un operador de búsqueda para poder acceder a un elemento concreto. El map se compone de dos elementos: la clave y el valor. La clave es la pareja de elementos x_i, x_j y el valor es 1 si la restricción es de tipo **ML** o -1 si es de tipo **CL**.

```
Pareja(int,int),int: map restricciones
```

- Los elementos x_i que pertenecen a los distintos **clusters** los almaceno en una matriz *clusters* de dimensión $k \times i$.

```
double: matriz[k][i] clusters
```

- Finalmente la **partición** C la represento en un vector de enteros *solucion*, en los que posición del vector i indica el elemento x_i y el contenido de la posición i, *solucion[i]* indica el cluster c_i al que pertenece.

```
int: vector[i] solucion
```

§2.3: Operadores comunes

Hay una serie de operadores que son comunes a los dos algoritmos, los describo a continuación:

- Operador **calcular_centroide(cluster i)** Calcula el centroide de un cluster i:

```
Para cada característica u del centroide i:
   u = 0

Para cada elemento j del cluster i:
   Para cada característica c, u del elemento j, centroide i:
      u += 1/k * c
```

- Operador **distancia_intracluster(cluster i)**: Calcula la distancia intracluster de un cluster i.

```
Para todos los clusters:
   d_intracluster = 0

Para cada elemento j del cluster i:
   Para cada característica c, u del elemento j, centroide i :
      d_intracluster += 1/k * abs(c - u) * abs(c - u)
```

- Operador **desviación_general()**: Calcula la desviación general del problema.

```
desv_gen = 0

Para cada distancia_intracluster i del cluster i:
   desv_gen += 1/k * i
```

- Operador **calcular_lambda()**: Calcula el parámetro λ

```
lambda = 0
double: d, d_max = 0
int: cluster

Para cada elemento e en posiciones:
   cluster = Cluster al que pertenece e
   Para cada número n de 0...k:
      Si n != cluster:
         Para cada elemento c de cluster n:
            d = Distancia entre n y e
            Si d es mayor que d_max:
               d_max = d

lambda = d_max / restricciones
```

- Operador **calcular_infact_sol(solucion)**: Calcula la infacibilidad de un vector solución dado

```
int: infactibilidad = 0
Para cada elemento i de la solucion:
   Desde j = i + 1 hasta el fin de solución:
      Si la pareja (i,j) pertence al set de restricciones:
         Si j igual -1 Y elemento i igual elemento j de sol:
            infactibilidad++
         Si j igual 1 Y elemento i distinto elemento j de sol:
            infactibilidad++

Devolver infactibilidad
```

Apartado 3:
Métodos de búsqueda

§3.1: Búsqueda Local

§3.1.1: Datos propios

En la búsqueda local es necesario controlar de alguna manera que vecinos se han generado y cuáles puede generar aún, por tanto, almaceno en un set el **vecindario** correspondiente a la solución que se está evaluando actualmente. El vecindario no es más que las posibles cambios que se pueden realizar en el vector solución, partiendo de un estado determinado.

```
1    Pareja(int,int): set vecindario
```

§3.1.2: Descripción del algoritmo

Lo primero es general la solución inicial de la que partirá la búsqueda local. Esta solución es completamente aleatoria. Una vez generada, generamos también el vecindario. Almacenamos los valores de función objetivo e infactibilidad así como el vector solución en variables auxiliares.
Comenzamos a generar vecinos, cada vez que se explora uno nuevo, se elimina del vecindario y si la función objetivo actual no mejora a la que almacenamos previamente, descartamos la función objetivo y la infactibilidad, recuperando las que teníamos almacenadas y generamos otro vecino.
Continuaremos así hasta que encontremos un vecino cuya función objetivo mejore a la almacenada, volvemos a guardar una copia de la infactibilidad, la función objetivo así cómo una copia del vector solución.
Seguiremos generando vecinos hasta que o bien las evaluaciones superen las 100000 o bien que no queden vecinos que explorar — el vecindario esta vacío—.
Como consideración, antes de salir del bucle principal leemos el estado de la solución para el ultimo vecino generado.

§3.1.3: Pseudocódigo del algoritmo

```
double: f_objetivo_ant, infactibilidad_ant
int: vector[] solucion_ant
int: i

solucion_inicial()
generar_vecindario()

f_objetivo_ant = f_objetivo
solucion_ant = solucion
infactibilidad_ant = infactibilidad

Hacer:
   generar_vecino()
   i++

   Si nueva f_objetivo es menor que f_objetivo_ant:
      f_objetivo_ant = f_objetivo
      solucion_ant = solucion
      infactibilidad_ant = infactibilidad
      generar_vecindario()

   Si no:
      solucion = solucion_ant
      infactibilidad = infactibilidad_ant

   Si no quedan vecinos que generar:
      Leer solucion actual

Mientras: i menor que 100000 Y quedan vecinos que explorar
```

§3.1.4: Operadores propios

- Operador **solucion_inicial()**: Genera una solucion inicial aleatoria. Rellena la matriz de clusters aleatoriamente asegurándose de que al menos hay mínimo un elemento en cada fila de la matriz y después carga estos datos en el vector solucion.

```
   int: vector[] index
   int: matriz[][] c

   Para número n de 0...i:
      Añadir n a index

   Ordenar index aleatoriamente

   Para cada fila j en c:
      Añadir a j elemento de index
      Pasar al siguiente elemento

   Para los elementos restantes en index:
      Añadir a fila aleatoria 0...k de c elemento de index
      Pasar al siguiente elemento

   Para cada número n de 0...k:
      Para los elementos e de la fila n de c:
         Añadir n a solucion en la posición c[n][e]

   Para cada número n de 0...k:
      calcular_centroide(n)
```

- Operador **generar_vecindario()**: Genera el vecindario correspondiente a una solución determinada. Genera todos los posibles cambios que se pueden hacer en el vector solucion y almacena las parejas (elemento, cluster) en el set vecindario.

```
Vaciar vecindario

Para cada elemento e en posiciones:
   Para cada número n de 0...k:
      Si n es distinto al cluster al que pertenece e
      Y en el cluster al que pertenece e hay al menos 1 elemento:
         Insertar en vecindario Pareja (e,n)
```

- Operador **generar_vecino()**: Generar un vecino se basa en generar una pareja aleatoria (elemento, cluster) y comprobar si existe en el vecindario, si no existe, se prueba con otra. Si existe, se elimina el elemento de la matriz de clusters, se resta a la infactibilidad total, la infactibilidad producida por la antigua asignación y se suma la infactibilidad producida por la nueva, por último se añade el elemento a la fila correspondiente de la matriz de cluster y se actualiza el vector solucion.

```
bool: salir = falso
int: pos, n, c

Mientras no salir Y quedan vecinos que explorar:
   salir = falso
   pos = número aleatorio 0...i
   n = número aleatorio 0...k
   Buscar Pareja (pos,n) en vecindario

   Si Pareja (pos,n) existe en vecindario:
      salir = verdadero
      Borrar Pareja (pos,n) de vecindario

      Para cada cluster j en el conjunto de cluster:
         Vaciar j

      Para cada elemento e de la solucion:
         Introducir e en el cluster solucion[e]

      solucion[pos] = n

      Para cada cluster j en el conjunto de cluster:
         calcular_centroide(j)

      desviacion_general()
      infactibilidad_solucion()
      f_objetivo = desv_gen + (infactibilidad*lambda)
```

§3.2: Algoritmos genéticos

§3.2.1: Datos propios

En los algoritmos de búsquedas basados en poblaciones necesitamos almacenar un conjunto de soluciones, al que llamaremos población. A cada solución de la población la llamaremos cromosoma. Para cada iteración del algoritmo, diremos que se crea una nueva generación de cromosomas que sustituye a los cromosomas anteriores de la población. A cada posición dentro de un cromosoma se la denomina gen.
Almacenamos en una variable el número de cromosomas que tiene una población. Además, almacenaremos cada generación de la población en una matriz. También debemos tener otra matriz para almacenar los cromosomas seleccionados en cada generación. Las evaluaciones las guardaremos en una variable que almacena el número de llamadas que se hace a la función que evalúa una solución.
Para ahorrar evaluaciones, a cada cromosoma, se le asocia un valor de su funcion objetivo que se calcula una sola vez. Estos valores se almacenan en vectores. También se almacena en un vector la mejor solución encontrada hasta el momento y su valoración.

```
1    int: poblacion;
2    int: ind_eval;
3    int: matriz generacion;
4    int: matriz seleccion;
5    double: f_generacion;
6    double: f_seleccion;
7    int: mejor_generacion;
8    double: f_mejor_generacion;
```

§3.2.2: Descripción del algoritmo

Los algoritmos se nombran de la siguiente manera **AG[Esquema]_[Cruce]**.
Se han implementado las versiones **AGG_UN**, **AGG_SF**, **AGE_UN**, **AGE_SF**.
Para facilitar la descripción, se describirán por separado los operadores de cruce de los esquemas de selección y reemplazo.

Operadores de cruze

Hay dos operadores de cruce:

- **Operador de cruce uniforme [UN]**: El operador de cruce uniforme, se basa en escoger $n/2$ números aleatorios en el rango $[0, n-1]$. De uno de los dos padres, seleccionamos los genes que coincidan con estos números y los copiamos en la descendencia. Los genes que quedan por asignar se obtienen del segundo padre.
- **Operador de segmento fijo [SF]**: El operador de segmento fijo, se basa en generar dos números aleatorios r y v en el rango $[0, n-1]$. Seleccionamos uno de los padres y copiamos en la descendencia el rango de genes $[r, ((r+v)\, mod\, n)-1]$. El resto de genes se determinan de la misma manera que en el operador de cruce uniforme.
  Este operador esta sesgado. Si seleccionamos el segmento de datos del padre que mejor valor de la función objetivo tenga, estamos **favoreciendo la explotación** y si lo hacemos al revés, estaremos **favoreciendo la exploración**. En este caso, se favorece la **explotación**.

