Práctica 2.b

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

Miguel Ángel Fernández Gutiérrez

Metaheurísticas

4º Doble Grado en Ingeniería Informática y Matemáticas

Universidad de Granada

mianfg.me

Índice

I Descripción del problema

El **problema de agrupamiento de restricciones** es una generalización del problema de agrupamiento convencional, al que le añadimos una serie de restricciones. Se formula como sigue:

> **Problema** (PAR, restricciones débiles). Dado un conjunto de datos $X = \{\vec{x_1}, ..., \vec{x_n}\}$, encontrar una partición $C = \{c_1, ..., c_k\}$, de modo que se minimicen la desviación general, $\bar{C}$, y la *infeasibility*(C) (número de restricciones de R, el conjunto de restricciones, incumplidas).

El objetivo de nuestro problema es obtener una partición *solución* que permita minimizar la función *fitness*, o f, definida como sigue:

$$f(solución) = d_{intra-cluster}(solución) + \lambda \cdot infeasibility(solución).$$

Se trata por tanto de un problema con dos objetivos, integrados en una función mono-objetivo mediante una suma ponderada con λ. Para cerciorarnos de la relevancia de *infeasibility*, fijaremos $\lambda = \frac{n}{|R|}$. Definiremos a continuación ciertos elementos y medidas del problema:[1]

- **Conjunto de datos**, X. Matriz de $n \times d$ valores reales, siendo n el número de instancias (o datos) y d la dimensión de cada instancia (todas las instancias tienen la misma dimensión).

  $$X = \{\vec{x_1}, ..., \vec{x_n}\},\ \vec{x_i} = (x_{[i,1]}, ..., x_{[i,d]}),\ x_{[i,j]} \in \mathbb{R}\ \ \forall i \in \{1, ..., n\}, j \in \{1, ..., d\}$$

- **Partición**, C. Es una asignación de cada $\vec{x_i}$ a un cluster:

  $$C = \{c_1, ..., c_k\}, c_i \subset \{1, ..., n\}$$

  Las soluciones a nuestro problema se representan en forma de particiones. Por tanto, dada una solución *solucion*, esta definirá una partición C de modo que $\texttt{solucion}[i] = j \iff j \in c_i$.
- **Centroide**, μ_i. Es el vector promedio de las instancias de X que componen al cluster:

  $$\mu_i = \frac{1}{|c_i|} \sum_{j \in c_i} \vec{x_j}$$

- **Desviación general de la partición**, $\bar{C}$. Es la media de las **desviaciones intra-cluster** para cada cluster, $\bar{c_i}$. Definimos ambas a continuación:

  $$\bar{c_i} = \frac{1}{|c_i|} \sum_{j \in c_i} \|\vec{x_j} - \vec{\mu_i}\|_2,\ \ \bar{C} = \frac{1}{k} \sum_{c_i \in C} \bar{c_i}$$

- La validez de una partición C está sujeta al cumplimiento de las siguientes restricciones:
  - $c_i \neq \emptyset\ \forall i \in \{1, ..., k\}$ (no hay clusters vacíos)
  - Propiedades de una partición: $c_i \cap c_j = \emptyset\ \forall i, j \in \{1, ..., k\}, i \neq j, \cup_{c_i \in C} c_i = \{1, ..., n\}$
- Para cada partición, calculamos su *infeasibility* como el número de restricciones incumplidas:

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

[1] No aparecen todos los elementos, sino los más imprescindibles. Seguimos la notación del seminario y del guion de la práctica.

II | Algoritmos implementados

En esta práctica hemos implementado tres algoritmos adicionales, dos genéticos y uno memético, con diversas variaciones.

1. Representación

Para resolver nuestro problema de agrupamiento de restricciones, haremos uso de cuatro tipos de estructuras:

- **Datos.** Es una matriz `datos` de valores reales de tamaño $n \times d$ que representa el conjunto de datos para cada problema, X.
- **Restricciones.** Dispondremos de dos estructuras para representar las restricciones, con el objetivo de ganar eficiencia:
  - Como una matriz `restr_mat` de dimensión $n \times n$ de modo que

$$\texttt{restr\_mat}[i,j] = \begin{cases} -1 & \text{si} \quad \vec{CL}_{[i,j]} \\ 1 & \text{si} \quad \vec{ML}_{[i,j]} \\ 0 & \text{en otro caso} \end{cases}$$

  - Como una lista `restr_mat` que, para cada restricción, almacena la 3-upla:

$$(i, j, r(i,j)), \quad \text{donde} \quad r(i,j) = 1 \text{ si } \exists \vec{ML}_{[i,j]} \quad \text{y} \quad r(i,j) = -1 \text{ si } \exists \vec{CL}_{[i,j]}$$

- **Solución.** Cada solución es una partición C que representamos como una lista de tamaño n con

$$\texttt{solucion}[i] = j \iff j \in c_i$$

- **Cromosomas.** Para los algoritmos genéticos, cada cromosoma de la población será un conjunto de posibles soluciones del problema (tal como hemos definido anteriormente). El *fitness* de cada cromosoma será almacenado en una lista con correspondencia uno-a-uno (cromosoma-valor de f).

2. Métodos comunes

A continuación se detallan ciertos métodos que se usarán en todos los algoritmos de las prácticas, y que de ahora en adelante referenciaremos continuadamente.

Número de datos por cluster: `contar_por_cluster`

Esta función calcula una lista con el número de datos (puntos) de cada cluster para una partición concreta.

Algoritmo (`contar_por_cluster`). Número de datos por cluster

Input: *solucion* (vector solución).
Output: *per_cluster* (lista de tamaño k, con el número de datos por cluster en cada posición).

```
per_cluster ←{0, 0, ..., 0}  (k)
for i in solucion do
    per_cluster[i] ←per_cluster[i] + 1;
end
return per_cluster;
```

Comprobar si solución es válida: `es_solucion`

Esta función comprueba si una solución cumple las restricciones. Tal y como está implementado el código, la única propiedad que debemos comprobar es si todos los clusters tienen al menos un elemento. Esto es bastante inmediato.

Algoritmo (`es_solucion`). Comprobar si solución es válida

Input: *solucion* (vector solución).
Output: *es_valida* (booleano, *True* si solución es válida, *False* en caso contrario).

```
es_valida ←contar_por_cluster(solucion).count(0) = 0;
return es_valida;
```

Generar solución: `generar_solucion`

Esta función genera una solución válida, aleatoriamente. *int_aleatorio*(a, b) es una función que genera un número entero aleatoriamente en el rango $[a, b)$, *shuffle*(*list*) es una función que permuta los elementos de *list* aleatoriamente.

Algoritmo (`generar_solucion`). Generar solución

Output: *solucion* (vector solución válido).

```
1 solucion ←{1, 2, ..., k};
2 for i in {k, ..., n − 1} do
3 |   solucion.append(int_aleatorio(0, k));
4 end
5 shuffle(solucion);
6 return solucion;
```

Reparar solución: `reparar_solucion`

Esta función repara una solución (hace que todos los clusters tengan al menos un elemento), asignando un dato aleatorio a un centroide que no tenga ninguno, hasta que la solución sea factible.

Algoritmo (`reparar_solucion`). Reparar solución

Input: *solucion* (vector solución, es **modificado** y hecho válido).

```
1 per_cluster ←contar_por_cluster(solucion);
2 index ←per_cluster.index(0) if (0 in per_cluster) else -1;
3 while index ≥ 0 do
4 |   to_cluster ←int_aleatorio(0, n);
5 |   per_cluster[index] ←per_cluster[index] + 1;
6 |   per_cluster[solucion[to_cluster]] ←per_cluster[solucion[to_cluster]] − 1;
7 |   solucion[to_cluster] ←index;
8 |   index ←per_cluster.index(0) if (0 in per_cluster) else -1;
9 end
```

La función *index*(*i*) usada en las líneas 2 y 8 tiene el comportamiento de la función homónima de la clase *list* de Python, que devuelve la primera posición de la lista donde se encuentra el valor *i*. En caso de que no esté en el array, se almacenará el valor -1 (en cuyo caso pararemos, dado que no hay clusters vacíos).

