Práctica 3

Búsquedas por Trayectorias 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:

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

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:

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

- 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:

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

[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 cuatro algoritmos adicionales de búsqueda por trayectorias.

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

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

Algoritmos comunes

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).

```
1 es_valida ←contar_por_cluster(solucion).count(0) = 0;
2 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.

Generar vecino: `generar_vecino`

Este es el operador vecino. Dada una solución, calcula un vecino mediante el cambio aleatorio de uno de los datos a un cluster aleatorio. Este operador asegura que la solución obtenida será factible.

Algoritmo (`generar_vecino`). Generar vecino

Input: *solucion* (vector solución válido). **Output:** *vecino* (vector vecino a *solucion* válido).

```
1  indice ←int_aleatorio(0, n);
2  nuevo_cluster ←int_aleatorio(0, k);
3  vecino ←solucion;
4  vecino[indice] ←nuevo_cluster;
5  while not es_solucion(vecino) do
6  |   vecino[indice] ←(vecino[indice] + 1) %k;
7  end
8  return vecino;
```

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 (`reparar_solucion`). Reparar solución

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;
```

También se dispone de una versión para calcular un centroide en concreto, llamada `calcular_centroide`, que tiene además como *input* el identificador del cluster, y que devuelve el centroide asociado a dicho cluster.

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, ..., 0} (k);
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 r0, r1, r2 ←restr[0], restr[1], restr[2];
2 c0, c1 ←solucion[r0], solucion[r1];
3 return c0 ≥ 0 and c1 ≥ 0 and ((r2 = −1 and c0 ≠ c1) or (r2 = −1 and c0 = c1));
```

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;
```

Dado que esta función es la computacionalmente más costosa, en el código se dispone de una implementación que permite pasar los centroides ya calculados.

2. Algoritmos implementados

2.1. 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*). Tendremos en cuenta que *evals* no supere a *max_evals* en la condición de parada de nuestros bucles.

2.2. Enfriamiento Simulado (ES)

En primer lugar, consideraremos el algoritmo de Enfriamiento Simulado. Este algoritmo pretende calentar (hacer que las soluciones convergan a óptimos) y enfriar (desplazarse de sus posiciones iniciales, mediante la inserción de mutaciones) con el objetivo de evitar los óptimos locales, y encontrar mejores soluciones que la Búsqueda Local convencional.

Algoritmo. Enfriamiento Simulado (implementación general)

```
1  T ←T_0; s ←generar_solucion(); s_best ←s;
2  while T > T_f do
3      for cont in {1, ..., L(T)} do
4          s' ←vecino(s);
5          if Δf < 0 or U(0,1) ≤ e^(−Δf/(k·T)) then
6              s ←s';
7              if f(s) < f(s_best) then
8                  s_best ←s;
9              end
10         end
11         T ←g(T);
12     end
13 end
14 return s_best;
```

A continuación, describiremos el algoritmo de ES para nuestro problema, considerando que:

- Usaremos como temperatura inicial:

$$T_0 = \frac{\mu \cdot f(s_{inicial})}{-\ln(\phi)}, \quad \text{con} \quad \mu = \phi = 0.3$$

- Seguiremos el esquema de enfriamiento geométrico, es decir,

$$g(T) = \alpha T, \alpha \in [0.9, 0.99]$$

  Tras varias pruebas, me decanté por $\alpha = 0.95$.

- En cuanto al operador de vecino y la exploración del entorno para $L(T)$, en cada iteración del bucle interno $L(T)$ se aplica un único movimiento para generar una solución vecina (implementado con *generar_vecino*(), ver algoritmos comunes) factible.

- Respecto a la condición de enfriamiento, enfriaremos buen cuando se haya generado un número máximo de vecinos *max_vecinos*, bien cuando se haya aceptado un número máximo de los vecinos generados *max_exitos*.

- El algoritmo finalizará cuando haya alcanzado el número máximo de iteraciones o cuando el número de éxitos en el último enfriamiento sea 0.