Esquemas de reemplazo y selección

Hay dos esquemas de reemplazo y selección:

- **Generacional** [**G**]: El esquema generacional funciona de la siguiente manera: seleccionamos tantos cromosomas como cromosomas tenga la población —50 en nuestro caso—. La forma de selección se basa en realizar un torneo binario, es decir, enfrentar a dos cromosomas aleatorios y añadir a la selección el que mejor valor tenga en su función objetivo. Una vez tenemos la selección, generamos tantos cromosomas como cromosomas haya en nuestra selección, utilizando el operador de cruce que corresponda. En el esquema generacional hay un 70 % de probabilidad de cruce, así que aplicando esperanza matemática, solo el 70 % de las parejas cruzan, y el otro 30 % se mantiene igual. Una vez se han cruzado los cromosomas de la población, se aplica el operador de mutación. Cada gen tiene un 0.001 % de probabilidad de mutación. Si un gen muta, cambia aleatoriamente su valor a otro posible cluster. Cuando todos los cromosomas han mutado, se aplica el operador de reparación que arregla las posibles infactibilidades en las soluciones —haber dejado un cluster vacío, por ejemplo—. Finalmente, se aplica el elitismo. El elitismo se basa en que, después de haber obtenido una nueva generación de cromosomas, se comprueba que la mejor solución de la generación anterior este presente en la nueva generación. Si no está presente, se sustituye por el peor cromosoma de la generación.
- **Estacionario** [**E**]: El esquema estacionario funciona de forma similar al generacional. En cuanto a la selección, en el esquema estacionario solo se seleccionan 2 cromosomas de cada generación, y como es lógico, su probabilidad de cruce es 100 %. A esta selección , también se le aplica el operador de mutación y de reparación —En las mismas condiciones que en el esquema generacional—. A la hora de devolver estos cromosomas a la población, se compara el mejor de la selección contra el peor de la población, y se añade a la población el mejor de los dos. Después, se comparan el segundo mejor contra el segundo peor y se realiza el mismo proceso.

El criterio de parada del algortimo es no superar las 100000 evaluaciones.

§3.2.3: Pseudocódigo del algoritmo

```
int: generacion = 0;

Generamos una poblacion inicial
//Igual que en la busqueda local.

Hacer:
   generacion++;
   Seleccionar el mejor de la población y guardarlo

   Aplicar seleccion

   Aplicar operador de cruce

   Si generacional:
      Aplicar elitismo y generacional
   Si estacionario:
      Aplicar estacionario

Mientras evaluaciones menor que 100000
```

§3.2.4: Operadores propios

- Operador **operador_seleccion(t)**: Selecciona t elementos de la población y los añade a la selección.

```
seleccionados = torneo_binario(t)
//Seleccionamos t elementos mediante torneo binario

Para i = 0 hasta fin de seleccionados:
   Añadir a seleccion el elemento i de poblacion
   Añadir a f_seleccion el elemento i de f_poblacion
```

- Operador **operador_cruce_uniforme(p1,p2)**: Aplica el operador de cruce uniforme a p1 y p2 para generar un nuevo vector solución.

```
int: genoma = tamaño p1 / 2
int: gen
int: vector genes

Para i = 0 hasta genoma:
   gen = Numero aleatorio
   Si gen no esta en genes:
      Añadir gen a genes

int: vector descendiente

Para i = 0 hasta tamaño p1:
   Si i esta en genes:
      Añadir a descendiente gen i de p1
   Si no:
      Añadir a descendiente gen i de p2

   Devolver descendiente
```

- Operador **operador_cruce_segmento(p1,f_p1,p2,f_p2)**: Aplica el operador de cruce por segmento fijo a p1 y p2 para generar un nuevo vector solución. Las valoraciones de p1 y p2 se pasan como parámetro para ahorrar evaluaciones innecesarias.

```
int: tam = tamaño p1
int: r = Numero aleatorio entre 0 y tam
int: v = Numero aleatorio entre 0 y tam
int: set genes_dominantes
int: genoma = (tam - v) / 2
int: gen
int: set genes

Para i = 0 hasta v:
   Añadir r+i%tam a genes_dominantes
Para i = 0 hasta genoma:
   gen = Numero aleatorio entre 0 y tam
   Si gen no se encuentra en genes Y no se encuentra en genes dominantes:
      Añadir gen a genes

int: vector mejor;
int: vector peor;

Escoger como mejor p1 o p2 segun f_p1 y f_p2
Escoger como peor el otro

int: vector descendiente
Para i = 0 hasta tam:
   Si i se encuentra en genes_dominantes:
      Añadir a descendiente, gen i del mejor o peor padre,
      segun se favorezca la exploración o la explotación
   Si no:
      Si i se encuentra en genes:
         Añadir a descendiente, gen i de p1
      Si no:
         Añadir a descendiente, gen i de p2

Devolver descendiente
```

- Operador **reparar_solucion(sol)**: Repara las infactibilidades presentes en un vector solución.

```
bool: reparacion = falso
int: aleatorio
int: tam_sol = tamaño sol - 1
int matriz c
int: vector c_vacio

Para i = 0 hasta tamaño de sol:
   Añadir i a c[sol[i]]

Para i = 0 hasta numero de cluster:
   Si c[i] esta vacio:
      reparacion = verdadero
      Añadir i a c_vacio


Si reparacion:
   Recorrer el vector c_vacio:
      aleatorio = Numero aleatorio entre 0 y tam_sol
      Mientras haya al menos un elemento en c[sol[aleatorio]]:
         aleatorio = Numero aleatorio entre 0 y tam_sol

      Eliminar aleatorio de c[sol[aleatorio]]
      Actualizar sol[aleatorio] a elemento actual de c_vacio
      Añadir aleatorio a c[sol[aleatorio]]
```

- Operador **mutar_solucion(sol)**: Aplica el operador de mutación a una solución.

```
int: aleatorio
int: mutación
int: p_mutacion = 1000

Para i = 0 hasta tamaño de sol:
   mutacion = Numero aleatorio entre 1 y p_mutacion
   Si mutacion igual a 1:
      aleatorio = Numero aleatorio entre 0 y n_cluster
      sol[i] = aleatorio
```

- Operador **aplicar_generacional()**: Aplica el esquema generacional a una población. Sustituimos la generación actual por la selección y conservamos el elitismo.

```
generacion = seleccion
f_generacion = f_seleccion

conservar_elitismo()

limpiar seleccion
limpiar f_seleccion
```

- Operador **conservar_elitismo()**: Comprueba si la mejor solución generada en un generación se encuentra presente en la siguiente generación, y si no es así, la introduce en la generación.

```
int: sustituir

Si mejor_generacion esta en seleccion:
   sustituir = seleccionar el peor de la generacion
   generacion[sustituir] = mejor_generacion;
   f_generacion[sustituir] = evaluar_solucion(mejor_generacion);
```

- Operador **aplicar_estacionario(n)**: Aplica el esquema estacionario a una población. Enfrenta al mejor seleccionado contra el peor de la generación iterativamente e introduce en la generación el mejor de ambos.

```
int: matriz estacion
double: f_estacion
int: matriz peores
double: f_peores
int: index_peor
double: max, min
int: i_max, i_min

Para i = 0 hasta n:
   index_peor = seleccionar el peor de la generacion

   Añadir generacion[index_peor] a peores
   Añadir f_generacion[index_peor] a f_peores

   Borrar generacion[index_peor] de seleccion
   Borrar f_generacion[index_peor] de f_seleccion

int: vector comprobado_peor
int: vector comprobado_selec

Para i = 0 hasta n:
   Para j = 0 hasta n:
      Si f_seleccion[j] menor que min Y j no esta en comprobado_selec:
         min = f_seleccion[j]
         i_min = j
   Para j = 0 hasta n:
      Si f_peores[j] mayor que max Y j no esta en comprobado_peor:
         max = f_peores[j]
         i_max = j

   Si f_seleccion[i_min] es menor que f_peores[i_max]:
      Añadir seleccion[i_min] a estacion
      Añadir f_seleccion[i_min] a f_estacion
   Si no:
      Añadir peores[i_min] a estacion
      Añadir f_peores[i_min] a f_estacion

   Añadir i_min a comprobado_selec
   Añadir i_max a comprobado_peor

Para i = 0 hasta n:
   Añadir estacion[i] a generacion
   Añadir f_estacion[i] a f_generacion

Limpiar seleccion
Limpiar f_seleccion
```

§3.3: Algoritmos meméticos

§3.3.1: Datos propios

Los algortimos meméticos se basan en la unión de los algoritmos genéticos con la búsqueda local, por lo que comparten los mismos datos propios.

```
int: poblacion;
int: ind_eval;
int: matriz generacion;
int: matriz seleccion;
double: f_generacion;
double: f_seleccion;
int: mejor_generacion;
double: f_mejor_generacion;
```

§3.3.2: Descripción del algoritmo

Los algoritmos meméticos implementados en esta práctica, están construidos sobre un algoritmo genético, con esquema generacional y que utiliza el operador de cruce uniforme, es decir, **AGG_UN**, puesto que es el algoritmo genético generacional que mejores resultados ha ofrecido durante las pruebas. Se implementan 3 versiones de algoritmos meméticos:

- **AM_10-1.0**: Esta versión aplica una BLS cada 10 generaciones a todos los elementos de la población.
- **AM_10-0.1**: Esta versión aplica una BLS cada 10 generaciones a 1 elemento de la población.
- **AM_10-0.1mej**: Esta versión aplica una BLS cada 10 generaciones al mejor elemento de la población.

La Búsqueda Local Suave (**BLS**) es una variación de la búsqueda local que se centra en perder el menor número posible de evaluaciones. Existe un número máximo de fallos que puede cometer la BLS. La BLS recorre de manera aleatoria los genes de un cromosoma. Si puede mejorar el cromosoma, aplica el mejor cambio posible, y si no puede mejorarlo más, acumula un fallo. La BLS termina cuando se han recorrido todos los genes o cuando se ha superado el máximo de fallos.