En el bucle *while* tratamos de evitar llamadas innecesarias a *contar_por_cluster*, dado que es de complejidad lineal. Para ello, decrementamos e incrementamos en 1 los valores de *per_cluster* in-situ.

Cálculo de centroides: `calcular_centroides`

Esta función calcula los centroides de cada cluster (tal y como se indica en la descripción del problema). Se usa una lista para contar el número de datos de cada centroide (como en *contar_per_cluster*, aunque en este caso está integrada en la función para una mayor eficiencia) y poder hacer la media.

Algoritmo (`calcular_centroides`). Cálculo de centroides

Input: *solucion* (vector solución, es **modificado** y hecho válido).
Output: *centroides* (lista de k centroides, de d reales cada uno).

```
1  centroides ←{{0.0, ...(d), 0.0}, ...(k), {0.0, ...(d), 0.0}};
2  count ←{0, ...(k), 0};
3  for i in {0, 1, ..., len(solucion) − 1} do
4  |  count[solucion[i]] ←count[solucion[i]] + 1;
5  |  for coord in {0, 1, ..., d − 1} do
6  |  |  centroides[solucion[i]][coord] ←centroides[solucion[i]][coord] + datos[i][coord];
7  |  end
8  end
9  for i in {0, 1, ..., k − 1} do
10 |  for j in {0, 1, ..., d − 1} do
11 |  |  centroides[i][j] ←centroides[i][j]/count[i];
12 |  end
13 end
14 return centroides;
```

Distancia media intra-cluster: `dmic`

Esta función calcula, dado un vector solución, la distancia media intra-cluster para un cierto cluster. A esta función también le pasaremos los centroides, dado que el recalcularlos en cada llamada sería poco eficiente.

Algoritmo (`dmic`). Distancia media intra-cluster

Input: *id_cluster* (entero positivo, número identificador del cluster),
solucion (vector solución), *centroides* (vector de centroides, calculados con *calcular_centroides*).
Output: *dmic* (real, distancia media intra-cluster para el cluster *cluster* dada *solucion*).

```
1  s ←{};
2  count ←{0, ...(k), 0};
3  for i in {0, 1, ..., n − 1} do
4  |  if solucion[i] = id_cluster then
5  |  |  s.append(distancia_euclidea(datos[i],centroides[id_cluster]));
6  |  end
7  end
8  return avg(s);
```

Desviación general: dg

Esta función calcula la desviación general para un vector solución.

> **Algoritmo** (dg). Desviación general
>
> **Input:** *solucion* (vector solución).
> **Output:** *dg* (real, desviación general de *solucion*).
>
> 1 *centroides* ←calcular_centroides(*solucion*);
> 2 **return** avg([dmic(i, *solucion*, *centroides*) **for** i **in** $\{0, 1, ..., k-1\}$]);

Nótese que aquí (concretamente en la línea 2) usamos una sintaxis muy eficiente y visual: los *list comprehension*. Lo usaremos más adelante en la memoria. Éstos tienen la siguiente estructura:

[*expression* **for** *item* **in** *iterable* **if** *condition*]

Verificar restricción: v

Esta función devuelve un valor booleano que indica, dado un vector *solucion*, si la restricción *restr* (en el formato especificado en la lista de restricciones, es decir, $\{i, j, r(i, j)\}$, ver descripción del problema) se verifica.

> **Algoritmo** (v). Verificar restricción
>
> **Input:** *solucion* (vector solución), *restr* (3-upla).
> **Output:** *se_verifica* (booleano, *True* si se verifica la restricción, *False* en caso contrario).
>
> 1 $r_0, r_1, r_2 \leftarrow$*restr*[0], *restr*[1], *restr*[2];
> 2 $c_0, c_1 \leftarrow$*solucion*[r_0], *solucion*[r_1];
> 3 **return** $c_0 \geq 0$ **and** $c_1 \geq 0$ **and** (($r_2 = -1$ **and** $c_0 \neq c_1$) **or** ($r_2 = -1$ **and** $c_0 = c_1$));

Calcular *infeasibility*: `infeasibility`

Esta función devuelve la *infeasibility* dado un vector solución.

> **Algoritmo** (`infeasibility`). Calcular *infeasibility*
>
> **Input:** *solucion* (vector solución).
> **Output:** *inf* (entero).

```
1 inf ←0;
2 for restr in restr_list do
3 |   if v(solucion, restr) then
4 |   |   inf ←inf + 1;
5 |   end
6 end
7 return inf;
```

Calcular *fitness*: `f`

Esta función calcula el fitness de una solución. Además, incrementa el valor del atributo *evals*, que usaremos para el criterio de parada.

Algoritmo (`f`). Calcular *fitness*

Input: *solucion* (vector solución).
Output: f (real).

```
1 evals ←evals + 1;
2 return dg(solucion) + infeasibility(solucion)*λ;
```

3. Consideraciones generales

Todos los algoritmos implementados comparten ciertas características que comentaremos a continuación.

En primer lugar, todos los algoritmos se instancian mediante una clase que hereda de la clase `PAR`, que contiene todos los métodos comunes descritos anteriormente. El constructor de todas las clases correspondientes a cada algoritmo siempre tiene los mismos parámetros, necesarios para cargar los datos e inicializar ciertas variables para nuestro problema. Para ejecutar los algoritmos, hay que llamar a la función `ejecutar_algoritmo`, cuyos parámetros varían en función del algoritmo a ejecutar (se detallarán individualmente).

Todas las clases tienen como atributo a *evals*, un contador que se incrementa en una unidad por cada evaluación de f (la función *fitness*). Saldremos del bucle cuando *evals* supere el número máximo de evaluaciones, *max_evals*.

4. Algoritmos genéticos

En la práctica 2, implementamos diversas variantes del algoritmo genético. El cuerpo de los algoritmos genéticos de esta práctica se detalla a continuación.

Algoritmo. Algoritmo Genético (implementación general)

Input: *max_evals* (entero positivo, número máximo de evaluaciones),
tam_poblacion (tamaño de población en cada generación),
tam_seleccion (número de cromosomas que se seleccionarán para la población intermedia),
p_cruce (probabilidad de cruce), *p_mutacion* (probabilidad de mutación),
cruce (función de cruce),
elitismo (booleano,*True* si queremos elitismo y *False* en caso contrario).
Output: *solucion* (mejor solución encontrada), *info* (información de ejecución).

```
1  t ←0;
2  evals ←0;
3  pob ←generar_poblacion_inicial();
4  fs ←evaluar(pob);
5  solucion, f_solucion ←mejor_de(pob, fs);
6  while evals ≤ max_evals do
7      t ←t + 1;
8      pob_new ←seleccionar(pob, fs, tam_seleccion);
9      cruzar(pob_new, p_cruce, cruce);
10     mutar(pob_new, p_mutacion);
11     pob ←reemplazar(pob, pob_new, fs, solucion, elitismo);
12     fs ←evaluar(pob);
13     best_solucion, f_best_solucion ←mejor_de(pob, fs);
14     if f_best_solucion < f_solucion then
15         solucion ←best_solucion;
16         f_solucion ←f_best_solucion;
17     end
18 end
```

Donde las funciones auxiliares se especifican a continuación (nótese que en la implementación no las hemos definido explícitamente, pero sí lo hacemos aquí para que el algoritmo genético general sea más legible).

Algoritmo (`generar_poblacion_inicial`).

Output: *pob* (población inicial, generada aleatoriamente).

```
1 pob ←{};
2 for i in {0, 1, ..., tam_poblacion} do
3 |   pob.append(generar_solucion());
4 end
5 return pob;
```

Algoritmo (`evaluar`).

Input: *pob* (población).
Output: *fs* (vector de *fitnesses* para cada cromosoma de la población).

```
1 return [f(ind) for ind in pob];
```

Algoritmo (`mejor_de`).

Input: *pob* (población), *fs* (vector de *fitnesses* para cada cromosoma de la población).
Output: *best* (mejor individuo de la población), *f_best* (*fitness* del mejor individuo de la población).

```
1 i_min ←min(fs);
2 best ←pob[i_min];
3 f_best ←fs[i_min];
4 return best, f_best;
```

Procederemos a analizar con más detalle cada uno de los operadores que se han especificado, para a continuación concretar los tipos de algoritmos genéticos que vamos a analizar (nótese que podemos implementar cada uno de los algoritmos genéticos simplemente variando los parámetros de `ejecutar_algoritmo`, que afectarán a las entradas de los operadores que vamos a detallar).

