UNIVERSIDAD
DE GRANADA

METAHEURÍSTICA

Práctica 1.
Problema del Agrupamiento con Restricciones

Lukas Häring García

1. Introducción.

Existen problemas que actualmente no se conoce ninguna estrategia para la búsqueda de una solución óptima en tiempo polinómico, es por ello que autores han ideado nuevas estrategias, cada vez más abstracta, para encontrar o aproximar la solución óptima en tiempo polinómico.

El problema propuesto por la asignatura es conocido como “Problema del Agrupamiento”, este problema se puede resolver en un tiempo $O(n^{dk+1} log\ n)$ [1], tal que son fijados “k”, el número de clusters, y “d”, la dimensión. “n” el número de entidades. Este tiempo está categorizado como polinómicos, es por ello que ha sido modificado para éste convertirse en un problema duro.

Ahora, el problema presenta dos tipos de restricciones, deben unirse **(*“Must-Link”*)** y no pueden unirse **(“*Not-Link*”)**. Utilizando los parámetros anteriores sobre unos *datasets*, vamos a proponer diferentes métodos para la aproximación de la solución óptima: La primera, la estrategia *codiciosa* **(Greedy)**, esta será el pilar para poder comparar los demás métodos; La segunda, la búsqueda local, en vez de explorar todo el conjunto de soluciones (que no es posible en tiempo polinómico), vamos a explorar un subconjunto de elementos e intentar converger a una solución aceptable; Otros métodos más ambiciosos inspirados en la naturaleza...

El propósito de esta práctica es la experimentación con algoritmos no rudimentarios para obtener una aproximación más o menos óptima en un tiempo aceptable, además de importar algoritmos bioinspirados para resolver algoritmos complejos.

2. Descripción de los algoritmos.

Para la resolución del problema, debemos optimizar una función de búsqueda, en nuestro caso debemos encontrar la disposición de k clusters o grupos tales que representen al máximo el conjunto de datos, es decir, que violen al mínimo las restricciones impuestas en el problema, por ello, la función a optimizar debe contener la cantidad de restricciones violadas.

El lenguaje utilizado es Python, un lenguaje utilizado para la investigación gracias a su facilidad para escribir algoritmos utilizando estructuras in-line. Además, se ha complementado utilizando la librería matemáticas, como son **numpy**, esta permite realizar cálculos matriciales de manera paralela y **spatial** que ayudará en cálculos con puntos.

Importamos los datasets y las restricciones utilizando el método de **numpy** *loadtxt*, que acepta como parámetros, la ruta del archivo, el tipo de archivo y el delimitador. Estos son cargados de forma individual en un método llamado *main*, que requiere el nombre del archivo, el porcentaje de restricciones y el número de clusters. Este importará los archivos necesarios y llama a las diferentes algoritmos expuesto más abajo, con sus parámetros requeridos.

Para calcular el total de restricciones violadas, se ha propuesto una estrategia eficiente (Véase “2.4 Optimizaciones”) pero no trivial, explotando la potencia de **numpy**, que de otra forma se habría implementado con un único núcleo.

Otro elemento común es la comprobación de que los clusters nunca están vacíos, es por ello que se ha creado un método *has_one* al que se le pasa el vector de asignación de la n-ésima entidad al k-ésimo cluster y devuelve Positivo si no está ninguno vacío.

El número de restricciones es calculado utilizando la suma de cada elemento de la matriz de restricciones en valor absoluto, para así poder sumar tanto las restricciones “**Must-Link**” como las “**Not-Link**”, restar la diagonal principal y quitar las restricciones simétricas, dividiendo entre dos. Este cálculo será útil para el cálculo de la **agregación** en el código el método es llamado “f(...)”, de manera más concreta es el factor multiplicativo de las restricciones violadas, que se obtiene de dividir la distancia de los elementos más lejanos entre la cantidad de restricciones, la distancia de estos dos elementos es calculada utilizando la matriz de distancia en

el método **lambda_factor**, Una vez obtenida la matriz de distancia que es obtenida de forma rápida con la librería **spatial** utilizando el método *distance_matrix*. finalmente buscamos los índices de estos dos elementos más lejanos y calculamos la distancia entre ellos según la métrica propuesta en la práticas L_2 euclídea.
La función "f(...)" también hace uso de la desviación general intracluster, esta está implementada en el método *general_deviation* este método coge el el vector de asignación y crea una matriz dentada de altura k, que contiene los pares de puntos para los k-ésimos puntos d dimensionales. De cada fila de la matriz dentada calculará la **distancia intracluster** y realizará la media aritmética. La distancia intra-cluster es calculada haciendo uso de la función *distance_intracluster* como tenemos restringido que los clusters siempre tienen que tener al menos un elemento, realizamos la media aritmética de la distancia euclídea sobre los elementos y su centroide, que es calculado con el método *group_centroid* que requiere la lista de elementos en el cluster, este es calculado como la media aritmética de la posición de cada uno de los puntos. Todos estos cálculos son realizados con funciones explícitas de **numpy** realizadas en paralelo dándonos tiempos de ejecuciones mínimos, como por ejemplo la sumatoria es realizado con el método "*sum*" o la distancia euclídea, como hemos visto antes, he intentado extrapolar el problema a un problema matricial y así poder utilizar al máximo **numpy**.