§3.3.3: Pseudocódigo del algoritmo

```
int: generacion = 0;

Generamos una poblacion inicial
//Igual que en la busqueda local.

Hacer:
   generacion++;
   Seleccionar el mejor de la población y guardarlo

   Aplicar seleccion

   Aplicar operador de cruce

   Aplicar elitismo y generacional

   Si generacion multiplo de 10:
      Aplicar BLS

Mientras evaluaciones menor que 100000
```

§3.3.4: Operadores propios

- Operador **mejor_valor(sol,f_sol,i)**: Este operador recibe una solución, su valoración y el gen que se quiere mejorar. Si existe una majora para este gen, la realiza y devuelve verdadero. Si no, devuelve falso.

```
double: min = f_sol
int: cluster_min
int: vector sol_comp
double: f_sol_comp
int: matriz c

Para j = 0 hasta tamaño de la solución:
   Añadr j a c[sol[j]]

Para j = 0 hasta tamaño de la solución:
   sol_comp = sol
   Si sol[i] distinto de j Y tamaño de c[j] mayor que 1:
      sol_comp[i] = j
      f_sol_comp = evaluar_solucion(sol_comp)
      Si min > f_sol_comp:
         cluster_min = j
         min = f_sol_comp

Si cluster_min no se ha asignado a nada:
   Devolver falso
Si no:
   sol[i] = cluster_min
   f_sol = evaluar_solucion(sol)
   Devolver verdadero
```

- Operador **busqueda_local_suave()**: Aplica la búsqueda local suave a una solución determinada.

```
int: i = 0
int: tam = tamaño solucion
int: vector rsi = {0,1,2....tam-1}
bool: mejora
int: fallos
int: umbral = 0.1*tam

Ordenar rsi aleatoriamente

mejora = verdadero
Mientras mejora O fallos menor que umbral Y i < tam:
   Si mejor_valor(sol,f_sol,rsi[i])
      mejora = verdadero
   Si no:
      fallos++

   i++
```

Apartado 4:
Descripción de los algoritmos de comparación

§4.1: K-medias Restringido Débil

§4.1.1: Descripción del algoritmo

El funcionamiento de este algoritmo funciona de la siguiente manera:
Se presupone que los valores de los centroides antes de comenzar con la ejecución son aleatorios.
Se establece un orden aleatorio para recorrer los nodos, pero que se mantenga en todas las iteraciones del algoritmo. Por cada iteración, se asigna a cada elemento $x_i \in X$ un cluster c_i. El criterio de asignación es siempre el mismo: se asigna al cluster que menos infactibilidad provoque y en caso de todas las asignaciones provoquen la misma infactibilidad, se asignará al cluster cuyo centroide μ_i sea más cercano. El algoritmo iterara hasta que no se produzca un cambio en el estado de la solución — en la matriz de clusters — y entonces terminará.
Es importante remarcar que es posible que el algoritmo se quede iterando de manera infinita si entra en un ciclo de asignaciones, esto se puede evitar eligiendo la semilla de generación de números aleatorios de manera correcta.

§4.1.2: Pseudocódigo del algoritmo

```
int: i = 0
bool: cambio_c
int: vector[] rsi
double: matriz[][] solucion_ant

Guardar en solucion_ant la matriz cluster

Para cada número de 0...i:
   Añadir número a vector rsi
Ordenar rsi aleatoriamente

Hacer:
   cambio_c = falso
   Para cada indice j en rsi:
      asignar_cluster(j)
   Para cada cluster c:
      Si fila c de solucion_ant es distinta a la fila c de clusters:
         calcular_centroide(c)
         cambio_c = verdadero
   Guardar en solucion_ant la matriz cluster

   Si cambio_c:
      Vaciar los clusters

   i++
Mientras: cambio_c

desviacion_general()
f_objetivo = desv_gen + infactibilidad
```

§4.1.3: Operadores propios

- Operador **asignar_cluster(elemento n)**: Este operador asigna el elemento n a un cluster siguiendo el criterio de asignación: De los que menos infactibilidad provoquen, el que tenga la menor distancia.

```
int: c, r_min, d_min, d = 0
Pareja(int,int): vector[] r

Para cada cluster i:
   Añadir la pareja (restricciones_incumplidas(n,i) , i) a r

Ordenar r en orden ascendente

r_min = r[0]
d_min = Infinito
Para cada indice j de 0...tamaño(r)
   d = distancia_nodo_cluster(n, r[j].segundo)
   Si d < d_min:
      d_min = d
      c = r[j].segundo

infactibilidad += r_min
Añadir n al cluster c
```

- Operador **restricciones_incumplidas(elemento n, cluster c)**: Este operador calcula el número de restricciones incumplidas que provoca la asignación del elemento n al cluster c.

```
incumplidas = 0

Para cada elemento i del cluster c:
   Buscar en restricciones la pareja (n,elemento i)
   Si existe Y valor es -1:
      incumplidas++

Para los cluster del conjunto distintos de c:
   Para cada elemento i del cluster j:
      Buscar en restricciones la pareja (n,elemento i)
      Si existe Y valor es 1:
         incumplidas++

Devolver incumplidas
```

Apartado 5: Procedimiento

§5.1: Estructura de datos

La implementación de la práctica se ha llevado a cabo en c++.
Para la estructura de datos he optado por una sola clase, llamada CCP — Constrained Clustering Problem — en la que están todos los datos necesarios para realizar el problema:

```
int n_cluster;
std::vector<std::vector<double>> posiciones;
std::vector<std::vector<double>> centroides;
std::map<std::pair<int,int>,int> restricciones;
std::vector<double> d_intracluster;
std::vector<int> solucion;

std::set<std::pair<int,int>> vecindario;
std::vector<std::vector<int>> clusters;

int poblacion;
int ind_eval;
std::vector<std::vector<int>> generacion;
std::vector<std::vector<int>> seleccion;
std::vector<double> f_generacion;
std::vector<double> f_seleccion;
std::vector<int> mejor_generacion;
double f_mejor_generacion;

double desv_gen;
double infactibilidad;
double lambda;
double f_objetivo;
```

He utilizado las clases map, set y vector de la STL.
Las restricciones se almacenan en un map debido a que al ser una estructura de datos de la STL, es posible recorrerlo de forma secuencial con un iterador y ademas, cuenta con el operador find, que permite saber si existe determinada combinación de elementos x_i y si la restricción es de tipo ML o CL. Por tanto me pareció mejor implementación que la propuesta de matriz y lista.
El vecindario se utiliza como una manera para poder saber cuando ha terminado el algoritmo de búsqueda local y no volver a explorar vecinos que ya he explorado previamente. Utilizo un set porque a diferencia del vector, no permite que existan parejas (elemento, cluster) duplicadas y ademas estas se ordenan automáticamente en orden ascendente.
Las matrices de generación y selección se implementan como vectores de vectores de enteros y las funciones objetivo asociadas a los cromosomas se almacenan en vectores de doubles.
Los operadores de los algoritmos descritos anteriormente se implementan como métodos de la clase CCP.

§5.2: Guía de Uso

El programa se compila utilizando la orden **make**.
Para ejecutar el programa es necesario ejecutar el archivo **BIN/clustering_exe** [*semilla*] [*conjunto*] [*algoritmo*] [*verbose*]. Estos parámetros se configuran de la siguiente manera:

- [*semilla*]: Es el número de semilla que se quiere ejecutar.
  - $[1-5]$: Ejecutan desde la primera semilla hasta la que se pase como argumento
  - [0]: Si se introduce el 0, el programa entra en modo gráfico. Esto significa que genera un output de datos para poder pintar una gráfica de la evolución del valor de la función objetivo con el paso de las generaciones —Solo funciona para algoritmos genéticos y meméticos—.
- [*conjunto*]: Numero del conjunto que se quiere ejecutar.
  - [*Rand*]: 1 para 10 % y 5 para 20 %
  - [*Iris*]: 2 para 10 % y 6 para 20 %
  - [*Ecoli*]: 3 para 10 % y 7 para 20 %
  - [*Newthyroid*]: 4 para 10 % y 8 para 20 %
  - [0]: Ejecuta todos los conjuntos anteriores
- [*algortimo*]: Algortimo que se quiere utilizar:
  - [*G*]: K-medias Restringido Débil.
  - [*BL*]: Búsqueda local.
  - [*AGG_UN*]: Algoritmo genético con operador de cruce uniforme y esquema generacional.
  - [*AGG_SF*]: Algoritmo genético con operador de cruce segmento fijo y esquema generacional.
  - [*AGE_UN*]: Algoritmo genético con operador de cruce uniforme y esquema estacionario.
  - [*AGE_SF*]: Algoritmo genético con operador de cruce segmento fijo y esquema estacionario.
  - $[AM\_10-1{,}0]$: Algoritmo memético que aplica la búsqueda local cada 10 generaciones a todos los cromosomas
  - $[AM\_10-0{,}1]$: Algoritmo memético que aplica la búsqueda local cada 10 generaciones al 10 % de los cromosomas
  - $[AM\_10-0{,}1mej]$: Algoritmo memético que aplica la búsqueda local cada 10 generaciones al 10 % de los mejores cromosomas
  - [*SEM*]: Activa el modo búsqueda de semillas.
- [*Verbose*]: Muestra por pantalla la traza de ejecución del algoritmo.

Como ejemplo básico, para obtener los resultados que se muestran a continuación, se debe ejecutar:
BIN/clustering_exe 5 0 [*algoritmo*]

La estructura de ficheros es la siguiente:

- cc.h: Cabecera de la clase CCP.
- cc.p: implementación de los métodos de la clase CCP.
- main.cpp: Implentación de la ejecución de los algoritmos greedy y BL.
- random.h y random.cpp : Cabeceras e implementación del generador de aleatorios.

Apartado 6:
Experimentos y análisis de resultados

§6.1: Semillas

Para la ejecución de los algoritmos se han seleccionado las siguientes semillas, utilizando un algoritmo para probar que no producen ciclos en ninguna de las ejecuciones del algoritmo greedy para ninguno de los conjuntos de datos. Toda semilla que en menos de 1000 iteraciones del algoritmo greedy no obtenga resultado es rechazada. Las semillas se han cambiado respecto a la práctica anterior debido al cambio de conjuntos de datos:

- 2024614690
- 2024676296
- 2024677261
- 2024740484
- 2024740899

El código utilizado para encontrar semillas se encuentra en la función **buscar_semilla()** que se incluye en el fichero main.cpp

§6.2: Análisis

Este análisis se va a estructurar en 3 partes. En primer lugar comentaré los resultados de los data sets de *Iris* y *Rand*. Estos se comentan en un mismo apartado debido a la gran similitud que existe entre ellos. Posteriormente se analizarán los resultados del data set *Ecoli*, que más complejidad y dificultad en el ajuste presenta, con clara diferencia. Por último se analizarán los resultados del nuevo data set, *Newthyroid*, que si bien presenta la misma estructura de clasificación que *Iris* y *Rand*, supone un aumento en la dificultad de cómputo respecto a estos. Ya que tanto los data sets como las semillas se han modificado, se han repetido las ejecuciones para los algoritmos greedy y de búsqueda local.