4.1. Operadores

Operador de selección de padres

Para la selección de padres, implementamos el torneo binario: seleccionamos aleatoriamente dos individuos de la población, y los comparamos usando f, insertando en la población aquel que tenga un f menor. Repetimos este procedimiento hasta obtener *tam_seleccion* padres.

Algoritmo (`seleccionar`). Selección de padres

Input: *pob* (población), *fs* (*fitnesses* de *pob*), *tam_seleccion* (número de padres a seleccionar).
Output: *padres* (población de padres seleccionados).

```
1 padres ←{};
2 for i in {0, 1, ..., tam_seleccion − 1} do
3 |   ind_1, ind_2 ←int_aleatorio(0, len(pob)), int_aleatorio(0, len(pob));
4 |   ind_best ←ind_1 if fs[ind_1] < fs[ind_2] else ind_2;
5 |   padres.append(pob[ind_best]);
6 end
7 return padres;
```

Aquí también hemos usado en la línea 4 la sintaxis de un operador de Python para hacer el algoritmo más legible: "*a* **if** *cond* **else** *b*" devolverá *a* si se verifica *cond*, y *b* en caso contrario.

Operador de cruce

En el caso del cruce, implementaremos dos operadores diferentes:

- **Cruce uniforme.** Tomamos la mitad de genes de un padre y la otra mitad del otro, aleatoriamente.
- **Cruce por segmento fijo.** Produce un nuevo individuo en base a dos padres, seleccionando de uno de ellos un segmento continuo de características y copiándolo sin modificación a la descendencia. Los genes que quedan por asignar combinan de manera uniforme características de ambos padres. Es un operador sesgado, pues siempre se seleccionan más genes del padre que se elige como portador del segmento. Nosotros siempre escogeremos el mismo (el primero), para dar más aleatoriedad.

Especificaremos cómo hemos implementado ambos operadores:

Algoritmo (`cruce_uniforme`). Cruce uniforme

Input: $padre_1$ (cromosoma), $padre_2$ (cromosoma).
Output: *hijo* (cromosoma).

```
asignaciones ←{1, ^{n/2)}..., 1, 2, ^{n/2)}..., 2};
shuffle(asignaciones);
hijo ←{};
for asignacion in asignaciones do
    if asignacion = 1 then
        hijo.append(padre_1[i]);
    else
        hijo.append(padre_2[i]);
    end
end
reparar_solucion(hijo);
return hijo;
```

Nótese que *reparar_solucion* dejará intacta a *solucion* si es válida. Esto es deseable, ya que comprobar si la solución es válida primero para repararla a continuación es más ineficiente.

Algoritmo (`cruce_segmento_fijo`). Cruce por segmento fijo

Input: $padre_1$ (cromosoma), $padre_2$ (cromosoma).
Output: *hijo* (cromosoma).

```
v, corte ←int_aleatorio(0, n), int_aleatorio(0, n);
hijo ←{None, ^{n)}..., None};
i ←corte;
for i in {0, 1, ..., v − 1} do
    hijo[i] ←padre_1[i];
    i ←(i + 1) %n;
end
while not hijo[i] do
    asignacion ←int_aleatorio(1,3);
    hijo[i] ←padre_1[i] if asignacion = 1 else padre_2[i];
    i ←(i + 1) %n;
end
reparar_solucion(hijo);
return hijo;
```

Ya solo basta hacer uso de estos operadores, que generan un hijo cada vez, para obtener la población de hijos deseada. Esto será lo que hará la función *cruzar*. Además, tendremos en cuenta que, para aumentar la eficiencia, utilizaremos el número esperado de cruces, cruzando los primeros $p_{cruce} \cdot \text{len}(pob)/2$ pares (podemos siempre usar los primeros porque ya están dispuestos de forma aleatoria).

Operador de mutación

Emplearemos el operador de mutación unforme. A continuación, la implementación de dicho operador para mutar un gen concreto.

> **Algoritmo** (`mutacion_uniforme`). Mutación uniforme de un gen
>
> **Input:** *pob* (población, es **modificada**), *crom* (entero positivo, cromosoma a mutar), *gen* (entero positivo, gen a mutar).
>
> ```
> 1 cambio ←int_aleatorio(0, k);
> 2 pob[com][gen] ←cambio;
> 3 while not es_solucion(pob[crom]) do
> 4 |   pob[crom][gen] ←(pob[crom][gen] + 1) % k;
> 5 end
> ```

En este caso estamos haciendo una reparación in-situ (no ejecutamos *reparar_solucion* para que sea más eficiente).

Teniendo en cuenta que tenemos una con una cierta probabilidad $p_{mutacion}$ que mute cada gen, para optimizar el programa, de nuevo usaremos la esperanza matemática en cada ejecución de *mutar*, y mutaremos el número esperado de genes de la población, que asciende a $E_m = p_{mutacion} \cdot n \cdot \text{len}(pob)$ genes.

En caso de que el número esperado de genes E_m se encuentre en $(0, 1)$, mutaremos un individuo en caso de que se verifique el siguiente suceso aleatorio: generamos un número real R en $(0, 1)$, y es $R < E$.

Una vez que conocemos cuántos individuos hemos de mutar, generaremos una tupla por mutación, que indicará el cromosoma y gen que queremos mutar. Finalmente, hacemos uso de *mutacion_uniforme* para mutar dicho gen.

Operador de reemplazamiento

Para poder obtener una nueva generación de individuos nos queda la última fase: la de reemplazamiento. La idea principal es poder cambiar la población por los nuevos individuos, ya sea parcial o totalmente. Esto dependerá del tipo de algoritmo genético que utilicemos. A continuación, los operadores de reemplazamiento para cada tipo.

En el caso de AGG reemplazamos la población por la nueva generada, e introduciendo la mejor solución hasta el momento en caso de querer elitismo.

> **Algoritmo** (`reemplazar`). Operador de reemplazamiento (generacional, AGG)
>
> **Input:** *pob* (generación anterior) *pob_new* (población tras mutación), *fs* (*fitnesses* de *pob*),

solucion (mejor individuo hasta el momento), *elitismo* (*True* si queremos elitismo)
Output: *pob_new* (nueva generación, tras reemplazamiento)

```
1 if elitismo then
2 |   pob_new[–1] ←solucion;
3 end
4 return pob_new;
```

En el caso de AGE (siempre elitista) elegimos de los dos peores valores de *pob* y los dos mejores de *pob_new* aquellos que tengan menor f, y los sustituimos en *pob* por los dos peores valores (es una forma sencilla de implementar el árbol de decisión sugerido).

Algoritmo (`reemplazar`). Operador de reemplazamiento (estacionario, AGE)

Input: *pob* (generación anterior) *pob_new* (población tras mutación), *fs* (*fitnesses* de *pob*).
Output: *pob_new* (nueva generación, tras reemplazamiento)

```
1 two_worst ←seleccionar_dos_peores(fs);
2 if elitismo then
3 |   pob_new[–1] ←solucion;
4 end
5 return pob_new;
```

Donde *seleccionar_dos_peores* y *seleccionar_dos_mejores* devuelven los índices de los dos peores y los dos mejores valores de las listas pasadas, respectivamente.

4.2. Variantes

Una vez vistos los elementos de un algoritmo genético, veamos cómo podemos implementar los algoritmos pedidos a partir de ellos.

4.2.1. Algoritmo Genético Generacional (AGG)

Los AGGs reemplazan en cada generación (casi) completamente a la población que ha sido seleccionada, cruzada y mutada. Dado que la idea principal de los algoritmos genéticos es recombinar soluciones para crear otras nuevas, no nos esforzaremos en guardar la población anterior, excepto en el caso de la mejor solución en caso de que queramos hacer el algoritmo elitista (como es el caso de esta práctica). Este elitismo permite al algoritmo obtener mejores soluciones, pues propicia la reproducción de aquellos individuos que tengan mejor *fitness*.

