

Búsqueda local y greedy para el Problema de Agrupamiento con Restricciones



Autor: Andrés Millán

DNI: 77432071H

Email: amilmun@correo.ugr.es

Grupo de prácticas: MH3

Tabla de contenidos

- Sobre esta memoria
 - Benchmark de uno o más algoritmos
 - Ejecutar uno o varios algoritmos para un dataset en particular
- Descripción del problema
- Procedimiento seguido para resolver la práctica
 - Crates usadas
 - Estructura del programa
 - La clase Clusters
 - Sobre las dimensiones
 - Elementos del espacio
 - Representación de las soluciones
 - Restricciones
 - Estadísticos
- Algoritmos considerados
 - Greedy
 - Descripción del algoritmo
 - Búsqueda local
 - Descripción del algoritmo
 - Implementación
- Análisis de resultados
 - Descripción de los casos del problema empleados
 - Benchmarking y resultados obtenidos
 - Greedy
 - Restricciones al 10%
 - Restricciones al 20%
 - Búsqueda local
 - Restricciones al 10%
 - Restricciones al 20%
- Síntesis
- Referencias

Sobre esta memoria

En esta memoria se recoge toda la información necesaria para resolver el problema del **agrupamiento con restricciones**, así como la documentación y el desarrollo de las prácticas de la asignatura Metaheurísticas.

Todo el código está subido en el repositorio de Github <https://github.com/Asmilex/Metaheuristicas>. Para ejecutarlo, es necesario **tener instalado Rust**. El comprimido de la entrega contendrá los mismos archivos que se encuentran en el repositorio. El único cambio será la fecha de la versión y la localización del PDF generado a partir de este archivo.

Se puede compilar y correr el proyecto con `cargo run --release`. Sin embargo, es necesario especificar ciertos parámetros de entrada, que dependerán de lo que se quiera hacer. Estas son las posibilidades:

Benchmark de uno o más algoritmos

Escribir en la línea de comandos `cargo run --release benchmark [algoritmos]`, donde [algoritmos] son uno o más elementos de la siguiente lista:

- greedy.
- bl.

Si no se especifica ninguno, se usarán todos. Cada algoritmo se ejecuta 5 veces por dataset (por lo que cada uno se realiza 30 veces). La información resultante se exportará al archivo `./data/csv/[nombre del algoritmo]_[dataset]_[número de restricciones].csv`, el cual contendrá las medidas necesarias para el posterior análisis que realizaremos.

Ejecutar uno o varios algoritmos para un dataset en particular

Para ejecutar un único algoritmo para un cierto dataset, se debe introducir en la línea de comandos `cargo run --release [dataset] {10, 20} [algoritmos]`, donde [dataset] puede valer bupa, glass o zoo, eligiendo qué conjunto de restricciones usar (10 o 20). La lista de algoritmos funciona de la misma manera que en el apartado anterior.

Descripción del problema

A lo largo de estas prácticas se resolverá el problema del **agrupamiento con restricciones**. Este es una modificación del clásico problema del *clustering*, el cual se describe de la siguiente forma:

Se nos presenta una lista de elementos con un cierto número de atributos. Los representaremos como vectores en $[0, 1]^d$. Debemos agruparlos en un cierto número de categorías, llamados **clústers**, de forma que se minimice la distancia entre estos vectores.

Nuestro matiz consiste en que les pondremos restricciones a los elementos a analizar. De esta forma, forzaremos a que dos vectores deban localizarse en en mismo clúster, o lo contrario; que deban estar en clusters distintos. Por tanto, no solo debemos conseguir una denominada *distancia intraclúster* baja, sino que se debe violar el mínimo número de restricciones posibles.

Durante estas prácticas propondremos diferentes algoritmos para resolver este problema. En la práctica 1, presentaremos soluciones sencillas basadas en algoritmos simples como **Greedy** o **Búsqueda Local**.

Procedimiento seguido para resolver la práctica

Todo el código está escrito en **Rust**. Es un lenguaje moderno, eficiente y fiable en cuestiones relativas a accesos a memoria. Su elección ha sido puro interés personal.

Crates usadas

Para facilitar la implementación, se han utilizado una serie de crates (nombre que reciben las librerías por parte de los rustáceos). Estas son:

- [Naglebra](#): una librería de álgebra lineal. Utilizada para operaciones con matrices y vectores.
- [Rand](#): para los generadores de números aleatorios.
- [Multimap](#): para el almacenamiento eficiente de las listas de restricciones.
- [Csv](#): para exportar los resultados a .csv.
- [Colored](#): para hacer más bonitas y legibles las salidas a consola.