§6.2.1: Iris & Rand

Los data sets de *Iris* y *Rand*, no presentan ninguna dificultad para los algoritmos genéticos y meméticos. Como podemos observar en la Tabla 6.10, todos los algortimos implementados en esta práctica consiguen converger a la solución óptima, tanto para *Rand* como para *Iris* con 10 % y 20 % de restricciones. Si bien es cierto, el algoritmo que peor ajuste realiza es el algoritmo greedy, pero como ya comentamos, el algoritmo greedy depende en gran medida de la colocación inicial de los centroides en el espacio de datos. Una buena colocación inicial nos ayuda a obtener una mejor solución.
Estos conjuntos son perfectos para ajustar el funcionamiento de los algoritmos genéticos y meméticos. Sus cromosomas no poseen una longitud excesiva y solo cuenta con 3 clusters para la clasificación. Esto facilita que el límite de iteraciones (100000) sea más que suficiente para que las poblaciones de cromosomas converjan a una solución óptima.

§6.2.2: Ecoli

El data set *Ecoli* si que presenta un reto a la hora de obtener una solución optima. Con un número mayor de clusters y de genes por cromosoma, el límite de evaluaciones puede quedarse corto si estas no se optimizan de la mejor manera posible. Cuantas menos evaluaciones redundantes seamos capaces de eliminar, más generaciones podrá alcanzar el algortimo genético o memético y mejores soluciones aportará. Esta dificultad añadida al cómputo se puede notar en que el tiempo necesario para agotar las evaluaciones en el data set *Ecoli* es 5 veces mayor que el necesario en los data sets *Rand* e *Iris*.
Tomando como referencia —tanto para 10 % de restricciones como para 20 %— los algoritmos greedy y de búsqueda local, todos los algoritmos genéticos y meméticos mejoran la solución aportada por estos dos, siendo el algoritmo greedy el que mejores soluciones aporta. Esto se puede deber a una buena colocación de los centroides iniciales.
También podemos apreciar que existen diferencias entre los algoritmos genéticos y meméticos. Estos últimos aportan mejores soluciones, lo que no es de extrañar, ya que a grandes rasgos se trata de un AGG_UN al que se le aplica una BLS. Entre el modelo generacional y el modelo estacionario, no podemos apreciar grandes diferencias en los resultados. Aunque si notar que para el modelo generacional, el operador que mejor resultado aporta es el operador de cruce uniforme, mientras que para el modelo estacionario, el operador que encuentra una mejor solución es el operador de cruce por segmento fijo. Esto podría explicarse porque, en el modelo generacional, al reemplazar a todos los individuos en cada generación, se favorece la exploración del espacio de soluciones —mientra que el operador de segmento fijo, al estar sesgado, favorece la explotación de las soluciones— permitiendo encontrar mejores óptimos. Para el modelo estacionario, al solo seleccionar a dos cromosomas cada generación, el operador de segmento fijo, explota a estos dos cromosomas, de manera que converjan a una mejor solución, en detrimento de una mayor exploración del espacio de cromosomas.
Entre los algoritmos meméticos, encontramos que para un 10 % de restricciones, la mejor soluciósn la aporta AM_10-0.1 y para un 20 % la aporta AM_10-0.1 mej. Estos dos meméticos, realizan la BLS a un 10 % de la población, por lo que podemos notar que eso es suficiente para encontrar buenas soluciones. AM_10-0.1 es el algoritmo más costoso, debido a que realiza una mayor cantidad de BLS. A la vista de los resultados, tantas BLS no significan una mejora en los resultados. Si bien, AM_10-1.0 mejora a los algoritmos genéticos en los que se basa, con aplicar la BLS a solo un 10 % es suficiente para encontrar las mismas o mejores soluciones.

§6.2.3: Newthyroid

Por último tenemos el data set *Newthyroid*, que se encuntra en la interseccion entre *Iris*, *Rand* y *Ecoli*. Posee el mismo número de cluster que *Iris* y *Rand*, pero su número de genes por cromosoma es mayor, acercandose a la dificultad de cómputo de *Ecoli* y aportando una diferenciación en las soluciones obtenidas.
En el caso de *Newthyroid*, es necesario establecer una separación entre el 10 % de restricciones y el 20 % de restricciones.
Para un 10 % de restricciones, el peor algoritmo es la búsqueda local, porque aunque parezca que algunos genéticos y meméticos aporten peores soluciones —fijándonos en la infactibilidad obtenida— el valor de la función objetivo que alcanza la búsqueda local es bastante mediocre. Por ello, todos los genéticos y meméticos mejoran esta solución.
Curiosamente, en los algoritmos genéticos, observamos el caso contrario a *Ecoli*. Los algoritmos AGG_SF y AGE_UN son los que aportan los mejores resultados respecto a los esquemas generacionales y estacionarios respectivamente. La solución que alcanzan estos algoritmos es de bastante calidad, superando a la aportada por la BL, exceptuando al algoritmo AGE_SF, que no es capaz de superar a la búsqueda local.
En el caso de los meméticos, alcanzan lo que podríamos llamar el optimo —la mejor solución encontrada para *Newthyroid* 10 %— los algoritmos AM_10-1.0 y AM_10-0.1. Por su parte AM_10-0.1mej, alcanza este óptimo solo en algunas iteraciones. De nuevo observamos que los algoritmos meméticos mejoran las soluciones aportadas por los genéticos.

Para un 20 % de restricciones, el resultado obtenido por la búsqueda local, mejora bastante, superando al aportado por greedy, pero quedandose atrás si lo comparamos con los algoritmos genéticos y meméticos.
En cuanto a los algoritmos genéticos, independientemente del esquema de reemplazo y selección, podemos ver que aquellos que utlizan el operador de cruce por segmento fijo, convergen al óptimo —la mejor solución encontrada para *Newthyroid* 20 %— mientras que los que implementan operador de cruce uniforme no. En este

caso vemos que la explotación que realiza el operador de segmento fijo, repercute positivamente en el resultado obtenido.
Por parte de los meméticos, todos convergen a la solución óptima. Por lo que podemos ver que la BLS que se aplica en estos algoritmos mejora las soluciones aportadas al utilizar el operador de cruce uniforme.

§6.2.4: Conclusión

Como conclusión, podemos ver que los algoritmos genéticos y meméticos son ampliamente superiores a los algoritmos de búsqueda local y greedy. Este resultado no es extraño, ya que, los algoritmos genéticos trabajan con poblaciones de varias decenas de soluciones a la vez, mientras que la BL o greedy buscan mejorar una única solución. La exploración que realizan los algoritmos basados en poblaciones es ampliamente superior a los que realizan los basados en trayectorias. Con el suficiente tiempo, los algoritmos genéticos y meméticos son capaces de encontrar soluciones más optimas, sin depender tanto del estado inicial. Esto, es en gran medida propiciado por el operador de mutación, que es el encargado de mejorar la diversidad de la población evitando que esta se estanque en un óptimo local y aumentado la exploración del espacio de soluciones.
También vemos que al implementar una búsqueda local en combinación, con un algoritmo genético —un memético— en un menor número de generaciones se consiguen mejores soluciones. Es importante también equilibrar cuantas veces se realiza esta búsqueda local dentro de la población y a cuantos individuos se aplica. Ya que esta operación es bastante costosa y si bien, podríamos obtener soluciones muy buenas, esto sería en detrimento del tiempo.

§6.3: Tablas

§6.3.1: Valores medios

Tabla 6.10: Resultados globales en el PAR con 10 % de restricciones

	Iris				Rand			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
K-medias Restringido Débil	0,62	15	0,95	0,01	0,91	16	1,37	0,00
Búsqueda Local	0,60	0	0,60	0,70	0,75	0	0,75	0,61
AGG_UN	0,60	0	0,60	29,11	0,75	0	0,75	29,13
AGG_SF	0,60	0	0,60	29,24	0,75	0	0,75	29,27
AGE_UN	0,60	0	0,60	29,67	0,75	0	0,75	29,62
AGE_SF	0,60	0	0,60	29,12	0,75	0	0,75	29,12
AM(10,1.0)	0,60	0	0,60	27,82	0,75	0	0,75	27,16
AM(10,0.1)	0,60	0	0,60	28,04	0,75	0	0,75	27,39
AM(10,0.1mej)	0,60	0	0,60	28,05	0,75	0	0,75	27,45

Tabla 6.10: Resultados globales en el PAR con 20 % de restricciones

	Iris				Rand			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
K-medias Restringido Débil	0,60	15	0,76	0,01	0,77	11	0,93	0,01
Búsqueda Local	0,60	0	0,60	0,67	0,75	0	0,75	0,64
AGG_UN	0,60	0	0,60	38,79	0,75	0	0,75	38,87
AGG_SF	0,60	0	0,60	39,05	0,75	0	0,75	39,12
AGE_UN	0,60	0	0,60	38,81	0,75	0	0,75	38,81
AGE_SF	0,60	0	0,60	39,17	0,75	0	0,75	39,13
AM(10,1.0)	0,60	0	0,60	37,04	0,75	0	0,75	37,10
AM(10,0.1)	0,60	0	0,60	37,11	0,75	0	0,75	37,15
AM(10,0.1mej)	0,60	0	0,60	37,10	0,75	0	0,75	37,18

Tabla 6.10: Resultados globales en el PAR con 10 % de restricciones

	Ecoli				Newthyroid			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
K-medias Restringido Débil	1.669,95	179	2.033,34	0,34	334,76	88	423,16	0,01
Búsqueda Local	1.791,66	329	2.457,27	38,34	430,97	43	497,11	2,41
AGG_UN	982,473	111	1.206,91	151,71	273,11	31	320,18	60,06
AGG_SF	977,12	149	1.278,12	152,94	276,95	29	321,87	60,18
AGE_UN	952,53	128	1.212,21	156,84	269,58	29	314,80	60,79
AGE_SF	947,54	124	1.198,31	155,18	267,36	45	337,18	60,11
AM(10,1.0)	908,77	69	1.048,54	157,38	270,80	15	293,88	59,62
AM(10,0.1)	885,09	76	1.039,84	155,78	270,80	15	293,88	58,47
AM(10,0.1mej)	887,64	73	1.034,70	152,41	272,32	64	370,45	58,53