Lo interesante de este tipo de algoritmos es que, a diferencia de la búsqueda local, los operadores

de selección, cruce y mutación nos permiten una mayor exploración del espacio de soluciones, evitando caer en mínimos locales. El contraste de exploración vs. explotación podrá hacerse ajustando los parámetros de probabilidades y tamaños, así como el elitismo (conseguimos un algoritmo más explorador si somos menos elitistas).

En este caso, deberemos usar los siguientes parámetros:

- En el operador de **selección**, seleccionamos tantos individuos como individuos tenga la población (en nuestro caso 50).
- En el operador de **cruce**, tendremos en cuenta la p_{cruce}, tal y como se especificó anteriormente (en nuestro caso será 0.7, luego el número esperado de parejas de padres que se cruzarán será de $\lfloor 0.7 \cdot 50/2 \rfloor = 17$ parejas).
- En el operador de **mutación**, tendremos en cuenta la $p_{mutacion}$ (en nuestro caso es $0.1/n$, luego el número esperado de genes a mutar es de $\lfloor 0.1/n \cdot n \cdot 50 \rfloor = 5$ genes).
- Usamos el operador de **reemplazamiento** especificado anteriormente.

Finalmente usaremos dos variantes, dependiendo de los operadores de cruce, obteniendo los algoritmos **AGG-UN** (con operador de cruce uniforme) y **AGG-SF** (con operador de cruce por segmento fijo).

4.2.2. Algoritmo Genético Estacionario (AGE)

Los AGEs son muy similares a los AGGs, sólo que seleccionamos dos padres (de nuevo por torneo binario), que siempre cruzaremos y mutaremos atendiendo al mismo criterio probabilístico que en el caso de los AGGs. Además, también variamos en el operador de reemplazamiento, pues mantenemos la misma población que en la generación anterior excepto por los dos peores individuos, que competirán con los nuevos para formar parte en la nueva generación (como ya hemos especificado). Este algoritmo es, por tanto, elitista, pues siempre estará presente la mejor solución en la población, y mantendremos las mejores soluciones generación tras generación: para poder insertar nuevos individuos (soluciones) a la población, éstos deberán ser considerablemente buenos.

En este caso, deberemos usar los siguientes parámetros:

- En el operador de **selección**, seleccionamos únicamente dos individuos.
- En el operador de **cruce**, hacemos $p_{cruce} = 1$ para que siempre cruce el único par de padres (ya que el número de pares de padres esperado con esa probabilidad es de $\lfloor 1 \cdot 2/2 \rfloor = 1$ parejas).
- En el operador de **mutación**, tendremos en cuenta la $p_{mutacion}$ (en nuestro caso es $0.1/n$, luego el número esperado de genes a mutar es de $\lfloor 0.1/n \cdot n \cdot 2 \rfloor = 0.2$ genes).
- Usamos el operador de **reemplazamiento** especificado anteriormente.

Del mismo modo que para AGG usaremos dos variantes, dependiendo de los operadores de cruce, obteniendo los algoritmos **AGE-UN** (con operador de cruce uniforme) y **AGE-SF** (con operador de cruce por segmento fijo).

5. Algoritmo memético (AM)

Los algoritmos meméticos son muy similares a los genéticos, pero con una mejora: en un momento determinado, se aplica un algoritmo de búsqueda local para mejorar la población. De este modo implementamos más explotación, pudiendo lograr un equilibrio más fino entre exploración y explotación, consiguiendo mejores soluciones y sin caer en óptimos locales. Es, por tanto, una *hibridación* de dos algoritmos: AG y BL.

En esta práctica he implementado AM a partir de AGG, dado que es el que mejor resultados ha proporcionado. Los análisis los haremos a partir de AGG-UN (que da mejores resultados que AGG-SF), aunque dada la modularidad del código es posible usar AGG-SF sin ningún inconveniente. También sería sencillo implementarlo haciendo uso de AGE-UN y AGE-SF.

Veamos el algoritmo de AM una vez comentado todo esto. Nótese que en el *input* hemos añadido parámetros que comentaremos en breve.

Algoritmo. Algoritmo Memético (implementación general)

Input: *max_evals* (entero positivo, número máximo de evaluaciones),
tam_poblacion (tamaño de población en cada generación),
tam_seleccion (número de cromosomas que se seleccionarán para la población intermedia),
p_cruce (probabilidad de cruce), *p_mutacion* (probabilidad de mutación),
cruce (función de cruce),
elitismo (booleano,*True* si queremos elitismo y *False* en caso contrario),
gens_ls (número de generaciones tras las que aplicar SLS),
prob_ls (probabilidad de aplicar SLS a cada cromosoma),
mej_ls (*True* si se decide aplicar SLS a los mejores cromosomas),
lim_fallos_ls (límite de fallos para SLS).
Output: *solucion* (mejor solución encontrada), *info* (información de ejecución).

```
1  t ←0;
2  evals ←0;
3  pob ←generar_poblacion_inicial();
4  fs ←evaluar(pob);
5  solucion, f_solucion ←mejor_de(pob, fs);
6  while evals ≤ max_evals do
7      t ←t + 1;
8      pob_new ←seleccionar(pob, fs, tam_seleccion);
9      cruzar(pob_new, p_cruce, cruce);
10     mutar(pob_new, p_mutacion);
11     local_boost(pob_new, gens_ls, p_ls, mej_ls, lim_fallos_ls);
12     pob ←reemplazar(pob, pob_new, fs, solucion, elitismo);
13     fs ←evaluar(pob);
14     best_solucion, f_best_solucion ←mejor_de(pob, fs);
15     if f_best_solucion < f_solucion then
16         solucion ←best_solucion;
17         f_solucion ←f_best_solucion;
18     end
19 end
```

Simplemente hemos incorporado una línea al algoritmo genético: la línea 11, encargada de realizar la mejora con BL en caso de que sea necesario. Veámoslo.

5.1. Soft Local Search (SLS)

En lugar de usar el algoritmo de búsqueda local implementado para la práctica 1, usamos uno un poco más suave, que mezcla BL y greedy, pues mejora levemente los cromosomas sin buscar de forma excesivamente profunda (agotando las evaluaciones). La búsqueda local procede como sigue:

Algoritmo. Soft Local Search (SLS)

Input: *max_evals* (entero positivo, número máximo de evaluaciones),
tam_poblacion (tamaño de población en cada generación),
tam_seleccion (número de cromosomas que se seleccionarán para la población intermedia),
p_cruce (probabilidad de cruce), *p_mutacion* (probabilidad de mutación),
cruce (función de cruce),
elitismo (booleano,*True* si queremos elitismo y *False* en caso contrario),
gens_ls (número de generaciones tras las que aplicar SLS),
prob_ls (probabilidad de aplicar SLS a cada cromosoma),

mej_ls (*True* si se decide aplicar SLS a los mejores cromosomas),
lim_fallos_ls (límite de fallos para SLS).

Output: *solucion* (mejor solución encontrada), *info* (información de ejecución).

```
1  RSI ←shuffle({0, 1, ..., n − 1});
2  f_solucion ←f(solucion);
3  fallos ←0;
4  mejora ←True;
5  i ←0;
6  while (mejora or fallos < lim_fallos) and i < n and evals ≤ max_evals do
7      mejora ←False;
8      fs ←{f_solucion + 1, ..^k) f_solucion + 1};
9      cc ←contar_por_cluster(solucion);
10     for j in {0, 1, ..., k − 1} do
11         if cc[j] > 1 and cc[solucion[RSI[i]]] > 1 then
12             solucion_aux ←solucion;
13             solucion_aux[RSI[i]] ←j;
14             fs[j] ←f(solucion_aux);
15         end
16         best_cluster = min(fs);
17         if fs[best_cluster] < f_solucion then
18             solucion[RSI[i]] ←best_cluster;
19             f_solucion ←fs[best_cluster];
20             mejora ←True;
21         else
22             fallos ←fallos + 1;
23         end
24         i ←i + 1;
25     end
26 end
```

Deberemos ejecutar SLS sobre los cromosomas de la población, atendiendo a ciertos criterios que comentamos a continuación.

5.2. Variantes

Aplicaremos SLS teniendo en cuenta los siguientes factores:

- **Número de generaciones** tras las que aplicar SLS. Sólo intentaremos aplicar SLS una vez que haya pasado un cierto número de generaciones, para no explotar demasiado las soluciones.