Estructura del programa

Se han dividido las funcionalidades clave del programa en distintos ficheros. Estos son:

- `main.rs`: contiene el código relacionado con la realización de una ejecución simple y de un benchmark, así como el parseo de argumentos.
- `file_io.rs`: aquí se ubican las funciones relacionadas con entrada/salida de archivos. En particular, la lectura de los ficheros de restricciones y la salida a los archivos csv.
- `utils.rs`: se definen las diferentes estructuras relacionadas con el problema. Por ejemplo, `ParametrosDataset` agrupa los datos necesarios para operar un cierto dataset.
- `algoritmo.rs`: todos los algoritmos implementados se encuentran aquí. Ahora mismo, estos son Greedy y Búsqueda Local.
- `clúster.rs`: las principales estructuras necesarias para resolver el problema se localizan en este fichero. Específicamente, la clase `Clusters`. Estudiaremos a fondo sus elementos.

La clase Clusters

Esta estructura supondrá el grueso de nuestro programa. Agrupará toda la información pertinente a la resolución del problema. En las siguientes secciones, describiremos sus elementos. No obstante, omitiremos las funciones de poco interés didáctico

Sobre las dimensiones

Necesitaremos tres medidas para generar una solución:

- `num_clusters` representa el número de clústers fijado por el problema.
- `dim_vectores` es el número de atributos del dataset.
- `num_elementos` es el número de vectores o muestras del dataset.

Elementos del espacio

La clase conoce en todo momento el conjunto de elementos del dataset que estamos tratando, así como sus distancias respectivas. Los miembros que se encargan de guardar esta información son `espacio` y `distancias` respectivamente. El cálculo de la variable λ , de la cual hablaremos más tarde, se guarda cuál es el máximo de las distancias al calcular la matriz `distancias`.

Para almacenar los vectores, hemos utilizado un vector de `Nalgebra::DVector`, un tipo de array dinámico con funciones de álgebra lineal. Esto nos será de gran ayuda, pues simplificará las operaciones del espacio vectorial con el que tratamos.

Representación de las soluciones

Las soluciones se representan con una lista de enteros, `lista_clusters`, de forma que, para una cierta entrada i de dicha lista:

- Si su valor es 0, entonces, ese elemento no tiene clúster asignado
- En otro caso, su valor está en el conjunto $\{1, \dots, num_clusters\}$.

Una solución solo se considerará válida si todo clúster tiene al menos un elemento asignado. Si algún elemento ha modificado su clúster, la clase automáticamente lo registra en la estructura `reuento_clusters`, por lo que nos resultará sencillo comprobar cuántos elementos hay en cada uno.

Restricciones

Representaremos las restricciones de dos formas distintas:

1. La primera de ellas es mediante una matriz dinámica de Nalgebra (`restricciones`) con entradas que toman valores en $0, 1, -1$. Para una cierta entrada $[(i, j)]$, si su valor es 0, no hay ninguna restricción aplicada del vector con posición i y el vector j . Si es 1, entonces es una restricción del tipo Must-Link; esto es, deben ir agrupadas en el mismo clúster. Si su entrada es -1 , ocurre lo contrario al caso anterior: estos dos vectores tienen una restricción del tipo Cannot-Link, y deben ir en clústers distintos. Esta estructura de datos nos resultará útil cuando queramos calcular el infeasibility de todo el sistema.
2. La segunda es un `hashmap` para cada tipo de restricción. Dado un cierto índice i , los hashmaps `restricciones_ML` y `restricciones_CL` devuelven todos los índices con los que tienen restricciones. Aceleran muchísimo el cálculo del infeasibility generado por la asignación de un clúster a un cierto elemento.

El número de restricciones se guarda al crear la matriz de restricciones.

Estadísticos

El interés de este problema reside en ser capaces de crear clústers lo más verosímiles posibles entre sí. Por tanto, para determinar cómo de buena es una solución, necesitamos algún tipo de estadístico que nos informe de ello. Debido a la naturaleza del problema, vamos a considerar dos: El **infeasibility** y el **fitness**.