Tabla 6.10: Resultados globales en el PAR con 20 % de restricciones

	Ecoli				Newthyroid			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
K-medias Restringido Débil	1.642,50	146	1.790,36	0,23	318,22	103	397,72	0,01
Búsqueda Local	1.673,46	340	2.018,01	48,48	311,87	20	327,56	3,19
AGG_UN	948,18	165	1.115,70	204,08	269,39	62	316,75	80,76
AGG_SF	946,03	175	1.123,67	204,81	270,80	41	302,33	81,46
AGE_UN	942,56	189	1.134,18	207,88	268,46	77	327,51	80,52
AGE_SF	950,07	148	1.099,96	206,69	270,80	41	302,33	81,01
AM(10,1.0)	917,43	123	1.041,60	223,52	270,80	41	302,33	78,11
AM(10,0.1)	927,62	98	1.026,67	201,03	270,80	41	302,33	77,53
AM(10,0.1mej)	939,57	104	1.044,90	203,37	270,80	41	302,33	77,54

§6.3.2: K-medias Restringido Débil

Restricciones del 10 %

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

	Iris				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	0,73	49	1,83	0,01	1,34	35	2,35	0,00
Ejecución 2	0,59	15	0,93	0,01	0,75	0	0,75	0,00
Ejecución 3	0,60	0	0,60	0,01	0,75	0	0,75	0,01
Ejecución 4	0,60	0	0,60	0,00	1,00	44	2,27	0,00
Ejecución 5	0,58	9	0,78	0,01	0,75	0	0,75	0,00
Media	0,62	15	0,95	0,01	0,91	16	1,37	0,00

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

	Ecoli				**Newthyroid**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	1.579,07	121	1.824,17	0,79	307,22	62	369,22	0,01
Ejecución 2	1.742,22	194	2.135,18	0,45	379,52	331	710,52	0,01
Ejecución 3	1.730,48	213	2.161,93	0,15	300,82	7	307,82	0,01
Ejecución 4	1.615,59	125	1.868,79	0,19	375,95	40	415,95	0,02
Ejecución 5	1.682,40	244	2.176,64	0,12	310,29	2	312,29	0,01
Media	1.669,95	179	2.033,34	0,34	334,76	88	423,16	0,01

Restricciones del 20 %

Tabla 6.1: Resultados obtenidos por el algoritmo greedy en el PAR con 20 % de restricciones

	Iris				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	0,62	56	1,25	0,01	0,88	38	1,43	0,01
Ejecución 2	0,59	17	0,78	0,01	0,75	0	0,75	0,01
Ejecución 3	0,60	0	0,60	0,01	0,75	0	0,75	0,01
Ejecución 4	0,60	0	0,60	0,01	0,73	16	0,96	0,01
Ejecución 5	0,60	0	0,60	0,01	0,75	0	0,75	0,00
Media	0,60	15	0,76	0,01	0,77	11	0,93	0,01

Tabla 6.1: Resultados obtenidos por el algoritmo greedy en el PAR con 20 % de restricciones

	Ecoli				**Newthyroid**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	1.681,44	95	1.777,66	0,11	310,74	118	401,46	0,01
Ejecución 2	1.768,64	236	2.007,66	0,48	307,12	236	488,57	0,02
Ejecución 3	1.473,69	177	1.652,95	0,21	356,00	101	433,66	0,01
Ejecución 4	1.605,64	98	1.704,89	0,16	306,93	57	350,76	0,01
Ejecución 5	1.683,07	124	1.808,66	0,16	310,33	5	314,18	0,01
Media	1.642,50	146	1.790,36	0,23	318,22	103	397,72	0,01

§6.3.3: Búsqueda Local

Restricciones del 10 %

Tabla 6.2: Resultados obtenidos por el algotimo búsqueda local en el PAR con 10 % de restricciones

	Iris				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	0,60	0	0,60	0,84	0,75	0	0,75	0,58
Ejecución 2	0,60	0	0,60	0,76	0,75	0	0,75	0,66
Ejecución 3	0,60	0	0,60	0,58	0,75	0	0,75	0,71
Ejecución 4	0,60	0	0,60	0,80	0,75	0	0,75	0,55
Ejecución 5	0,60	0	0,60	0,53	0,75	0	0,75	0,56
Media	0,60	0	0,60	0,70	0,75	0	0,75	0,61

Tabla 6.2: Resultados obtenidos por el algotimo búsqueda local en el PAR con 10 % de restricciones

	Ecoli				**Newthyroid**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	1.588,36	148	1.888,15	51,43	522,06	77	640,49	1,53
Ejecución 2	2.055,26	881	3.839,80	19,15	322,25	4	328,40	3,16
Ejecución 3	1.410,74	105	1.623,43	60,98	348,59	10	363,97	3,51
Ejecución 4	1.946,41	282	2.517,63	28,28	453,24	39	513,23	2,08
Ejecución 5	1.957,52	227	2.417,33	31,87	508,72	85	639,46	1,77
Media	1.791,66	329	2.457,27	38,34	430,97	43	497,11	2,41

Restricciones del 20 %

Tabla 6.2: Resultados obtenidos por el algotimo búsqueda local en el PAR con 20 % de restricciones

	Iris				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	0,60	0	0,60	0,68	0,75	0	0,75	0,72
Ejecución 2	0,60	0	0,60	0,60	0,75	0	0,75	0,68
Ejecución 3	0,60	0	0,60	0,62	0,75	0	0,75	0,59
Ejecución 4	0,60	0	0,60	0,80	0,75	0	0,75	0,58
Ejecución 5	0,60	0	0,60	0,65	0,75	0	0,75	0,64
Media	0,60	0	0,60	0,67	0,75	0	0,75	0,64

Tabla 6.2: Resultados obtenidos por el algotimo búsqueda local en el PAR con 20 % de restricciones

	Ecoli				**Newthyroid**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	1.601,88	262	1.867,23	52,78	349,60	25	368,83	3,45
Ejecución 2	1.613,00	273	1.889,49	64,09	300,55	14	311,31	3,84
Ejecución 3	1.530,28	22	1.552,56	49,31	304,48	11	312,94	2,87
Ejecución 4	1.785,86	589	2.382,39	34,28	313,30	0	313,30	2,76
Ejecución 5	1.836,28	555	2.398,38	41,94	291,43	52	331,41	3,02
Media	1.673,46	340	2.018,01	48,48	311,87	20	327,56	3,19

§6.3.4: AGG_UN

Restricciones del 10 %

Tabla 6.3: Resultados obtenidos por AGG_UN en el PAR con 10 % de restricciones

	Iris				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	0,60	0	0,60	28,89	0,75	0	0,75	29,00
Ejecución 2	0,60	0	0,60	29,14	0,75	0	0,75	29,17
Ejecución 3	0,60	0	0,60	29,13	0,75	0	0,75	29,17
Ejecución 4	0,60	0	0,60	29,20	0,75	0	0,75	29,12
Ejecución 5	0,60	0	0,60	29,21	0,75	0	0,75	29,19
Media	0,60	0	0,60	29,11	0,75	0	0,75	29,13

Tabla 6.3: Resultados obtenidos por AGG_UN en el PAR con 10 % de restricciones

	Ecoli				**Newthyroid**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	948,51	33	1.015,35	158,53	270,80	15	293,88	59,75
Ejecución 2	951,15	72	1.096,99	150,21	270,80	15	293,88	60,16
Ejecución 3	1.015,42	57	1.130,88	149,84	282,33	93	425,38	60,12
Ejecución 4	912,67	140	1.196,25	149,86	270,80	15	293,88	60,13
Ejecución 5	1.084,63	252	1.595,08	150,10	270,80	15	293,88	60,15
Media	982,473	111	1.206,908	151,71	273,11	31	320,18	60,06

Restricciones del 20 %

Tabla 6.3: Resultados obtenidos por AGG_UN en el PAR con 20 % de restricciones

	Iris				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	0,60	0	0,60	39,04	0,75	0	0,75	39,19
Ejecución 2	0,60	0	0,60	39,08	0,75	0	0,75	39,08
Ejecución 3	0,60	0	0,60	39,04	0,75	0	0,75	39,11
Ejecución 4	0,60	0	0,60	39,05	0,75	0	0,75	39,09
Ejecución 5	0,60	0	0,60	39,05	0,75	0	0,75	39,13
Media	0,60	0	0,60	39,05	0,75	0	0,75	39,12

Tabla 6.3: Resultados obtenidos por AGG_UN en el PAR con 20 % de restricciones

	Ecoli				**Newthyroid**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	975,28	284	1.262,91	204,67	263,72	144	374,43	80,94
Ejecución 2	926,71	163	1.091,79	203,95	270,80	41	302,33	80,64
Ejecución 3	944,22	85	1.030,31	203,92	270,80	41	302,33	80,78
Ejecución 4	904,59	165	1.071,70	203,94	270,80	41	302,33	80,72
Ejecución 5	990,13	130	1.121,79	203,92	270,80	41	302,33	80,73
Media	948,18	165	1.115,70	204,08	269,39	62	316,75	80,76

§6.3.5: AGG_SF

Restricciones del 10 %

Tabla 6.4: Resultados obtenidos por AGG_SF en el PAR con 10 % de restricciones

	Iris				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	0,60	0	0,60	29,30	0,75	0	0,75	29,21
Ejecución 2	0,60	0	0,60	29,26	0,75	0	0,75	29,29
Ejecución 3	0,60	0	0,60	29,27	0,75	0	0,75	29,25
Ejecución 4	0,60	0	0,60	29,18	0,75	0	0,75	29,36
Ejecución 5	0,60	0	0,60	29,20	0,75	0	0,75	29,22
Media	0,60	0	0,60	29,24	0,75	0	0,75	29,27

Tabla 6.4: Resultados obtenidos por AGG_SF en el PAR con 10 % de restricciones

	Ecoli				**Newthyroid**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	1.043,76	108	1.262,52	161,22	270,80	15	293,88	60,56
Ejecución 2	926,30	57	1.041,76	151,23	301,55	86	433,83	60,12
Ejecución 3	1.056,77	367	1.800,16	150,88	270,80	15	293,88	60,07
Ejecución 4	923,36	71	1.067,17	150,64	270,80	15	293,88	60,06
Ejecución 5	935,39	140	1.218,97	150,72	270,80	15	293,88	60,08
Media	977,12	149	1.278,12	152,94	276,95	29	321,87	60,18

Restricciones del 20 %

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

	Iris				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	0,60	0	0,60	39,04	0,75	0	0,75	39,19
Ejecución 2	0,60	0	0,60	39,08	0,75	0	0,75	39,08
Ejecución 3	0,60	0	0,60	39,04	0,75	0	0,75	39,11
Ejecución 4	0,60	0	0,60	39,05	0,75	0	0,75	39,09
Ejecución 5	0,60	0	0,60	39,05	0,75	0	0,75	39,13
Media	0,60	0	0,60	39,05	0,75	0	0,75	39,12