Lo especificamos con el parámetro *gens_ls*.

- **Probabilidad** de aplicar SLS a cada cromosoma. Como en otros casos probabilísticos, usaremos la esperanza matemática para una mayor eficiencia. Lo especificaremos con *prob_ls*.
- Aplicar a los **mejores cromosomas** o aleatoriamente. Podremos aplicar SLS sobre los mejores cromosomas de la población o sobre un subconjunto cualquiera, detallaremos esta posibilidad mediante *mej_ls*.
- Detallaremos un **límite de fallos** (decimos que fallamos si no mejoramos la solución) para parar el algoritmo.

Todos estos factores se tienen en cuenta en la función siguiente, que es la que aparece en el cuerpo del algoritmo memético:

Algoritmo (`local_boost`). Boost para AM con SLS

Input: *pob* (población de cromosomas, es **modificado**),
gens (número de generaciones tras las que aplicar SLS),
prob (probabilidad de aplicar SLS a cada cromosoma),
mej (*True* si se decide aplicar SLS a los mejores cromosomas),
lim_fallos (límite de fallos para SLS).

```
1  if t %gens = 0 then
2      if mej then
3          fs ←[(i, f(ind)) for ind in enumerate(pob)];
4          fs.ordenar(ascendente y usando primera posición como clave);
5          seleccionar ←las primeras posiciones de los ⌊len(pob)*prob⌋ primeros elementos de fs;
6      else
7          seleccionar ←subconjunto aleatorio de ⌊len(pob)*prob⌋ elementos de
             {0, 1, ..., len(pob) – 1};
8      end
9      for i in seleccionar do
10         soft_local_search(pob[i], lim_fallos);
11     end
12 end
```

Ahora estamos en condiciones de especificar los AM con los que trabajaremos:

- **AM-(10,1.0):** *gens_ls* = 10, *prob_ls* = 1 (siempre aplica SLS a los cromosomas seleccionados), *mejores_ls* = False (no aplica SLS sobre los mejores seleccionados).
- **AM-(10,0.1):** *gens_ls* = 10, *prob_ls* = 0.1, *mejores_ls* = False (no aplica SLS sobre los mejores seleccionados).
- **AM-(10,0.1mej):** *gens_ls* = 10, *prob_ls* = 0.1, *mejores_ls* = True (aplica SLS sobre los mejores seleccionados).

III | Desarrollo y ejecución

Para el desarrollo de esta práctica he usado el lenguaje de programación Python, usando librerías incorporadas al sistema (comencé usando otras adicionales como *numpy*, pero consumía considerablemente más recursos).

Ejecución del programa

Para ejecutar el programa, use Python 3 y ejecute el script `main.py`. Este script hace uso de `argparse` para poder pasar valores, por lo que simplemente inserte en la consola

```
python main.py -h
```

y un texto de ayuda con todos los comandos posibles aparecerá en pantalla. A continuación algunos ejemplos:

- Ejecutar 4 repeticiones de AGG-UN usando el set `bupa` con 10 % de restricciones:
  ```
  python main.py -N bupa -P 10 -A agg -fc un -R 4
  ```
- Ejecutar 5 repeticiones de AM-UN(10, 0.1mej) usando el set `glass` con 20 % de restricciones, e imprimir los resultados en un fichero CSV llamado `./resultados.csv`:
  ```
  python main.py -N glass -P 20 -A am -fc un -R 5 -pl 0.1 -ml
  ```
- Ejecutar 5 repeticiones de AGE-SF usando el set `zoo` con 10 % de restricciones, e imprimir los resultados en un fichero CSV llamado `./resultados.csv`:
  ```
  python main.py -N zoo -P 10 -A age -fc sf -R 5 -C -CF ./resultados.csv
  ```

IV | Análisis de resultados

1. Resultados de los experimentos

Tras ejecutar todas las iteraciones pedidas con diferentes sets de datos y restricciones, y usando la semilla por defecto (puede modificarse en los comandos del programa, pero por defecto es 1514280522468089), he obtenido los siguientes resultados:

	zoo				glass				bupa			
	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t
Ejecución 1	22	0.5925	0.6378	57.1657	205	0.1832	0.2778	116.2915	666	0.1525	0.2707	212.0975
Ejecución 2	23	0.5899	0.6372	56.5577	92	0.1882	0.2307	116.0726	654	0.1576	0.2737	216.2333
Ejecución 3	23	0.5438	0.5911	59.1848	85	0.1899	0.2292	117.0219	613	0.1486	0.2574	211.3598
Ejecución 4	29	0.6173	0.6770	56.6152	161	0.1877	0.2620	116.9504	698	0.1436	0.2675	211.6150
Ejecución 5	23	0.5949	0.6423	56.5805	180	0.2105	0.2936	113.5355	646	0.1509	0.2656	211.0249
Media	24.0000	0.5877	0.6371	57.2208	144.6000	0.1919	0.2587	115.9744	655.4000	0.1506	0.2670	212.4661
Desv.	2.8284	0.0269	0.0306	1.1266	53.5938	0.0107	0.0285	1.4237	30.8837	0.0051	0.0062	2.1420

Tabla 1. AGG-UN con 10 % de restricciones

	zoo				glass				bupa			
	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t
Ejecución 1	60	0.5913	0.6571	68.5291	140	0.2340	0.2682	155.8382	1348	0.1479	0.2725	324.5205
Ejecución 2	89	0.6090	0.7066	68.0159	224	0.1863	0.2410	156.0958	1272	0.1432	0.2607	331.2028
Ejecución 3	83	0.5358	0.6268	68.2232	166	0.1926	0.2331	175.1346	1557	0.1447	0.2886	333.8804
Ejecución 4	50	0.5814	0.6362	67.8987	205	0.2248	0.2749	160.0151	1299	0.1455	0.2655	335.9009
Ejecución 5	48	0.5826	0.6352	68.1055	211	0.1909	0.2425	155.6102	1384	0.1412	0.2691	337.5223
Media	66.0000	0.5800	0.6524	68.1545	189.2000	0.2057	0.2519	160.5388	1 372.0000	0.1445	0.2713	332.6054
Desv.	18.9341	0.0271	0.0323	0.2409	34.9814	0.0220	0.0184	8.3582	112.1093	0.0025	0.0106	5.0989

Tabla 2. AGG-UN con 20 % de restricciones

	zoo				glass				bupa			
	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t
Ejecución 1	27	0.5376	0.5932	57.2415	79	0.1884	0.2249	114.8167	748	0.1483	0.2811	213.8463
Ejecución 2	31	0.6070	0.6708	57.6380	77	0.1963	0.2318	114.6524	778	0.1510	0.2891	216.3507
Ejecución 3	23	0.5873	0.6346	57.8755	104	0.2331	0.2811	112.2363	725	0.1449	0.2736	216.6703
Ejecución 4	14	0.5859	0.6147	57.9323	97	0.1872	0.2319	111.8735	729	0.1498	0.2793	211.4569
Ejecución 5	18	0.5957	0.6328	56.8569	137	0.2251	0.2883	112.4865	627	0.1480	0.2593	211.2843
Media	22.6000	0.5827	0.6292	57.5088	98.8000	0.2060	0.2516	113.2131	721.4000	0.1484	0.2765	213.9217
Desv.	6.8044	0.0266	0.0286	0.4547	24.2734	0.0215	0.0305	1.4071	56.7741	0.0023	0.0111	2.5735

Tabla 3. AGG-SF con 10 % de restricciones

	zoo				glass				bupa			
	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t
Ejecución 1	118	0.5012	0.6305	67.7199	165	0.1890	0.2293	155.9800	1607	0.1404	0.2889	328.1550
Ejecución 2	57	0.5910	0.6535	68.1000	216	0.2047	0.2574	158.6826	1471	0.1412	0.2771	324.4157
Ejecución 3	57	0.5822	0.6447	67.8542	269	0.2196	0.2853	159.1273	1349	0.1442	0.2689	330.4098
Ejecución 4	66	0.5833	0.6557	67.6592	317	0.2126	0.2900	159.2276	1505	0.1444	0.2834	332.1087
Ejecución 5	105	0.5007	0.6159	68.0531	164	0.1913	0.2314	159.3561	1500	0.1459	0.2845	324.9037
Media	80.6000	0.5517	0.6401	67.8773	226.2000	0.2034	0.2587	158.4747	1 486.4000	0.1432	0.2805	327.9986
Desv.	28.8149	0.0464	0.0167	0.1958	66.6986	0.0132	0.0287	1.4174	92.4543	0.0023	0.0078	3.3596