- **Infeasibility** es una medida de cuántas restricciones han sido violadas en conjunto; es decir, cuántos elementos con restricción Cannot-Link han caído en el mismo clúster, y cuántos vectores con restricción del tipo Must-Link se encuentran en clústers separados. La función `infeasibility()` nos permite conocer esto.
Sin embargo, no siempre nos interesa saber cuál es el estado de todo el sistema, sino cómo de malo sería meter un elemento en un cierto clúster. Para esto sirve la función `infeasibility Esperada(indice, clúster)`. Es una forma mucho más rápida de comprobar incrementos y decrementos en el sistema.
- En este problema, tanto las restricciones incumplidas como la distancia entre los elementos son importantes. Por ello, el **fitness** considera ambas. Éste se define de la siguiente manera:

$$\text{fitness} = \text{desviación general de la partición} + \lambda \cdot \text{infeasibility}$$

donde la desviación general de la partición es la media de la suma de las distancias medias intraclúster, y λ se define como el cociente entre el máximo de las distancias en el sistema y el número de restricciones totales del sistema. Se puede conocer gracias a la función `fitness()`.

Algoritmos considerados

Los algoritmos implementados en la práctica 1 son capaces de generar una solución partiendo de un objeto de la clase Clusters sin asignaciones. Es el único punto en el que podrán encontrarse en este estado.

Dado que todos los métodos de resolución que programemos requieren aleatoriedad, fijaremos unas semillas para todos los generadores del programas. Se encuentran almacenadas en la clase utils.rs/Semillas. Estas son: 328471273, 1821789317287, 128931083781, 1802783721873, 9584985309. El generador usado es el que recomienda la documentación del crate rand: StdRng. El algoritmo que utiliza es el Chacha block cipher de 12 rondas.

Greedy

El algoritmo **Greedy K-medias aplicado a clustering con restricciones** es capaz de proporcionarnos una solución relativamente buena en muy pocos milisegundos. Su implementación es muy sencilla, así como la idea que hay tras éste.

Descripción del algoritmo

Partiendo de un clúster vacío, pero con todos los elementos cargados, consideramos una serie de centroides aleatorios, tantos como número de clústers debamos generar. Recorremos los elementos del espacio de forma aleatoria, de manera que asignamos cada uno al clúster en el que menor número de restricciones se viola (esto es, de menor infeasibility). En caso de empate, se asigna al clúster con cenoide más cercano a nuestro punto, entendiendo por cercano a aquel centroide que minimiza la distancia euclídea. Se actualizan los centroides, y se repite todo hasta que la solución se estabilice.

El pseudocódigo, por tanto, quedaría así:

```
Greedy_COPKM

1. Generar centroides aleatorios con distribución uniforme en R^d.
2. Barajar los indices de forma aleatoria y sin repetición.
3. Mientras se produzcan cambios en el clúster:
    3.1. Para cada índice, mirar qué incremento supone en la infeasibility al asignarlo a un clúster. Tomar el menor de estos.
    3.2. Actualizar los centroides
```

```
Finished release [optimized] target(s) in 0.09s
Running `target\release\metaheuristicas.exe zoo 20 greedy`  

Comienza la lectura de los archivos
    > Se empieza a leer el archivo "./data/PAR/zoo_set.dat"
    > Se empiezan a leer las restricciones "./data/PAR/zoo_set_const_20.const"  

Finalizada la lectura del clúster ✓  

> Ejecutando greedy_COPKM para el cálculo de los clusters
> Cálculo del cluster finalizado en 5 iteraciones ✓  

Greedy calculado en 1.285ms ✓  

Información del cluster:
    > Número de clusters: 7
    > Lista con los clusters: [3, 3, 1, 4, 7, 5, 7, 3, 6, 3, 6, 7, 6, 3, 3, 7, 6, 7, 3, 7, 6, 3, 7, 3, 1, 2, 7, 1, 3, 3, 3, 4, 3, 7, 7, 3, 4, 3, 6, 1, 1, 5, 7, 3, 3, 1, 2, 1, 4, 6, 3, 2, 2, 1, 7, 3, 3, 7, 3, 4, 3, 1, 3, 7, 3, 3, 1, 5, 7, 3, 6, 3, 6, 6, 3, 3, 3, 1, 3, 1, 3, 6, 3, 3, 2, 3, 3, 5, 7, 5]
    > Elementos en cada cluster: [13, 5, 41, 5, 6, 11, 28]
    > Número de restricciones: 912
    > Elementos en el espacio: 101
    > Centroides:  

[  

    0  

    0  

    1  

    0  

    0  

    1  

    0.6923076923076923  

    1  

    1  

    0  

    0.87692307692307693  

    1  

    0  

    1  

    0.87692307692307693  

    0.3076923076923077
]
```

En la sección [Análisis de resultados](#) comprobaremos cómo de buena es la solución obtenida.