Teniendo en cuenta todo esto, podemos describir la versión de ES utilizada:

Algoritmo. Enfriamiento Simulado (implementación para el PAR)

Input: *max_evals* (entero positivo, número máximo de evaluaciones),
max_vecinos (entero positivo, por defecto $10n$),
max_exitos (entero positivo, por defecto $0.1 max\_vecinos$).

Output: s_{best} (mejor solución encontrada), *info* (información de ejecución).

```
1  s ←generar_solucion(); f ←f(s);
2  s_best ←s; f_best ←f;
3  T ←T_0; n_vecinos, n_exitos ←–1, –1;
4  while evals ≤ max_evals and n_exitos ≠ 0 do
5      n_vecinos, n_exitos ←0, 0;
6      while n_vecinos < max_vecinos and n_exitos < max_exitos and evals ≤ max_evals do
7          s_new ←generar_vecino(s); f_new ←f(s);
8          n_vecinos ←n_vecinos + 1;
9          Δf ←f_new − f;
10         if Δf < 0 or U(0,1) ≤ e^(−Δf/T) then
11             n_exitos ←n_exitos + 1;
12             s ←s_new; f ←f_new;
13             if f < f_best then
14                 s_best ←s; f_best ←f;
15             end
16         end
17     end
18     T ←α · T;
19 end
```

Observamos cómo, en concreto en la línea 10, se evidencia la capacidad exploratoria de este algoritmo. Al aceptar soluciones peores con cierta probabilidad en base a la temperatura (más probable conforme mayor temperatura haya), permitimos una mayor exploración al principio, llegando en las últimas ejecuciones (cuando la temperatura ha caído) a explotar las buenas soluciones encontradas.

2.3. Búsqueda Multiarranque Básica (BMB)

El algoritmo de Búsqueda Multiarranque Básica es un algoritmo sencillo, que ejecuta una búsqueda local sobre una solución aleatoria un cierto número de veces, y devuelve la mejor encontrada. El esquema general de BMB es como sigue:

Algoritmo. Búsqueda Multiarranque Básica (implementación general)

```
1  while not condición de terminación do
2      s ←generar_solucion();
3      s' ←busqueda_local(s);
4      actualizar(s_best, s');
5  end
6  return s_best;
```

En nuestro caso, aprovecharemos la clase `BL` ya implementada (práctica 1) que hace una búsqueda local. El comportamiento por defecto del algoritmo es generar una solución inicial e ir iterando sobre ella. Teniendo esto en cuenta podemos pasar a exponer la implementación para nuestro problema:

Algoritmo. Búsqueda Multiarranque Básica (implementación para el PAR)

Input: *max_evals* (entero positivo, número máximo de evaluaciones),
num_ejecuciones (entero positivo, número de ejecuciones de la BL).

Output: s_{best} (mejor solución encontrada), *info* (información de ejecución).

```
1  evals ←0;
2  s_best, f_best, evals_bl ←BL(max_evals=max_evals/num_ejecuciones);
3  evals ←evals + evals_bl;
4  for i in {1, ..., num_ejecuciones − 1} do
5  |  s, f, evals_bl ←BL(max_evals=max_evals/num_ejecuciones);
6  |  evals ←evals + evals_bl;
7  |  if f < f_best then
8  |  |  s_best←s; f_best←f;
9  |  end
10 end
```

Nótese cómo hemos hecho uso del criterio de parada de BL de la práctica 1: dado que queremos realizar 100000 evaluaciones de la función objetivo, y queremos hacer 10 repeticiones de la búsqueda local, le pasamos a cada ejecución de BL un máximo de 10000 evaluaciones. Hemos parametrizado esto con *max_evals* (el número máximo de evaluaciones de la función objetivo de BMB) y con *num_ejecuciones* (el número de ejecuciones de BL), de modo que cada ejecución de la BL tendrá como condición de parada un máximo de *max_evals*/*num_ejecuciones* evaluaciones de la función objetivo.

2.4. Búsqueda Local Reiterada (ILS)

En el caso de la Búsqueda Local Reiterada, generamos una solución aleatoria y aplicamos BL sobre ella. A continuación, estudiamos si esta solución es mejor que la encontrada hasta el momento, realizando una mutación sobre la mejor de ambas y aplicando BL sobre la solución mutada. Continuamos este proceso un cierto número de veces, y devolvemos la mejor solución encontrada.