Tabla 4. AGG-SF con 20 % de restricciones

	zoo				glass				bupa			
	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t
Ejecución 1	93	1.2030	1.3944	57.3517	548	0.3814	0.6343	110.4119	937	0.2359	0.4023	209.2977
Ejecución 2	102	1.0775	1.2874	57.5787	571	0.3541	0.6176	112.0981	955	0.2379	0.4074	211.6526
Ejecución 3	100	1.1294	1.3352	57.4528	583	0.3789	0.6479	113.1744	954	0.2294	0.3988	212.0487
Ejecución 4	93	1.1803	1.3717	57.6532	526	0.3760	0.6187	118.4978	972	0.2299	0.4025	212.7796
Ejecución 5	87	1.2465	1.4255	57.2799	593	0.3568	0.6305	112.7304	959	0.2290	0.3992	211.6402
Media	95.0000	1.1674	1.3628	57.4632	564.2000	0.3694	0.6298	113.3825	955.4000	0.2324	0.4020	211.4838
Desv.	6.0415	0.0656	0.0535	0.1546	27.1606	0.0129	0.0124	3.0459	12.5419	0.0042	0.0035	1.3067

Tabla 5. AGE-UN con 10 % de restricciones

	zoo				glass				bupa			
	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t
Ejecución 1	194	1.1029	1.3156	65.7340	1207	0.3596	0.6543	154.7032	1923	0.2214	0.3991	325.8906
Ejecución 2	213	1.2016	1.4351	67.4753	1152	0.3506	0.6319	154.1017	1996	0.2173	0.4017	330.4296
Ejecución 3	179	1.1854	1.3816	67.3662	1113	0.3434	0.6151	154.6239	1882	0.2292	0.4031	330.1140
Ejecución 4	236	0.9739	1.2327	65.9430	1142	0.3522	0.6310	157.0120	1914	0.2280	0.4048	331.3085
Ejecución 5	208	1.1476	1.3757	65.5735	1127	0.3636	0.6387	158.2996	1962	0.2177	0.3989	324.7475
Media	206.0000	1.1223	1.3481	66.4184	1 148.2000	0.3539	0.6342	155.7481	1 935.4000	0.2227	0.4015	328.4981
Desv.	21.3659	0.0912	0.0772	0.9251	36.0514	0.0079	0.0142	1.8148	44.2809	0.0056	0.0025	2.9625

Tabla 6. AGE-UN con 20 % de restricciones

	zoo				glass				bupa			
	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t
Ejecución 1	95	1.2866	1.4821	55.8708	589	0.4004	0.6722	111.1139	959	0.2309	0.4012	211.4993
Ejecución 2	95	1.2210	1.4164	56.1654	573	0.3768	0.6413	111.0041	989	0.2372	0.4128	213.4594
Ejecución 3	108	1.2536	1.4758	56.1451	590	0.3763	0.6486	110.6201	967	0.2405	0.4122	216.4060
Ejecución 4	92	1.1074	1.2967	55.9450	573	0.3984	0.6628	110.7179	986	0.2299	0.4049	216.8788
Ejecución 5	92	1.1177	1.3070	56.0183	583	0.3840	0.6530	110.7544	956	0.2339	0.4037	215.8599
Media	96.4000	1.1972	1.3956	56.0289	581.6000	0.3872	0.6556	110.8421	971.4000	0.2345	0.4069	214.8207
Desv.	6.6558	0.0808	0.0894	0.1268	8.2946	0.0116	0.0121	0.2077	15.2742	0.0044	0.0052	2.2756

Tabla 7. AGE-SF con 10 % de restricciones

	zoo				glass				bupa			
	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t
Ejecución 1	237	1.1187	1.3786	65.2834	1139	0.3940	0.6721	157.5524	1981	0.2264	0.4094	329.0791
Ejecución 2	214	1.2595	1.4942	65.3534	1188	0.3640	0.6540	157.8080	1940	0.2334	0.4126	348.3544
Ejecución 3	200	1.1094	1.3287	65.3639	1150	0.3513	0.6321	157.7215	1968	0.2375	0.4193	359.1324
Ejecución 4	222	1.0793	1.3227	65.7270	1214	0.3658	0.6621	157.3251	1963	0.2357	0.4170	358.5703
Ejecución 5	189	1.1515	1.3587	65.2917	1188	0.3509	0.6410	153.9955	1962	0.2281	0.4094	357.1484
Media	212.4000	1.1437	1.3766	65.4039	1 175.8000	0.3652	0.6523	156.8805	1 962.8000	0.2322	0.4136	350.4569
Desv.	18.7163	0.0697	0.0695	0.1842	30.7278	0.0175	0.0160	1.6232	14.8223	0.0048	0.0045	12.7210

Tabla 8. AGE-SF con 20 % de restricciones

	zoo				glass				bupa			
	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t
Ejecución 1	13	0.5201	0.5468	56.5949	349	0.2315	0.3925	122.5520	788	0.1908	0.3307	217.0311
Ejecución 2	17	0.5855	0.6205	58.0422	135	0.2299	0.2922	121.5838	787	0.2055	0.3452	214.0379
Ejecución 3	16	0.5210	0.5539	60.9728	253	0.2587	0.3754	119.6040	744	0.2022	0.3343	214.7066
Ejecución 4	23	0.5852	0.6326	56.9832	272	0.2365	0.3620	122.4085	737	0.2029	0.3337	219.7179
Ejecución 5	19	0.5835	0.6226	59.0416	274	0.2369	0.3633	121.5836	805	0.2041	0.3470	219.2140
Media	17.6000	0.5591	0.5953	58.3270	256.6000	0.2387	0.3571	121.5464	772.2000	0.2011	0.3382	216.9415
Desv.	3.7148	0.0352	0.0413	1.7605	77.2612	0.0116	0.0383	1.1758	29.9115	0.0059	0.0074	2.5645

Tabla 9. AM-(10,1.0) con 10 % de restricciones

	zoo				glass				bupa			
	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t
Ejecución 1	40	0.5738	0.6177	72.4609	303	0.2256	0.2996	164.8232	1577	0.1933	0.3390	338.4524
Ejecución 2	45	0.5821	0.6314	74.5165	473	0.2270	0.3425	165.1472	1456	0.1863	0.3208	338.3872
Ejecución 3	44	0.5805	0.6287	73.6048	213	0.2180	0.2700	162.0157	1603	0.1933	0.3414	336.1009
Ejecución 4	42	0.5794	0.6255	76.7455	176	0.2168	0.2598	161.1498	1550	0.1763	0.3195	336.4180
Ejecución 5	44	0.5811	0.6294	74.9778	380	0.2525	0.3452	160.3781	1462	0.1890	0.3241	336.1299
Media	43.0000	0.5794	0.6265	74.4611	309.0000	0.2280	0.3034	162.7028	1 529.6000	0.1876	0.3290	337.0977
Desv.	2.0000	0.0032	0.0054	1.5987	121.3239	0.0144	0.0397	2.1656	67.1513	0.0070	0.0104	1.2135

Tabla 10. AM-(10,1.0) con 20 % de restricciones

	zoo				glass				bupa			
	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t
Ejecución 1	16	0.5864	0.6194	61.8786	62	0.1850	0.2136	126.3394	272	0.1217	0.1700	213.6387
Ejecución 2	18	0.5860	0.6231	66.0939	59	0.1856	0.2129	123.4821	251	0.1082	0.1527	213.2842
Ejecución 3	19	0.5839	0.6230	62.9529	54	0.1872	0.2121	122.3885	283	0.1215	0.1718	215.5114
Ejecución 4	10	0.5814	0.6020	61.1323	54	0.1882	0.2131	124.3896	283	0.1161	0.1664	218.6842
Ejecución 5	15	0.5809	0.6117	61.1925	49	0.1830	0.2056	121.4098	277	0.1108	0.1600	215.6590
Media	15.6000	0.5837	0.6158	62.6500	55.6000	0.1858	0.2114	123.6019	273.2000	0.1157	0.1642	215.3555
Desv.	3.5071	0.0026	0.0090	2.0600	5.0299	0.0020	0.0033	1.8978	13.2363	0.0061	0.0078	2.1467