Búsqueda local

Búsqueda local es la primera metaheurística que programaremos. Aunque es un algoritmo conceptualmente sencillo, supone una mejora en ciertos aspectos con respecto a Greedy. Se basa en la exploración de vecinos a la solución actual, tomando el mejor de entre los posibles. Por su naturaleza, suele generar óptimos locales.

Descripción del algoritmo

Como hemos citado, se exploran los vecinos de una cierta solución, mirando en cada iteración cuáles son las mejores soluciones. Se define un vecino como la asignación de clúster c al i -ésimo elemento del espacio partiendo de una solución actual. Debemos verificar que la posible solución generada con este operador es válida, pues en otro caso, no tiene sentido seguir.

El concepto de mejor solución es el que proporciona el fitness. Es decir, en cada iteración, se comprobará si el vecino tiene un fitness menor que el actual. Si es así, la siguiente solución a explorar será esta.

Implementación

El pseudocódigo del algoritmo es el siguiente:

```
Generar una solución válida inicial.

Hasta que no se haya alcanzado un óptimo local
    | Guardar la información de la solución actual relevante: fitness, infeasibility,
    | Barajar los índices {0, ..., num_elementos}
    |
    | Para i en los índices barajados
    |     | Barajar los clústers {1, ..., num_clusters}
    |     |
    |     | Para c en los clústers barajados
    |     |     | Si el clúster del i-ésimo elemento no es c
    |     |     |     | Comprobar si el vecino nuevo es válido
    |     |     |     | En ese caso, comprobar si tiene un fitness menor.
    |     |     |         | Si es así, reexplorar todos los índices de nuevo y actualizar la información de la solución nueva.
    |     |     |
    |     |     | Si se ha encontrado una nueva solución, ignorar el resto de índices.
```

Las operaciones más relevantes de este algoritmo las realizamos en la función `bl_fitness_possible_sol(i, c, antiguo_infeasibility)`. Por cómo está gestionado internamente el clúster, es mucho más rápido verificar que la solución es válida dentro del propio clúster, y no fuera. Esta función se encarga de asignar temporalmente el vecino, comprobar si es válido, y en caso de serlo, devolver qué fitness produciría. El motivo de que necesite el antiguo infeasibility es por eficiencia. En vez de calcular todo el sistema para cada vecino nuevo, se calcula de la siguiente manera:

Infeasibility nuevo = (infeasibility antiguo) - (infeasibility producido por el cluster antiguo para el vector i) + (infeasibility producido por el nuevo cluster c para el vector i)

Como mencionamos en uno de los apartados anteriores, el cálculo de esta delta es muchísimo más rápido que el de todo el sistema. En la siguiente sección comprobaremos cuánto tarda en total un bechmark.

```
Finished release [optimized] target(s) in 5.62s
Running "target\release\metaheuristicas.exe zoo 28 bl"
○ Comienza la lectura de los archivos
    • Se empieza a leer el archivo "./data/PAR/zoo_set.dat"
    • Se empiezan a leer las restricciones "./data/PAR/zoo_set_const_28.const"
○ Finalizada la lectura del cluster /

• Ejecutando búsqueda local para el cálculo de los clusters
• Cálculo del cluster finalizado en 200 ms ✓

Información del cluster:
Información del cluster:
    • Número de clusters: 7
    • Lista con los clusters: [5, 5, 6, 1, 4, 7, 4, 5, 1, 5, 7, 4, 7, 5, 3, 4, 7, 4, 5, 4, 1, 5, 4, 5, 6, 1, 4, 6, 5, 5, 5, 5, 1, 5, 4, 4, 5, 4, 5, 2, 5, 7, 6, 6, 1, 4, 5, 5, 6, 4, 5, 1, 5, 6, 4, 7, 5, 1, 1, 6, 4, 5, 5, 4, 5, 2, 5, 6, 5, 5, 4, 5, 5, 1, 4, 5, 7, 5, 7, 7, 5, 5, 5, 5, 5, 6, 5, 5, 7, 5, 5, 5, 1, 5, 5, 1, 4, 1]
    • Elementos en cada cluster: [14, 3, 1, 28, 40, 13, 18]
    • Número de restricciones: 912
    • Elementos en el espacio: 101
    • Centroídes:
```

0
0
0.8571428571428571
0
0
0.8571428571428571
0.5
0.5714285714285714
0.6428571428571428
0.3571428571428571
0
0.375
0.5
0
0.14285714285714285

Análisis de resultados

En esta sección discutiremos los resultados obtenidos por ambos algoritmos. Presentaremos los parámetros de los datasets utilizados, las distancias óptimas conocidas de estas, y cuánto se acercan Greedy y Búsqueda Local a esta.