Algoritmo. Búsqueda Local Reiterada (implementación general)

```
1  s_0 ←generar_solucion();
2  s ←busqueda_local(start=s_0);
3  s_best ←s;
4  while not condición de terminación do
5  |  s' ←mutar(s, historia);
6  |  s'' ←busqueda_local(start=s');
7  |  actualizar(s_best, s'');
8  |  s ←criterio_aceptacion(s, s', historia);
9  end
10 return s_best;
```

Para la implementación de este algoritmo volvemos a reutilizar la clase `BL` de la práctica 1. Sin embargo, añadimos un parámetro adicional, `solucion_inicial`, para poder pasar directamente a BL la solución que queremos explotar. El criterio de aceptación seguido es siempre el mejor, y la mutación es similar a la realizada en prácticas anteriores, y la detallaremos más adelante.

Algoritmo. Búsqueda Local Reiterada (implementación para el PAR)

Input: *max_evals* (entero positivo, número máximo de evaluaciones),
num_ejecuciones (entero positivo, número de ejecuciones de la BL),
v (entero positivo, número de elementos aleatorios en mutación).
Output: s_{best} (mejor solución encontrada), *info* (información de ejecución).

```
1  evals ←0;
2  s_best, f_best, evals_bl ←BL(max_evals=max_evals/num_ejecuciones);
3  evals ←evals + evals_bl;
4  for i in {1, ..., num_ejecuciones − 1} do
5  |  s ←s_best;
6  |  mutar(s, v);
7  |  s, f, evals_bl ←BL(max_evals=max_evals/num_ejecuciones, solucion_inicial=s);
8  |  evals ←evals + evals_bl;
9  |  if f < f_best then
10 |  |  s_best←s; f_best←f;
11 |  end
12 end
```

Haremos un inciso en cómo al ejecutar `BL` sin explicitar una solución inicial, la propia implementación de `BL` genera la solución inicial aleatoria, por lo que he creído conveniente aprovechar esta funcionalidad.

Respecto al operador de mutación, la mecánica de éste es muy sencilla: dado un vector *solucion* y un cierto parámetro v, generaremos una posición aleatoria r_0 de *solucion* y modificaremos las posiciones del vector de r_0 en adelante (ciclando sobre éste) por clusters aleatorios hasta haber modificado v posiciones (llegamos hasta la posición $(r_0 + v)\,\%n$). Si recordamos qué representa el vector solución, vemos que estamos reasignando puntos consecutivos a clusters de forma aleatoria. Finalmente, reparamos la solución para que sea factible (usando el algoritmo descrito unas páginas atrás). El resultado es el siguiente:

Algoritmo (`mutar`). Operador de mutación para ILS

Input: *solucion* (vector solución a mutar, es modificado), v (entero positivo).

```
1  r_0 ←int_aleatorio(0, n);
2  r ←r_0;
3  while v > 0 do
4  |  solucion[r] ←int_aleatorio(0, k);
5  |  v ←v − 1;
6  |  r ←(r + 1) %n;
7  end
8  reparar_solucion(solucion);
```

2.5. Algoritmo híbrido ILS-ES

Este algoritmo es idéntico al anterior, con la diferencia de que en lugar de usar BL para refinar las soluciones iniciales usamos el ILS implementado anteriormente.

La implementación de este algoritmo y del anterior se encuentra en la misma función, permitiendo modificar el algoritmo a usar mediante un parámetro de ésta.