Tabla 11. AM-(10,0.1) con 10 % de restricciones

	zoo				glass				bupa			
	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t
Ejecución 1	45	0.5815	0.6309	74.0249	152	0.1827	0.2198	158.7437	511	0.1275	0.1747	329.0137
Ejecución 2	59	0.5873	0.6520	71.4608	186	0.1811	0.2265	158.2309	586	0.1128	0.1669	335.4318
Ejecución 3	41	0.5823	0.6272	76.4344	155	0.1814	0.2193	158.5480	669	0.1212	0.1830	333.4228
Ejecución 4	64	0.5855	0.6557	76.7454	171	0.1866	0.2284	162.1308	491	0.1082	0.1535	334.2275
Ejecución 5	50	0.5755	0.6303	75.9759	176	0.1859	0.2288	162.0322	452	0.1105	0.1523	331.1422
Media	51.8000	0.5824	0.6392	74.9283	168.0000	0.1835	0.2246	159.9371	541.8000	0.1160	0.1661	332.6476
Desv.	9.5760	0.0045	0.0135	2.2084	14.3353	0.0025	0.0047	1.9664	86.2073	0.0081	0.0133	2.5652

Tabla 12. AM-(10,0.1) con 20 % de restricciones

	zoo				glass				bupa			
	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t
Ejecución 1	17	0.5859	0.6209	60.4849	55	0.1887	0.2141	119.2926	308	0.1100	0.1647	211.4287
Ejecución 2	15	0.5858	0.6167	61.8156	49	0.1887	0.2113	124.0736	280	0.1120	0.1617	215.7236
Ejecución 3	28	0.5844	0.6420	63.1862	53	0.1877	0.2122	122.5665	307	0.1128	0.1673	220.5844
Ejecución 4	18	0.5399	0.5770	60.1830	56	0.1878	0.2136	124.4910	265	0.1092	0.1562	214.5471
Ejecución 5	14	0.5859	0.6147	60.7471	57	0.1888	0.2151	118.3778	254	0.1149	0.1600	211.0034
Media	18.4000	0.5764	0.6143	61.2834	54.0000	0.1883	0.2133	121.7603	282.8000	0.1118	0.1620	214.6575
Desv.	5.5946	0.0204	0.0235	1.2288	3.1623	0.0005	0.0015	2.7834	24.3660	0.0023	0.0043	3.8747

Tabla 13. AM-(10,0.1mej) con 10 % de restricciones

	zoo				glass				bupa			
	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t
Ejecución 1	71	0.5822	0.6601	72.8583	176	0.1869	0.2298	160.7078	501	0.1107	0.1570	333.8219
Ejecución 2	68	0.5808	0.6553	71.8995	190	0.1864	0.2328	159.1364	554	0.1134	0.1646	333.1788
Ejecución 3	37	0.5793	0.6198	74.4408	154	0.1935	0.2311	157.9068	534	0.1117	0.1611	339.7582
Ejecución 4	43	0.5780	0.6251	73.0153	177	0.1867	0.2299	158.8660	420	0.1152	0.1540	333.8764
Ejecución 5	88	0.4965	0.5930	73.0414	178	0.1868	0.2302	158.0743	441	0.1102	0.1509	334.4869
Media	61.4000	0.5634	0.6307	73.0510	175.0000	0.1881	0.2308	158.9383	490.0000	0.1122	0.1575	335.0244
Desv.	21.0784	0.0374	0.0276	0.9077	13.0384	0.0030	0.0012	1.1165	57.9957	0.0021	0.0055	2.6864

Tabla 14. AM-(10,0.1mej) con 20 % de restricciones

	zoo				glass				bupa			
	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t
COPKM	13.0000	1.0939	1.0900	0.3818	8.0000	0.3883	0.3879	1.3975	36.0000	0.2164	0.2167	14.4021
BL	24.0000	0.5602	0.6096	3.9671	71.2000	0.1873	0.2200	17.5575	576.0000	0.1440	0.2463	40.4819
AGG-UN	24.0000	0.5877	0.6371	57.2208	144.6000	0.1919	0.2587	115.9744	655.4000	0.1506	0.2670	212.4661
AGG-SF	22.6000	0.5827	0.6292	57.5088	98.8000	0.2060	0.2516	113.2131	721.4000	0.1484	0.2765	213.9217
AGE-UN	95.0000	1.1674	1.3628	57.4632	564.2000	0.3694	0.6298	113.3825	955.4000	0.2324	0.4020	211.4838
AGE-SF	96.4000	1.1972	1.3956	56.0289	581.6000	0.3872	0.6556	110.8421	971.4000	0.2345	0.4069	214.8207
AM-(10,1.0)	17.6000	0.5591	0.5953	58.3270	256.6000	0.2387	0.3571	121.5464	772.2000	0.2011	0.3382	216.9415
AM-(10,0.1)	15.6000	0.5837	0.6158	62.6500	55.6000	0.1858	0.2114	123.6019	273.2000	0.1157	0.1642	215.3555
AM-(10,0.1mej)	18.4000	0.5764	0.6143	61.2834	54.0000	0.1883	0.2133	121.7603	282.8000	0.1118	0.1620	214.6575

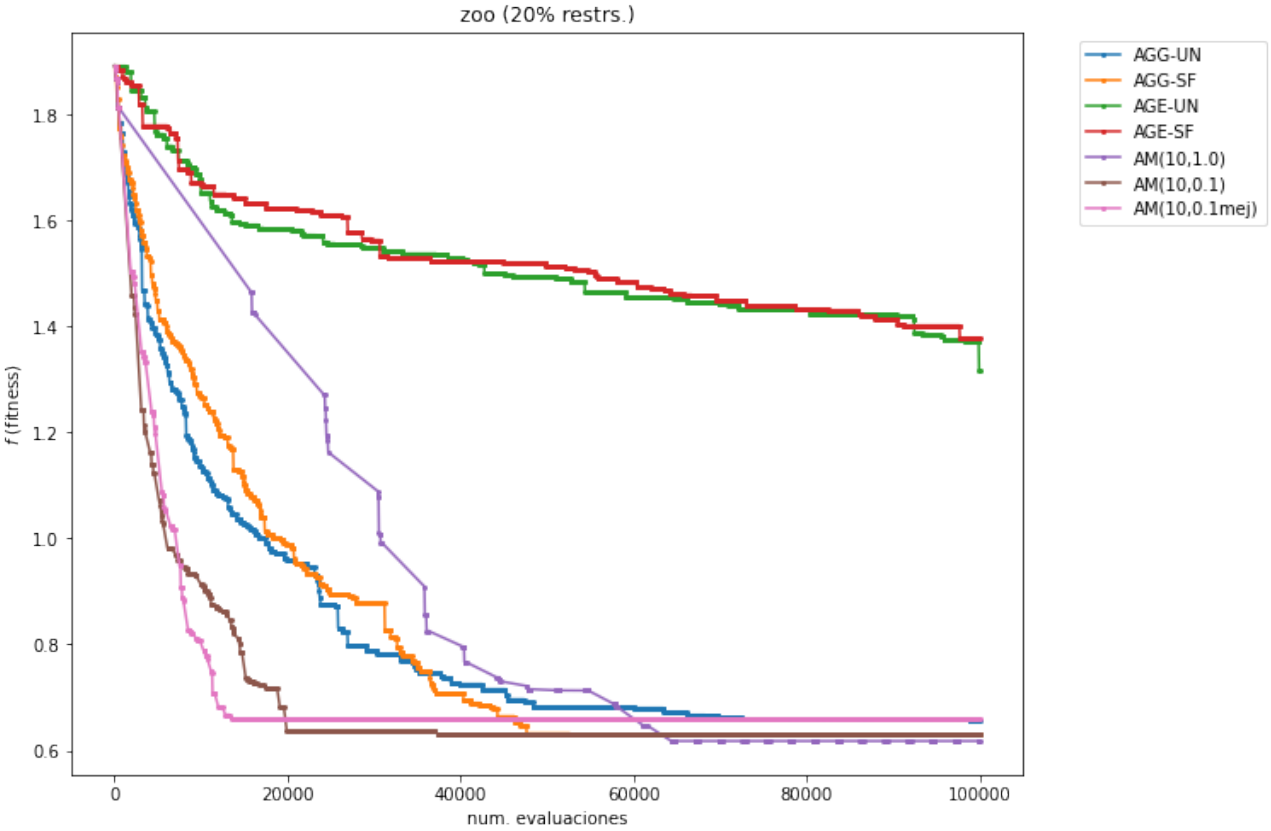
Tabla 15. Resultados globales con 10 % de restricciones