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

	Ecoli				**Newthyroid**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	942,36	160	1.104,41	205,44	270,80	41	302,33	82,22
Ejecución 2	926,56	176	1.104,81	205,01	270,80	41	302,33	81,43
Ejecución 3	948,74	167	1.117,87	204,56	270,80	41	302,33	81,23
Ejecución 4	972,93	204	1.179,54	204,40	270,80	41	302,33	81,15
Ejecución 5	939,53	170	1.111,71	204,65	270,80	41	302,33	81,25
Media	946,03	175	1.123,67	204,81	270,80	41	302,33	81,46

§6.3.6: AGE_UN

Restricciones del 10 %

Tabla 6.5: Resultados obtenidos por AGE_UN en el PAR con 10 % de restricciones

	Iris				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	0,60	0	0,60	29,00	0,75	0	0,75	28,96
Ejecución 2	0,60	0	0,60	29,83	0,75	0	0,75	29,81
Ejecución 3	0,60	0	0,60	29,84	0,75	0	0,75	29,78
Ejecución 4	0,60	0	0,60	29,80	0,75	0	0,75	29,77
Ejecución 5	0,60	0	0,60	29,87	0,75	0	0,75	29,78
Media	0,60	0	0,60	29,67	0,75	0	0,75	29,62

Tabla 6.5: Resultados obtenidos por AGE_UN en el PAR con 10 % de restricciones

	Ecoli				**Newthyroid**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	964,72	110	1.187,53	159,40	270,80	15	293,88	59,83
Ejecución 2	889,55	146	1.185,28	161,51	270,80	15	293,88	61,00
Ejecución 3	959,79	147	1.257,55	154,93	264,67	87	398,48	61,01
Ejecución 4	993,33	112	1.220,20	155,57	270,80	15	293,88	61,06
Ejecución 5	955,28	126	1.210,50	152,81	270,80	15	293,88	61,05
Media	952,53	128	1.212,21	156,84	269,58	29	314,80	60,79

Restricciones del 20 %

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

	Iris				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	0,60	0	0,60	38,88	0,75	0	0,75	38,88
Ejecución 2	0,60	0	0,60	38,80	0,75	0	0,75	38,80
Ejecución 3	0,60	0	0,60	38,80	0,75	0	0,75	38,80
Ejecución 4	0,60	0	0,60	38,76	0,75	0	0,75	38,76
Ejecución 5	0,60	0	0,60	38,81	0,75	0	0,75	38,81
Media	0,60	0	0,60	38,81	0,75	0	0,75	38,81

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

	Ecoli				**Newthyroid**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	938,74	170	1.110,92	215,75	270,80	41	302,33	80,71
Ejecución 2	909,79	152	1.063,73	206,70	270,80	41	302,33	80,57
Ejecución 3	1.042,89	210	1.255,58	207,66	270,80	41	302,33	80,67
Ejecución 4	946,14	147	1.095,02	204,69	259,11	220	428,26	80,71
Ejecución 5	875,24	267	1.145,65	204,62	270,80	41	302,33	79,96
Media	942,56	189	1.134,18	207,88	268,46	77	327,51	80,52

§6.3.7: AGE_SF

Restricciones del 10 %

Tabla 6.6: Resultados obtenidos por el AGE_SF en el PAR con 10 % de restricciones

	Iris				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	0,60	0	0,60	29,21	0,75	0	0,75	29,24
Ejecución 2	0,60	0	0,60	29,08	0,75	0	0,75	29,15
Ejecución 3	0,60	0	0,60	29,12	0,75	0	0,75	29,09
Ejecución 4	0,60	0	0,60	29,10	0,75	0	0,75	29,06
Ejecución 5	0,60	0	0,60	29,08	0,75	0	0,75	29,09
Media	0,60	0	0,60	29,12	0,75	0	0,75	29,12

Tabla 6.6: Resultados obtenidos por el AGE_SF en el PAR con 10 % de restricciones

	Ecoli				**Newthyroid**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	1.029,46	108	1.248,22	159,93	270,80	15	293,88	60,16
Ejecución 2	931,86	130	1.195,19	160,92	264,67	87	398,48	60,12
Ejecución 3	938,80	216	1.376,33	153,32	270,80	15	293,88	60,14
Ejecución 4	897,44	91	1.081,77	151,09	270,80	15	293,88	60,03
Ejecución 5	940,13	74	1.090,02	150,64	259,70	95	405,82	60,10
Media	947,54	124	1.198,31	155,18	267,36	45	337,18	60,11

Restricciones del 20 %

Tabla 6.6: Resultados obtenidos por AGE_SF en el PAR con 20 % de restricciones

	Iris				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	0,60	0	0,60	39,09	0,75	0	0,75	39,17
Ejecución 2	0,60	0	0,60	39,35	0,75	0	0,75	39,09
Ejecución 3	0,60	0	0,60	39,16	0,75	0	0,75	39,13
Ejecución 4	0,60	0	0,60	39,07	0,75	0	0,75	39,16
Ejecución 5	0,60	0	0,60	39,18	0,75	0	0,75	39,10
Media	0,60	0	0,60	39,17	0,75	0	0,75	39,13

Tabla 6.6: Resultados obtenidos por AGE_SF en el PAR con 20 % de restricciones

	Ecoli				**Newthyroid**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	914,14	137	1.052,89	208,09	270,80	41	302,33	81,03
Ejecución 2	950,95	235	1.188,95	208,45	270,80	41	302,33	81,21
Ejecución 3	919,39	121	1.041,93	205,53	270,80	41	302,33	81,11
Ejecución 4	950,53	115	1.067,00	206,40	270,80	41	302,33	81,09
Ejecución 5	1.015,33	132	1.149,02	205,00	270,80	41	302,33	80,62
Media	950,07	148	1.099,96	206,69	270,80	41	302,33	81,01

Tabla 6.7: Resultados obtenidos por AM_10-1.0 en el PAR con 20 % de restricciones

	Iris				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	0,60	0	0,60	37,07	0,75	0	0,75	37,11
Ejecución 2	0,60	0	0,60	37,04	0,75	0	0,75	37,07
Ejecución 3	0,60	0	0,60	37,06	0,75	0	0,75	37,15
Ejecución 4	0,60	0	0,60	37,00	0,75	0	0,75	37,11
Ejecución 5	0,60	0	0,60	37,02	0,75	0	0,75	37,07
Media	0,60	0	0,60	37,04	0,75	0	0,75	37,10

Tabla 6.7: Resultados obtenidos por AM_10-1.0 en el PAR con 20 % de restricciones

	Ecoli				**Newthyroid**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	927,73	119	1.048,26	229,96	270,80	41	302,33	79,36
Ejecución 2	913,78	151	1.066,71	224,78	270,80	41	302,33	79,57
Ejecución 3	913,78	106	1.021,14	225,20	270,80	41	302,33	79,55
Ejecución 4	932,80	98	1.032,05	220,49	270,80	41	302,33	76,56
Ejecución 5	899,08	139	1.039,86	217,19	270,80	41	302,33	75,50
Media	917,43	123	1.041,60	223,52	270,80	41	302,33	78,11

§6.3.8: AM_10-1.0

Restricciones del 10 %

Tabla 6.7: Resultados obtenidos por AM_10-1.0 en el PAR con 10 % de restricciones

	Iris				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	0,60	0	0,60	27,24	0,75	0	0,75	27,41
Ejecución 2	0,60	0	0,60	28,04	0,75	0	0,75	27,15
Ejecución 3	0,60	0	0,60	27,96	0,75	0	0,75	26,99
Ejecución 4	0,60	0	0,60	27,93	0,75	0	0,75	27,04
Ejecución 5	0,60	0	0,60	27,92	0,75	0	0,75	27,22
Media	0,60	0	0,60	27,82	0,75	0	0,75	27,16

Tabla 6.7: Resultados obtenidos por AM_10-1.0 en el PAR con 10 % de restricciones

	Ecoli				**Newthyroid**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	916,37	71	1.060,19	164,32	270,80	15	293,88	58,34
Ejecución 2	884,32	81	1.048,39	154,76	270,80	15	293,88	60,04
Ejecución 3	906,88	72	1.052,72	148,74	270,80	15	293,88	59,85
Ejecución 4	931,88	56	1.045,32	158,80	270,80	15	293,88	59,99
Ejecución 5	904,42	65	1.036,09	160,26	270,80	15	293,88	59,86
Media	908,77	69	1.048,54	157,38	270,80	15	293,88	59,62

Tabla 6.8: Resultados obtenidos por AM_10-0.1 en el PAR con 20 % de restricciones

	Ecoli				Newthyroid			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	927,73	98	1.026,99	207,63	270,80	41	302,33	77,54
Ejecución 2	933,20	85	1.019,28	199,43	270,80	41	302,33	77,56
Ejecución 3	930,98	122	1.054,54	199,72	270,80	41	302,33	77,49
Ejecución 4	902,16	120	1.023,70	200,62	270,80	41	302,33	77,55
Ejecución 5	944,03	64	1.008,85	197,78	270,80	41	302,33	77,49
Media	927,62	98	1.026,67	201,03	270,80	41	302,33	77,53

Restricciones del 20 %

§6.3.9: AM_10-0.1

Restricciones del 10 %

Tabla 6.8: Resultados obtenidos por AM_10-0.1 en el PAR con 10 % de restricciones

	Iris				Rand			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	0,60	0	0,60	27,26	0,75	0	0,75	27,44
Ejecución 2	0,60	0	0,60	28,27	0,75	0	0,75	27,44
Ejecución 3	0,60	0	0,60	28,25	0,75	0	0,75	27,29
Ejecución 4	0,60	0	0,60	28,22	0,75	0	0,75	27,30
Ejecución 5	0,60	0	0,60	28,17	0,75	0	0,75	27,49
Media	0,60	0	0,60	28,04	0,75	0	0,75	27,39

Tabla 6.8: Resultados obtenidos por AM_10-0.1 en el PAR con 10 % de restricciones