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 consumían 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 ES usando el set `bupa` con 10 % de restricciones:

  ```
  python main.py -N bupa -P 10 -A es -R 4
  ```

- Ejecutar 5 repeticiones de BMB 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 bmb -R 5 -C -CF ./resultados.csv
  ```

- Ejecutar 5 repeticiones de ILS-ES usando el set `zoo` con 10 % de restricciones, cambiando la semilla a 100:

  ```
  python main.py -N zoo -P 10 -A ils -ai es -R 5 -S 100
  ```

Es importante destacar que los algoritmos ILS e ILS-ES están implementados en la misma función, por lo que:

- Para ejecutar **ILS**, use los parámetros `-A ils -ai bl` o directamente `-A ils` (es el por defecto).
- Para ejecutar **ILS-ES**, use los parámetros `-A ils -ai es`.

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	17	0.5936	0.6286	16.2485	554	0.2612	0.5168	28.6681	1062	0.2414	0.4300	71.2345
Ejecución 2	43	0.5242	0.6127	15.8425	532	0.3149	0.5604	28.5388	1044	0.2434	0.4287	70.4300
Ejecución 3	30	0.5609	0.6226	15.5820	513	0.2596	0.4963	28.9172	1054	0.2428	0.4299	71.1775
Ejecución 4	20	0.5259	0.5670	17.4066	478	0.2947	0.5153	29.0655	1016	0.2476	0.4280	71.3281
Ejecución 5	23	0.5928	0.6401	17.1612	557	0.2896	0.5466	28.4932	1042	0.2462	0.4312	70.3587
Media	26.6000	0.5595	0.6142	16.4482	526.8000	0.2840	0.5271	28.7366	1 043.6000	0.2443	0.4296	70.9058
DT	10.3586	0.0341	0.0282	0.8037	32.5991	0.0235	0.0259	0.2469	17.4011	0.0025	0.0012	0.4706

Tabla 1. ES 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	106	0.5011	0.6173	20.0932	1197	0.2941	0.5863	39.8730	2089	0.2421	0.4351	106.5031
Ejecución 2	113	0.5012	0.6251	19.2801	1180	0.3010	0.5891	39.3450	2054	0.2456	0.4354	106.5676
Ejecución 3	57	0.5941	0.6566	18.2770	1146	0.3118	0.5915	39.4749	2029	0.2405	0.4279	108.0249
Ejecución 4	121	0.4993	0.6320	17.8135	869	0.2720	0.4842	40.0867	2027	0.2433	0.4306	108.5506
Ejecución 5	57	0.5869	0.6494	19.3064	1200	0.2982	0.5911	38.7617	2015	0.2451	0.4312	108.2085
Media	90.8000	0.5365	0.6361	18.9540	1 118.4000	0.2954	0.5684	39.5083	2 042.8000	0.2433	0.4321	107.5709
DT	31.3081	0.0494	0.0165	0.9066	141.0613	0.0146	0.0472	0.5130	29.4652	0.0021	0.0032	0.9643

Tabla 2. ES 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	76	0.5227	0.6009	28.5166	118	0.1879	0.2152	155.3637	904	0.1257	0.2059	340.7244
Ejecución 2	50	0.5365	0.5880	25.0643	110	0.1868	0.2122	151.1665	988	0.1229	0.2106	341.7575
Ejecución 3	46	0.5438	0.5911	25.4561	106	0.1893	0.2138	149.3994	870	0.1331	0.2103	340.3467
Ejecución 4	56	0.5137	0.5713	26.3098	106	0.1884	0.2129	145.1130	844	0.1301	0.2051	340.1851
Ejecución 5	42	0.5438	0.5870	27.0868	104	0.1892	0.2132	157.1090	1046	0.1296	0.2224	340.7936
Media	54.0000	0.5321	0.5876	26.4867	108.8000	0.1883	0.2134	151.6303	930.4000	0.1283	0.2109	340.7615
DT	13.3417	0.0134	0.0107	1.3782	5.5857	0.0010	0.0011	4.7860	84.3967	0.0040	0.0069	0.6121

Tabla 3. BMB 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	104	0.5859	0.6429	34.1859	350	0.1870	0.2298	219.0031	1606	0.1250	0.1991	574.9877