Descripción de los casos del problema empleados

Los datasets usados reciben el nombre de Zoo, Glass, y Bupa. Los dos primeros presentan una dificultad similar, mientras que el último requiere de un mayor tiempo de cómputo.

	Zoo	Glass	Bupa
Atributos	16	9	5
Clusters	7	7	16
Instancias	101	214	345
Distancia óptima	0.9048	0.36429	0.229248

Sobre estos conjuntos se ha impuesto un número de restricciones: al 10% y al 20%. Naturalmente, cuantas más restricciones, más costoso resulta computar una solución aceptable, pues el sistema contempla una mayor cantidad de enlaces entre sus elementos.

Los ficheros ubicados en `./data/PAR` guardan la información sobre los datasets.

Benchmarking y resultados obtenidos

Cada algoritmo se ha ejecutado 5 veces por dataset. Como tenemos 3 datasets y 2 restricciones para cada uno, nos encontramos un total de 30 ejecuciones por algoritmo. Estudiaremos las siguientes medidas:

- **Tasa de infeasibility:** número de restricciones que se incumplen en la solución.
- **Error de la distancia:** Es el valor absoluto de la diferencia de la desviación intracluster media y la distancia óptima conocida hasta ahora.
- **Agregado:** el fitness de la solución.
- **Tiempo de ejecución** (ms).

Greedy

Restricciones al 10%

	Zoo				Glass				Bupa			
	Tasa_inf	Error_dist	Agr	T (ms)	Tasa_inf	Error_dist	Agr	T (ms)	Tasa_inf	Error_dist	Agr	T (ms)
Ejecución 1	5	0,0799	1,0226	0	6	0,0012	0,3689	2	28	0,0072	0,2294	12
Ejecución 2	0	0,0236	0,9284	2	3	0,0111	0,3784	2	33	0,0076	0,2457	15
Ejecución 3	1	0,0446	0,9570	1	1	0,0163	0,3816	2	33	0,0042	0,2422	14
Ejecución 4	0	0,0081	0,8966	1	3	0,0125	0,3798	3	12	0,0092	0,2231	10
Ejecución 5	0	0,0570	0,9618	1	7	0,0331	0,4043	2	37	0,0025	0,2366	14
Media	1,2	0,0426	0,9533	1	4	0,0171	0,3866	2	28,6	0,0062	0,2354	13

Restricciones al 20%

	Zoo				Glass				Bupa			
	Tasa_inf	Error_dist	Agr	T (ms)	Tasa_inf	Error_dist	Agr	T (ms)	Tasa_inf	Error_dist	Agr	T (ms)
Ejecución 1	2	0,0516	0,9645	1	0	0,0086	0,3556	1	12	0,0129	0,2439	5
Ejecución 2	1	0,0738	0,9827	0	1	0,0103	0,3544	1	7	0,0170	0,2131	5
Ejecución 3	3	0,0813	0,9982	1	0	0,0517	0,3125	1	8	0,0111	0,2415	6
Ejecución 4	3	0,0927	1,0096	1	0	0,0086	0,3556	1	19	0,0091	0,2410	5
Ejecución 5	1	0,1119	1,0208	0	0	0,0086	0,3556	4	0	0,0024	0,2317	7
Media	2	0,0823	0,9951	0,6	0,2	0,0176	0,3467	1,6	9,2	0,0105	0,2342	5,6

Búsqueda local

Restricciones al 10%

	Zoo				Glass				Bupa			
	Tasa_inf	Error_dist	Agr	T (ms)	Tasa_inf	Error_dist	Agr	T (ms)	Tasa_inf	Error_dist	Agr	T (ms)
Ejecución 1	11	0,2940	0,6940	126	51	0,1760	0,2384	618	116	0,1196	0,1407	10738
Ejecución 2	11	0,2918	0,6963	161	50	0,1696	0,2438	460	118	0,1217	0,1391	5789
Ejecución 3	12	0,2385	0,7571	120	60	0,1791	0,2442	641	142	0,1057	0,1615	4228
Ejecución 4	27	0,1631	0,9462	186	50	0,1738	0,2396	453	115	0,1202	0,1398	6993
Ejecución 5	11	0,2992	0,6888	112	54	0,1694	0,2479	545	95	0,1183	0,1363	6717
Media	14,4	0,2573	0,7565	141	53	0,1736	0,2428	543,4	117,2	0,1171	0,1435	6893