	Ecoli				Newthyroid			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	887,98	70	1.029,77	153,73	270,80	15	293,88	56,96
Ejecución 2	903,25	57	1.018,71	157,67	270,80	15	293,88	58,92
Ejecución 3	869,35	86	1.043,55	158,18	270,80	15	293,88	58,76
Ejecución 4	837,80	121	1.082,89	154,34	270,80	15	293,88	58,83
Ejecución 5	927,07	48	1.024,30	154,95	270,80	15	293,88	58,90
Media	885,09	76	1.039,84	155,78	270,80	15	293,88	58,47

Restricciones del 20 %

Tabla 6.8: Resultados obtenidos por AM_10-0.1 en el PAR con 20 % de restricciones

	Iris				Rand			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	0,60	0	0,60	37,11	0,75	0	0,75	37,13
Ejecución 2	0,60	0	0,60	37,07	0,75	0	0,75	37,18
Ejecución 3	0,60	0	0,60	37,10	0,75	0	0,75	37,16
Ejecución 4	0,60	0	0,60	37,16	0,75	0	0,75	37,15
Ejecución 5	0,60	0	0,60	37,09	0,75	0	0,75	37,14
Media	0,60	0	0,60	37,11	0,75	0	0,75	37,15

§6.3.10: AM_10-0.1mej

Restricciones del 10 %

Tabla 6.9: Resultados obtenidos por AM_10-0.1mej en el PAR con 10 % de restricciones

	Iris				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	0,60	0	0,60	27,31	0,75	0	0,75	27,50
Ejecución 2	0,60	0	0,60	28,36	0,75	0	0,75	27,42
Ejecución 3	0,60	0	0,60	28,20	0,75	0	0,75	27,44
Ejecución 4	0,60	0	0,60	28,18	0,75	0	0,75	27,32
Ejecución 5	0,60	0	0,60	28,21	0,75	0	0,75	27,57
Media	0,60	0	0,60	28,05	0,75	0	0,75	27,45

Tabla 6.9: Resultados obtenidos por AM_10-0.1mej en el PAR con 10 % de restricciones

	Ecoli				**Newthyroid**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	908,22	50	1.009,50		277,68	98	428,41	57,10
Ejecución 2	915,67	65	1.047,33	157,79	282,62	96	430,27	59,01
Ejecución 3	855,09	94	1.045,49	151,52	270,80	15	293,88	58,82
Ejecución 4	906,44	69	1.046,21	151,44	259,70	95	405,82	58,82
Ejecución 5	852,81	85	1.024,98	148,88	270,80	15	293,88	58,88
Media	887,64	73	1.034,70	152,41	272,32	64	370,45	58,53

Restricciones del 20 %

Tabla 6.9: Resultados obtenidos por AM_10-0.1mej en el PAR con 20 % de restricciones

	Iris				**Rand**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	0,60	0	0,60	37,09	0,75	0	0,75	37,15
Ejecución 2	0,60	0	0,60	37,10	0,75	0	0,75	37,22
Ejecución 3	0,60	0	0,60	37,07	0,75	0	0,75	37,14
Ejecución 4	0,60	0	0,60	37,12	0,75	0	0,75	37,17
Ejecución 5	0,60	0	0,60	37,10	0,75	0	0,75	37,20
Media	0,60	0	0,60	37,10	0,75	0	0,75	37,18

Tabla 6.9: Resultados obtenidos por AM_10-0.1mej en el PAR con 20 % de restricciones

	Ecoli				**Newthyroid**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
Ejecución 1	945,96	85	1.032,05	209,88	270,80	41	302,33	77,54
Ejecución 2	930,96	96	1.028,19	207,24	270,80	41	302,33	77,59
Ejecución 3	919,99	116	1.037,48	199,69	270,80	41	302,33	77,48
Ejecución 4	923,90	112	1.037,33	200,32	270,80	41	302,33	77,53
Ejecución 5	977,04	111	1.089,46	199,74	270,80	41	302,33	77,58
Media	939,57	104	1.044,90	203,37	270,80	41	302,33	77,54

§6.4: Experimentos

§6.4.1: Graficas

A modo de extra visual, he generado dos graficas. Estas gráficas muestran cómo se reduce el valor de la función objetivo según aumenta el número de generaciones de cromosomas. Concretamente ambas gráficas muestran los resultados sobre el conjunto *Ecoli* con un 10 % de restricciones y una semilla.

En la Figura 6.2 vemos la comparativa entre los 4 tipos de algoritmos geneticos implementados.

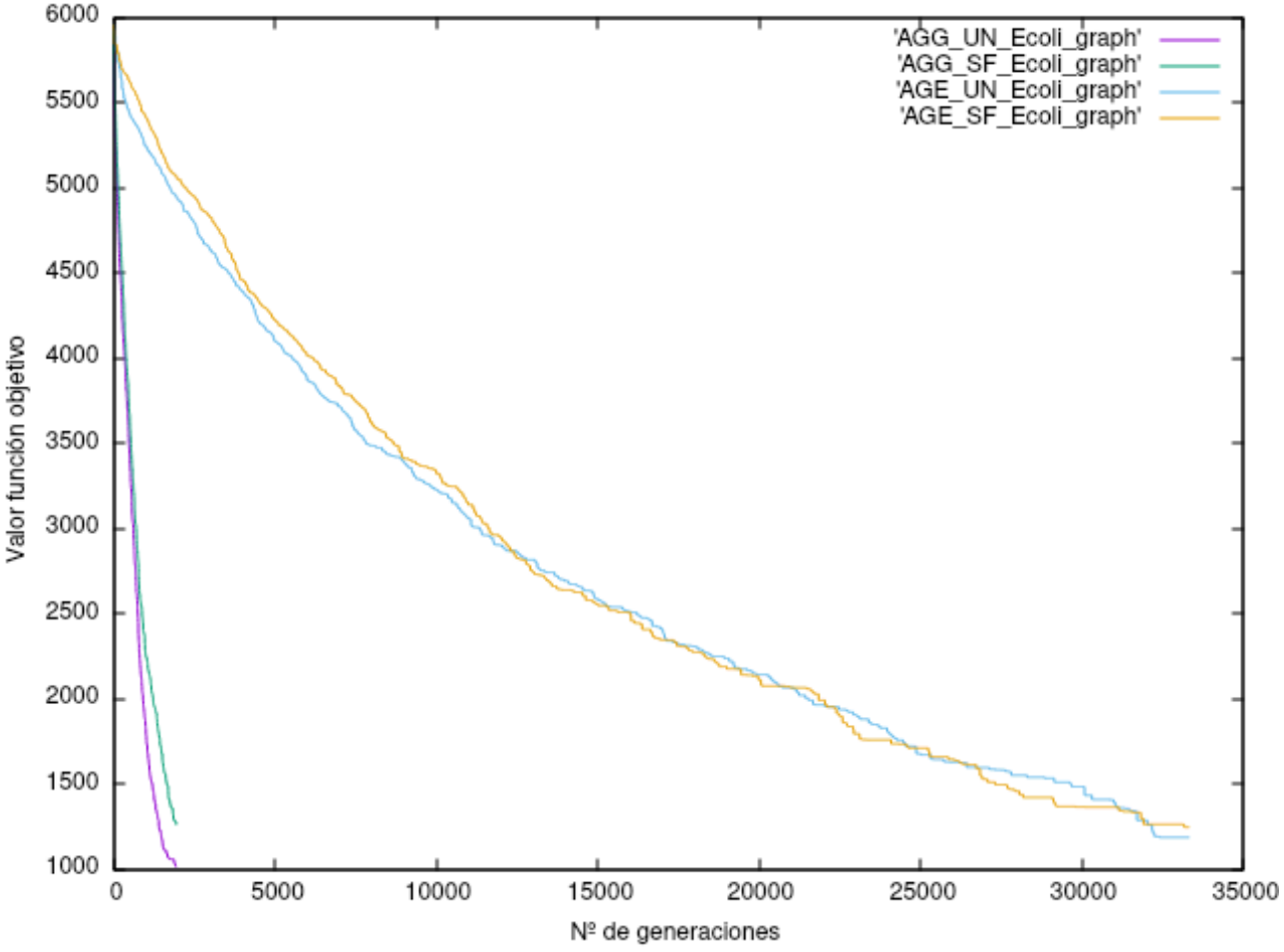

Figura 6.1: Algoritmo genéticos

Podemos notar que los algoritmos que implementan un esquema generacional, convergen muy rápidamente a una solución óptima. Esto se explica porque en un esquema generacional se escogen y evalúan 50 cromosomas —en nuestro caso— por generación, mientras que en el esquema estacionario simplemente se seleccionan y evalúan 2. Por ello vemos que la curva de los estacionaros desciende suavemente y alcanza soluciones de la misma calidad, pero habiendo alcanzado un número notablemente mayor de generaciones.

En la figura 6.3 vemos la misma gráfica que antes, pero ahora centrada en los algoritmos meméticos.