Ejecución 2	96	0.5837	0.6363	34.9306	344	0.1850	0.2270	230.7295	1468	0.1337	0.2015	576.5795
Ejecución 3	108	0.5808	0.6400	31.5341	348	0.1869	0.2294	239.4241	1580	0.1285	0.2014	574.9065
Ejecución 4	224	0.5008	0.6236	37.3386	370	0.1812	0.2264	219.4204	1804	0.1232	0.2065	574.3977
Ejecución 5	120	0.5688	0.6346	38.2319	338	0.1889	0.2302	216.7378	1802	0.1246	0.2079	575.0208
Media	130.4000	0.5640	0.6355	35.2442	350.0000	0.1858	0.2285	225.0630	1 652.0000	0.1270	0.2033	575.1784
DT	53.0358	0.0360	0.0074	2.6597	12.0830	0.0029	0.0017	9.6982	147.2753	0.0042	0.0037	0.8228

Tabla 4. BMB 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	45	0.5286	0.5749	20.6874	147	0.1877	0.2216	143.3957	342	0.1192	0.1496	456.2849
Ejecución 2	21	0.5929	0.6145	20.8429	156	0.1865	0.2225	113.6921	411	0.1099	0.1464	458.6775
Ejecución 3	36	0.5929	0.6299	18.1854	150	0.1875	0.2221	119.5338	336	0.1196	0.1495	463.2392
Ejecución 4	30	0.5880	0.6189	19.8879	153	0.1880	0.2233	120.3233	405	0.1170	0.1529	462.9911
Ejecución 5	27	0.5977	0.6254	21.9771	159	0.1858	0.2225	123.4867	387	0.1203	0.1547	468.3521
Media	31.8000	0.5800	0.6127	20.3161	153.0000	0.1871	0.2224	124.0863	376.2000	0.1172	0.1506	461.9090
DT	9.1488	0.0289	0.0220	1.4051	4.7434	0.0009	0.0006	11.3594	35.1525	0.0043	0.0032	4.6510

Tabla 5. ILS 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	105	0.5853	0.6428	29.3917	126	0.2447	0.2600	230.6507	729	0.1189	0.1526	810.2386
Ejecución 2	111	0.5794	0.6402	25.8330	369	0.2001	0.2451	203.2146	633	0.1254	0.1546	804.6430
Ejecución 3	138	0.5823	0.6580	24.5254	390	0.1965	0.2441	188.3520	738	0.1241	0.1582	807.1726
Ejecución 4	207	0.5100	0.6235	34.1069	129	0.2449	0.2606	223.2574	717	0.1250	0.1581	818.2614
Ejecución 5	120	0.5909	0.6567	24.6683	417	0.1969	0.2478	208.5091	843	0.1185	0.1574	815.0446
Media	136.2000	0.5696	0.6442	27.7051	286.2000	0.2166	0.2515	210.7967	732.0000	0.1224	0.1562	811.0721
DT	41.4934	0.0336	0.0141	4.0824	145.8722	0.0258	0.0081	16.7028	74.8198	0.0034	0.0025	5.5835

Tabla 6. ILS 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	339	0.8310	1.0635	83.0511	2073	0.3960	0.7149	215.2371	3159	0.2489	0.4359	465.8137
Ejecución 2	342	0.8624	1.0969	82.0608	2061	0.4066	0.7237	215.3606	3198	0.2478	0.4371	465.6470
Ejecución 3	279	0.9320	1.1234	81.8831	1992	0.4115	0.7179	215.0223	3162	0.2506	0.4378	467.4481
Ejecución 4	369	0.8752	1.1283	101.2422	1944	0.4116	0.7106	216.2835	3102	0.2489	0.4325	469.7113
Ejecución 5	321	0.8859	1.1061	82.2719	2007	0.4117	0.7204	214.8443	3198	0.2476	0.4369	465.7023
Media	330.0000	0.8773	1.1036	86.1018	2 015.4000	0.4075	0.7175	215.3496	3 163.8000	0.2488	0.4360	466.8645
DT	33.2716	0.0369	0.0258	8.4754	52.7096	0.0067	0.0050	0.5584	39.3217	0.0012	0.0021	1.7594

Tabla 7. ILS-ES 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	555	0.8869	1.0898	110.6179	3939	0.4059	0.7265	347.0682	6192	0.2473	0.4380	815.4145
Ejecución 2	729	0.8511	1.1176	110.0882	3753	0.4248	0.7302	345.1915	6123	0.2460	0.4346	815.7767