	zoo				glass				bupa			
	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t	T_{inf}	d_{intra}	Agr.	t
COPKM	0.2000	0.9048	0.9044	0.3235	0.6000	0.3557	0.3562	1.2311	1.0000	0.2391	0.2391	9.0563
BL	52.0000	0.5859	0.6429	3.4448	191.0000	0.1860	0.2326	20.2458	929.8000	0.1288	0.2147	53.4925
AGG-UN	66.0000	0.5800	0.6524	68.1545	189.2000	0.2057	0.2519	160.5388	1 372.0000	0.1445	0.2713	332.6054
AGG-SF	80.6000	0.5517	0.6401	67.8773	226.2000	0.2034	0.2587	158.4747	1 486.4000	0.1432	0.2805	327.9986
AGE-UN	206.0000	1.1223	1.3481	66.4184	1 148.2000	0.3539	0.6342	155.7481	1 935.4000	0.2227	0.4015	328.4981
AGE-SF	212.4000	1.1437	1.3766	65.4039	1 175.8000	0.3652	0.6523	156.8805	1 962.8000	0.2322	0.4136	350.4569
AM-(10,1.0)	43.0000	0.5794	0.6265	74.4611	309.0000	0.2280	0.3034	162.7028	1 529.6000	0.1876	0.3290	337.0977
AM-(10,0.1)	51.8000	0.5824	0.6392	74.9283	168.0000	0.1835	0.2246	159.9371	541.8000	0.1160	0.1661	332.6476
AM-(10,0.1mej)	61.4000	0.5634	0.6307	73.0510	175.0000	0.1881	0.2308	158.9383	490.0000	0.1122	0.1575	335.0244

Tabla 16. Resultados globales con 20 % de restricciones

2. Análisis de resultados

Para analizar los resultados, nos ayudaremos de los datos recopilados en las tablas anteriores y de la siguiente gráfica, que analiza la evolución de f conforme al número de iteraciones. Además, no hemos puesto los datos para cada iteración, sino únicamente cuando tenemos una generación nueva (tras el reemplazo). De esta forma podremos extraer algunas conclusiones interesantes que no podríamos extraer únicamente analizando los resultados finales.



Gráfica 1. Progreso de la ejecución en cada generación, zoo 20 % restricciones

En primer lugar, analizaremos los algoritmos genéticos individualmente. Comenzando por AGG, vemos que los mejores resultados se obtienen usando el cruce uniforme, aunque el cruce por segmento fijo consigue equipararse a éste en datasets más reducidos (incluso llegando a superarlo en algunas ocasiones). Esto puede deberse a que el operador de cruce por segmento fijo está sesgado: prioriza al primer padre. Debido a esto puede ser que nuestras soluciones no consigan explorar tanto, perdiendo diversidad. Más adelante veremos cómo esta diversidad resulta muy favorable si comparamos AGG con AGE. Independientemente de esto, podríamos haber implementado una versión de cruce por segmento fijo que favorezca al mejor padre, y quizás hubiésemos obtenido resultado más favorables. En AGE la situación es similar: el cruce uniforme también logra mejores soluciones (en todos los parámetros), por las mismas razones.

Comparando AGG con AGE, vemos cómo AGG consigue obtener mejores soluciones, a pesar de que AGE sea más elitista. Esto es algo que me extraña, y que pienso que pueda deberse a dos cosas: por una parte, que la implementación de AGE suponga más evaluaciones de f y por tanto el programa pare antes de que pueda obtener soluciones mejores; por otra, y principalmente, en AGG tenemos más oportunidades de mejora que en AGE, dado que la población se va reemplazando continuamente (y siempre tenemos guardada la mejor solución obtenida, por supuesto). AGE, por su parte, tiene menos oportunidades para mejorar (en AGG generamos más diversidad en una pasada, que puede mejorar la solución con más facilidad). Una observación algo evidente pero interesante es que, dado que el número de evaluaciones es fijo para ambos algoritmos, AGE para en un número de generaciones mucho mayor que AGG (podemos ver de hecho en la gráfica cómo se van creando nuevas generaciones con más frecuencia). En resumen, la variabilidad de AGG es un factor muy favorable que lo posiciona frente a AGE.

Tengamos ahora en cuenta a los algoritmos meméticos. Éstos son notablemente mejores que cualquier otro algoritmo genético, tanto en velocidad de convergencia como en la calidad de las soluciones obtenidas. Un comportamiento esperable si tenemos en cuenta que AM aprovecha la potencia de la búsqueda local y evita caer en óptimos locales gracias a la exploración de los algoritmos genéticos.

Dado que la velocidad de convergencia en AM es alta sería interesante hacer otro criterio de parada (vemos en la gráfica como en el caso de AM-(10,0.1mej) la convergencia se hace poco pasadas las 10000 iteraciones).

Comparando finalmente los tres algoritmos meméticos usados vemos cómo el mejor es el que aplica SLS sobre las mejores soluciones. Esto también tiene sentido, dado que es esperable que sea mejor potenciar las mejores soluciones mientras que mantenemos la diversidad del resto de la población.

En resumen, los algoritmos meméticos nos permiten encontrar un balance entre la exploración y la explotación, no cayendo en óptimos locales (convergencia rápida) o explorando el espacio sin llegar a aprovechar soluciones prometedoras (divergencia). Para estos problemas es, por tanto, mejor utilizar algoritmos meméticos que genéticos. Sin usar los mejores elementos no aprovechamos la potencia de SLS. Adicionalmente, de entre los algoritmos que no usan los mejores elementos vemos cómo, en general, no merece la pena aplicar SLS a toda la población seleccionada (en general, es mejor AM-(10,0.1) que AM-(10,1) en nuestros experimentos). Vemos además cómo al aplicar SLS sobre

toda la población seleccionada (AM-(10,1)) cada generación consume muchas más evaluaciones de la función objetivo (y cómo SLS consume muchas evaluaciones), y es en este algoritmo cuando vemos más claramente el comportamiento de optimizar localmente cada 10 generaciones. También vemos cómo hay zonas de estancamiento, propiciadas por que la SLS no logre mejorar las soluciones (y se pare la búsqueda local). Aquí también puede ser visible la necesidad de implementar otros criterios de parada.

Finalmente, compararemos los algoritmos implementados en esta práctica con los implementados en la práctica anterior (COPKM y BL). Es claro que el greedy es el que actúa peor (y de forma menos "inteligente"), dado que no está buscando optimizar nuestra f – de hecho, prioriza reducir el *infeasibility*, y luego la distancia intra-cluster. En el caso de BL, el algoritmo es capaz de guiarse mejor hacia soluciones que optimicen f. Además, es sorprendente ver cómo BL puede obtener mejores soluciones que algunos genéticos, esto se debe a lo mucho que explotan las soluciones. Pero, de nuevo, no debemos ser optimistas, porque tenemos siempre el peligro de caer en óptimos locales.

Como último comentario, hablemos de los tiempos de ejecución. En general, los algoritmos de esta práctica consumen muchos recursos por evaluar muchas veces f y modificar las soluciones de forma aleatoria, teniendo que recalcular los centroides con mucha frecuencia (si las modificaciones fuesen más sutiles podríamos optimizar estos cálculos). Es por eso que los algoritmos anteriormente implementados tienen tiempos mucho más favorables.