Restricciones al 20%

	Zoo				Glass				Bupa			
	Tasa_inf	Error_dist	Agr	T (ms)	Tasa_inf	Error_dist	Agr	T (ms)	Tasa_inf	Error_dist	Agr	T (ms)
Ejecución 1	18	0,1728	0,8045	117	37	0,1141	0,2694	486	264	0,1199	0,1461	810!
Ejecución 2	18	0,2193	0,7581	160	30	0,1134	0,2664	468	209	0,1175	0,1408	703!
Ejecución 3	18	0,1532	0,8242	71	38	0,1185	0,2655	628	224	0,1151	0,1453	455!
Ejecución 4	10	0,1711	0,7740	105	36	0,1142	0,2687	575	162	0,1088	0,1429	584!
Ejecución 5	15	0,1743	0,7909	161	49	0,1145	0,2752	486	256	0,1210	0,1439	645!
Media	15,8	0,1782	0,7904	122,8	38	0,1149	0,2691	528,6	223	0,1165	0,1438	6400,:

Síntesis

Sinteticemos los resultados anteriores en una sola tabla, y analicemos cuáles son las consecuencias de las ideas tras los algoritmos:

	Zoo				Glass				Bupa			
	Tasa_inf	Error_dist	Agr	T (ms)	Tasa_inf	Error_dist	Agr	T (ms)	Tasa_inf	Error_dist	Agr	T (ms)
COPKM 10%	1, 2	0, 0426	0, 9533	1	4	0, 0171	0, 3866	2	28, 6	0, 0062	0, 2354	13
COPKM 20%	2	0, 0823	0, 9951	0, 6	0, 2	0, 0176	0, 3467	1, 6	9, 2	0, 0105	0, 2342	5, 6
BL 10%	14, 4	0, 2573	0, 7565	141	53	0, 1736	0, 2428	543, 4	117, 2	0, 1171	0, 1435	6893
BL 20%	15, 8	0, 1782	0, 7904	122, 8	38	0, 1149	0, 2691	528, 6	223	0, 1165	0, 1438	6400, 2

Debemos abordar la discusión desde varios puntos de vista.

En primer lugar, hablemos sobre el infeasibility y la agregación obtenida. En todos los conjuntos de datos, Greedy consigue un menor número de restricciones incumplidas. En Zoo, la diferencia es de 12 unidades aproximadamente, pero en Bupa incrementa hasta los 90. A priori, podríamos decir que Greedy es objetivamente mejor. Sin embargo, debemos fijarnos en la agregación: en contrapartida, ésta es siempre menor en la Búsqueda Local. Esto se debe a que BL intenta minimizar el fitness, y no el infeasibility. Aún así, el error de la distancia cometido por Greedy es significativamente menor al de Búsqueda Local.

Atendiendo a los tiempos de ejecución, podemos observar que Greedy consigue una solución muy buena en cuestión de pocos milisegundos, mientras que Búsqueda Local requiere hasta 7s como ocurre en el dataset Bupa.

Por tanto, debemos preguntarnos: ¿Qué debemos priorizar? Si lo que necesitamos es una baja agregación, debemos optar por Greedy. En cualquier otro caso, nuestra versión de Greedy es mejor.

Personalmente, la diferencia de 0.2 en la agregación no consigue justificar lo cohesionados que están los clústers de Greedy y la poca tasa de infeasibility que producen. Además, extrapolando la información que tenemos hasta ahora, si aplicáramos estos dos algoritmos que tenemos hasta ahora, Búsqueda Local necesitaría mucho más tiempo de cómputo para calcular una solución similar o incluso peor a Greedy. Por estos motivos, no podría justificar su uso.

Finalmente, es necesario mencionar que en las siguientes prácticas implementaremos metaheurísticas mucho más avanzadas que Búsqueda Local, como los algoritmos genéticos. Espero ver superados los resultados de Greedy rápidamente, tanto en infeasibility como en agregación y en desviación con respecto a la distancia óptima. Será interesante comprobar cuántos segundos tardan en ejecutarse, ya que complicará la elección de un algoritmo sobre otro.

Referencias

- [El libro oficial de Rust](#)
- [Imagen de la portada](#)
- [Documentación de Nalgebra](#)
- [Documentación de Multimap](#)
- [StackOverflow](#)
- Material de teoría y de prácticas