Ejecución 3	588	0.8620	1.0769	110.0666	3936	0.4130	0.7333	345.1822	6315	0.2433	0.4378	815.2076
Ejecución 4	771	0.8558	1.1376	110.2560	3801	0.4247	0.7340	353.2910	6102	0.2469	0.4348	815.5749
Ejecución 5	720	0.8640	1.1272	109.9928	3939	0.4076	0.7281	345.4474	6168	0.2445	0.4344	815.2582
Media	672.6000	0.8640	1.1098	110.2043	3 873.6000	0.4152	0.7304	347.2361	6,180.0000	0.2456	0.4359	815.4464
DT	94.9963	0.0138	0.0256	0.2505	89.8098	0.0091	0.0032	3.4745	83.4356	0.0017	0.0018	0.2340

Tabla 8. ILS-ES 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
ES	26.6000	0.5595	0.6142	16.4482	526.8000	0.2840	0.5271	28.7366	1 043.6000	0.2443	0.4296	70.9058
BMB	54.0000	0.5321	0.5876	26.4867	108.8000	0.1883	0.2134	151.6303	930.4000	0.1283	0.2109	340.7615
ILS	31.8000	0.5800	0.6127	20.3161	153.0000	0.1871	0.2224	124.0863	376.2000	0.1172	0.1506	461.9090
ILS-ES	330.0000	0.8773	1.1036	86.1018	2 015.4000	0.4075	0.7175	215.3496	3 163.8000	0.2488	0.4360	466.8645

Tabla 9. 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	1.000	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
ES	90.8000	0.5365	0.6361	18.9540	1 118.4000	0.2954	0.5684	39.5083	2 042.8000	0.2433	0.4321	107.5709
BMB	130.4000	0.5640	0.6355	35.2442	350.0000	0.1858	0.2285	225.0630	1 652.0000	0.1270	0.2033	575.1784
ILS	136.2000	0.5696	0.6442	27.7051	286.2000	0.2166	0.2515	210.7967	732.0000	0.1224	0.1562	811.0721
ILS-ES	672.6000	0.8640	1.1098	110.2043	3 873.6000	0.4152	0.7304	347.2361	6,180.0000	0.2456	0.4359	815.4464

Tabla 10. Resultados globales con 20 % de restricciones

	10 % restricciones				20 % restricciones				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	1.000	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
ES	90.8000	0.5365	0.6361	18.9540	1 118.4000	0.2954	0.5684	39.5083	2 042.8000	0.2433	0.4321	107.5709
BMB	130.4000	0.5640	0.6355	35.2442	350.0000	0.1858	0.2285	225.0630	1 652.0000	0.1270	0.2033	575.1784
ILS	136.2000	0.5696	0.6442	27.7051	286.2000	0.2166	0.2515	210.7967	732.0000	0.1224	0.1562	811.0721
ILS-ES	672.6000	0.8640	1.1098	110.2043	3 873.6000	0.4152	0.7304	347.2361	6,180.0000	0.2456	0.4359	815.4464

Tabla 10. Resultados globales con 20 % de restricciones

2. Análisis de resultados

Antes de proceder al análisis de resultados, daremos una justificación de por qué en el caso de este problema he escogido el esquema de temperatura geométrico en lugar del esquema de Cauchy. La razón es sencilla, y es que el método de Cauchy decrementaba la temperatura muy rápido, como podemos ver en la siguiente gráfica, resultado de la ejecución de ES en `glass` con 10 % de restricciones (igualmente ocurre con el resto de sets).

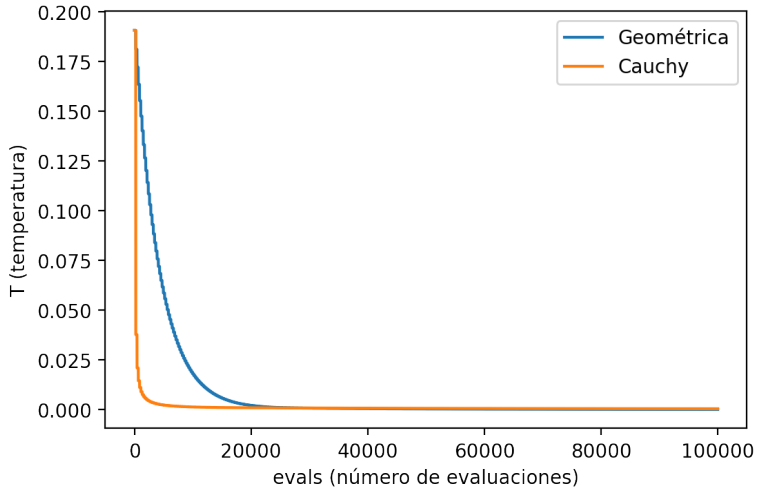

Figura 1: Velocidad de disminución de temperatura

Recordemos que el esquema de Cauchy es aquel que actualiza las temperaturas de acuerdo a la siguiente regla:

$$g(T) = \frac{T}{1 + \beta \cdot T}, \text{ con } \beta = \frac{T_0 - T_f}{M \cdot T_0 \cdot T_f}$$