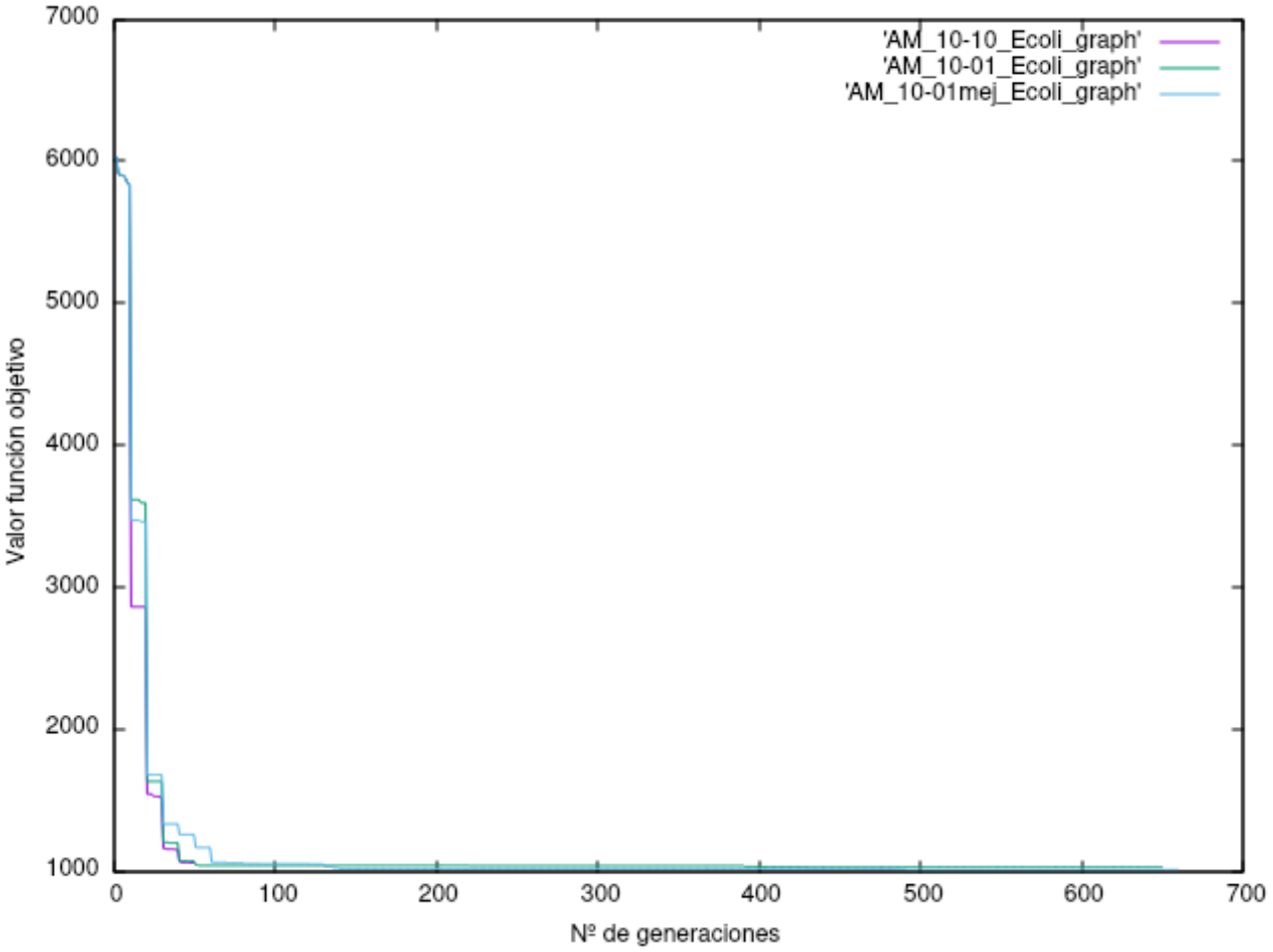

Figura 6.2: Algortimos meméticos

Vemos que es una curiosa gráfica. Los pequeños llanos representan la progresión normal del algoritmo genético. Por otro lado, las pronunciadas —prácticamente verticales— reducciones en el valor de la función objetivo, son provocadas por la BLS. Podemos ver así la gran repercusión que tiene aplicar la BLS a los cromosomas de la población, tanto en la mejora del valor de la función objetivo como en el número de generaciones que se alcanza. Aplicar la BLS, es un proceso sumamente costoso en evaluaciones, por tanto, se tarda mucho menos tiempo en agotar el máximo de evaluaciones posible. Finalmente se aprecia, que llega un momento en que estos algoritmos se estancan en un óptimo —cosa que sucede bastante deprisa— y no pueden escapar de él ni siquiera con ayuda de la BLS.

§6.4.2: Exploración vs Explotación en segmento fijo

Como segundo extra, se ha decido comprobar, como afecta el sesgo del operador de cruce por segmento fijo, a las posibles soluciones. Estas pruebas se han realizado con los conjuntos de *Ecoli* y *Newthyroid*, puesto que son los que obtienen soluciones más diversas. Recordar también que este estudio busca saber como afecta este sesgo a estos conjuntos de datos en concreto. Sabemos por el teorema de "No free lunch" que una metaheurística que sea muy buena para determinado problema, puede ser nefasta para otro.
En primer lugar, definamos el sesgo del operador de cruce por segmento fijo. Si en este operador los genes que se seleccionan de manera fija, son los del mejor de los dos padres, estamos favoreciendo la **explotación** —puesto que nos quedamos con un mayor número de genes de una mejor solución— y si estos son del peor de los padres, favorecemos la **exploración** puesto que diversificamos la solución, con la esperanza de que esto nos permita encontrar una solución mejor en el futuro.

Viendo los resultados, notamos otra vez que lo que es mejor en el conjunto *Ecoli*, es menos óptimo para *Newthyroid*. *Ecoli* se ve favorecido por la explotación de la solución, mientras que *Newthyroid* se ve favorecido por la exploración. Esta tendencia sucede cuando hablamos del conjunto de 10 % de restricciones, mientras que esto no pasa con el conjunto de 20 % —en el que en ambos conjuntos, la explotación favorece a encontrar el óptimo.
Respecto a *Ecoli* y *Newthyroid* para el 20 %, la diferencia que produce este sesgo en la solución puede deberse a que, en *Ecoli* el espacio de soluciones es más complejo —debido a la mayor longitud de los cromosomas y el número de cluster— lo que hace que explotar buenas soluciones sea mejor opción que explorar unas nuevas, mientras que en *Newthyroid* al ser un espacio mas "simple", la exploración favorece encontrar mejores soluciones.

Exploración VS Explotación en Segmento Fijo 10 %

	Ecoli				**Newthyroid**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
AGG_SF Explotación	977,12	149	1.278,12	155,63	276,95	29	321,87	62,21
AGG_SF Exploración	1.252,10	368	1.997,92	159,36	270,80	15	293,88	62,25
AGE_SF Explotación	947,54	124	1.198,31	158,03	267,36	45	337,18	62,25
AGE_SF Exploración	1.030,86	348	1.735,77	161,51	269,57	29	314,80	62,23

Exploración VS Explotación en Segmento Fijo 10 %

	Ecoli				**Newthyroid**			
	Tasa_C	***Tasa_inf***	***Agr.***	**T**	***Tasa_C***	***Tasa_inf***	***Agr.***	**T**
AGG_SF Explotación	946,03	175	1.123,67	207,23	270,80	41	302,33	81,74
AGG_SF Exploración	1.154,02	543	1.704,17	233,60	268,84	67	320,20	91,51
AGE_SF Explotación	950,07	148	1.099,96	208,72	270,80	41	302,33	81,10
AGE_SF Exploración	961,84	201	1.165,01	236,76	273,01	106	354,20	91,52

Finalmente, en la siguiente figura, vemos la gráfica que compara el valor de la función objetivo con el número de generaciones para las diferentes configuraciones del operador de cruce por segmento fijo. También se han añadido los algoritmos genéticos con operador de cruce uniforme, para hacer más interesante la comparación. Estos datos se han obtenido para la ejecución de una semilla en *Ecoli* 10 %

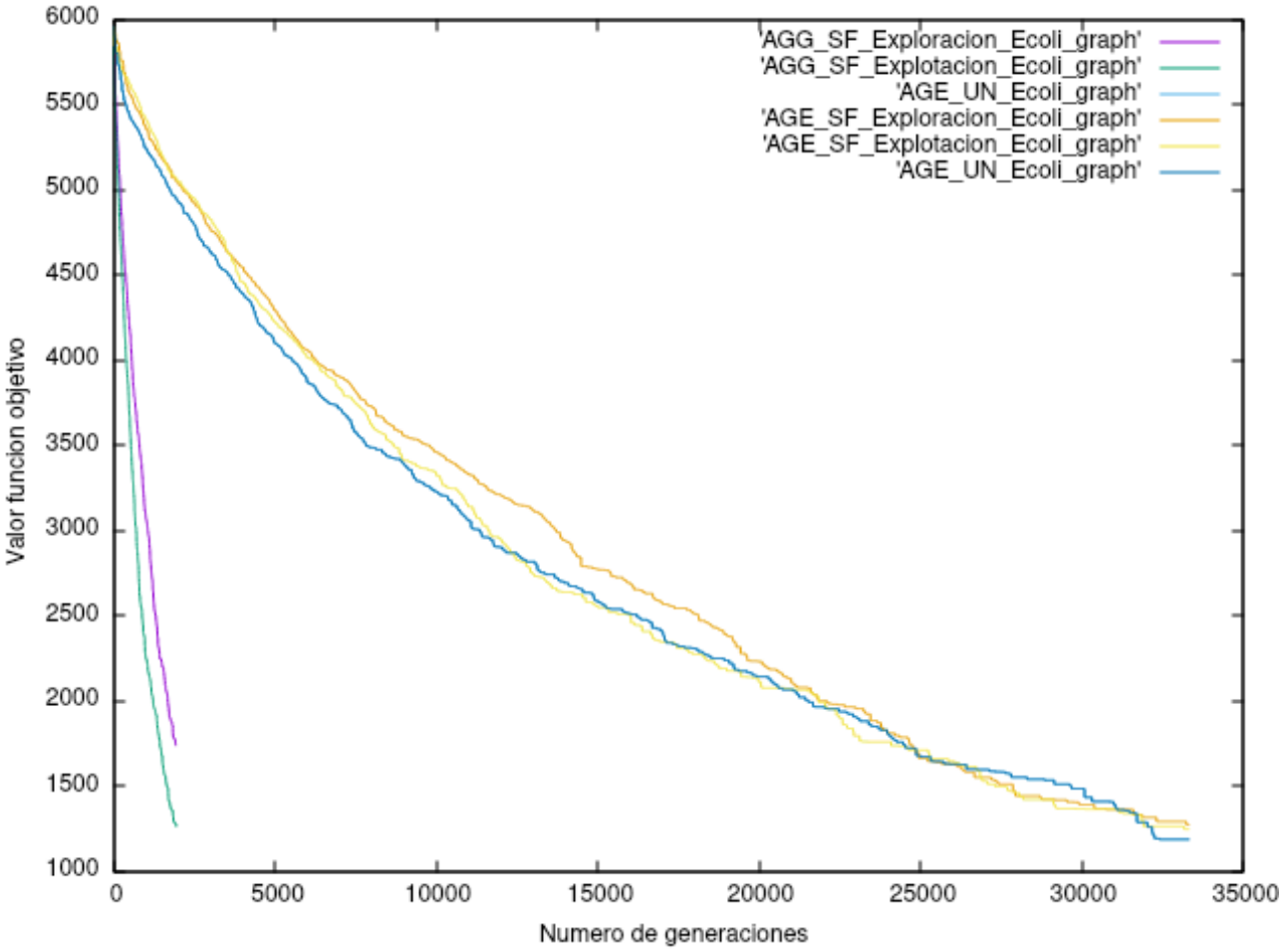

Figura 6.3:

Finalmente, podemos concluir que si bien existe diferencia en las soluciones debido al sesgo de este operador de cruce, estas no son lo suficientemente grandes, como para establecer que una configuración sea mejor que otra.