Para las pruebas estuve probando diversos valores, decantándome por $T_f = 0.001$ y $M = 100000$. Mediante el uso del esquema geométrico obtuve mejores resultados, debido a que al tener un enfriamiento más lento permitimos explorar más el espacio de soluciones. Es por ello por lo que acabé decantándome por el enfriamiento con una razón geométrica.

Procederemos a analizar y comparar los algoritmos empleados. Comenzaremos comparando BL con el resto de algoritmos. Es muy llamativo cómo nuestro algoritmo de BL es capaz de llegar a soluciones muy buenas únicamente explotándolas. ES, sin embargo, a pesar de pretender encontrar un equilibrio entre exploración y explotación permitiendo soluciones peores, no consigue llegar a intensificar estas soluciones lo suficiente como para poder adelantarse a la BL. Más interesante aún es ver cómo el único algoritmo que no obtiene mejores resultados es ILS-ES. Veremos más adelante al comparar todos los algoritmos implementados en esta práctica por qué ILS-ES es (notablemente) inferior al resto.

Del mismo modo, comparando COPKM con los algoritmos implementados (ya que la comparación con BL ya la hicimos en la práctica 1) vemos cómo, de nuevo, el único algoritmo que es peor es ILS-ES.

Compararemos a continuación los algoritmos que hemos implementado entre sí. Lo primero que nos llama la atención es, como comentamos, la inferioridad de ILS-ES. Esto puede deberse a que ILS-ES es un algoritmo que, en un mismo paso, pretende explorar y explotar el espacio de soluciones: procedemos a una mutación (exploración) para luego ejecutar ES, que en su explotación también vuelve a usar exploración. El resultado de este desbalance es que obtenemos soluciones que no están suficientemente optimizadas. Es por ello por lo que el resto de algoritmos desempeña mucho mejor, lo cual es especialmente llamativo en el caso de ILS, pues es el mismo algoritmo empleando BL en lugar de ES. Dado que este último logra tener un mayor balance entre diversificación e intensificación, vemos que los resultados son notablemente mejores.

Por otra parte, vemos cómo BMB da resultados similares a ES, lo cual es bastante llamativo, ya que lo esperable sería que la BMB no consiguiese intensificar las soluciones lo suficiente (en cada iteración de BL dejamos una décima parte de las evaluaciones para optimizar la solución).

Vemos además como ILS es mejor que el resto de algoritmos en los conjuntos de datos más complicados. Esto también nos evidencia cómo ILS consigue un buen balance en exploración y explotación.

Finalmente, respecto a la eficiencia de los algoritmos, vemos cómo BL es notablemente mejor que el resto. Esto se debe, principalmente, a que en esta búsqueda logramos implementar optimizaciones para tener que recalcular los centroides lo mínimo posible. Esto es algo que no podemos conseguir en el resto de algoritmos, dado que las mutaciones introducen elementos impredecibles que nos impiden recalcular los centroides de formas más eficientes.