Para calcular la **región de vecinos del algoritmo** local (*generate_neighbourhood)*, como se ha indicado solo se realizará una única transformación, es por ello que vamos a tomar todas las posibles mutaciones unitarias sobre la lista de partida, como tenemos n entidades y k cluster con inicialmente un i-ésimo cluster por entidad entre el intervalo 0 … k - 1, por lo que necesitaremos como máximo $n(k-1)$ mutación válida, es decir, sin dejar ningún cluster vacío. Para **optimizar esta generación** y evitar "braching", es decir tener un if que compruebe que ese i-ésimo elemento coincide con la mutación, se realizará un bucle que vaya de forma ordenada entidad a entidad e introducirá pares válidos (posición, cluster), de tal forma en que se devolverán los pares (n, 0), ... , (n, j-1) donde j es el cluster ya asignado y luego (n, j+1), …, (n, k - 1), es decir un bucle con dos bucles anidados, cada uno de ellos comprobando que la transformación no deja ningún cluster vacío, finalmente, se utiliza la librería **numpy** para realizar permutaciones aleatorias de dichos pares, utilizando *random.permutation*.

Las semillas utilizadas para cada iteración se muestran a continuación: [10,11, 12, 13,14], estas semillas son utilizadas en el mismo orden en el que aparecen, son ejecutadas para cada ejecución de los algoritmos, así nos aseguramos que en

diferentes ejecuciones, los valores de retorno sean siempre los mismos. Cómo se utilizan diferentes formas utilidades de los generadores aleatorios de distintas librerías, debemos asignarlas a la librería **random** y **numpy** utilizando *random.seed(...)* y *np.random.seed(...)*.

El método *main* llama a los tres algoritmos de forma secuencial, para ello hace uso del método *algorithm* que calcula los datos que piden para la práctica (la media, el agregado, la desviación, etc.), además, será el encargado de dividir la matriz de restricciones (Véase "2.4 Optimizaciones"). Este hace uso de un puntero al algoritmo, las entidades, las restricciones y el número de agrupaciones.

Para ejecutar el código, debemos tener en nuestra carpeta, otra con el nombre de datasets y en ella, todos los archivos ".dat" y ".const", pasándole como parámetro a nuestra función, el nombre del dataset sin la extensión "_set...".

También se ha implementado un método para visualizar los puntos en un espacio bidimensional, este método recibe el nombre de *plot2d,* haciendo uso de la librería **matplotlib** y recibiendo como parámetros: las entidades, el vector de asignación, los centroides y la matriz negativa/positiva, se pasan dichos puntos uno a uno a la función *scatter* y nos dibuja dichos puntos con los colores pre-asignados.

2.1. Algoritmo Greedy (COPKM).

El algoritmo Codicioso (Greedy) propuesto inicialmente, no ofrecía solución en muchos de los datasets ofrecidos, es por eso que se modificó para que este tuviera siempre solución aunque no fuera la mejor.

Inicialmente generamos un vector $RSI =$ [0...N], siendo N la cantidad de entidades y se realiza unas permutaciones aleatorias para no acceder de manera ordenada, haciendo que el problema no generalice en distintas situaciones iniciales. Además, vamos a necesitar una configuración inicial de los centroides, para ello, se ha implementado la función *random_centroids* que recibe como parámetro la lista de entidades y el número de clusters, para ello, buscamos el valor máximo en cada dirección dimensional, utilizando la función ***amin*** $M^{in}{}_i$ y ***amax*** $M^{ax}{}_i$ de numpy, creamos una matriz diagonal con la diferencia de estos dos valores, dándonos así una matriz de escalado S_{NxN} de dichos intervalos ($\Delta_i = M^{ax}{}_i - M^{in}{}_i$), ahora creamos una matriz R_{NxK} que va a representar a K centroides aleatorios en el intervalo [0..1], Transformamos dicho intervalo multiplicando ambas matrices $(S_{NxN} \cdot R_{NxK})^t$, estos valores están entre el intervalo [0...max(Δ_i)]. Falta trasladar el intervalo para que

así, estos estén generados en el hipercubo que encierra a las entidades, $M^{in}{}_i + (S_{NxN} \cdot R_{NxK})^t$, haciendo que el intervalo se traslade hacia para cada componente, tal que [$M^{in}{}_i$ ...max($M^{in}{}_i + \Delta_i$)].

Una vez generado los centroides, vamos a generar un vector de asignación inicial, lleno de -1, que va a mantener el mejor vector encontrado hasta el momento. Nuestro algoritmo va a comprobar para cada vector de asignación generado, si este es distinto al antiguo, si es así, entonces lo reemplazará, en caso contrario, el algoritmo terminará (convergerá). Para cada iteración del algoritmo, se creará un nuevo vector de asignación y se va a recorrer de forma ordenada, el vector de índices anteriormente desordenado RSI, para cada entidad, se va a calcular la distancia a sus centroides y su *infeasability* con la asignación inicial al cluster 0, suponiendo que es el mejor encontrado hasta ahora. Ahora para los demás cluster, se va a repetir el proceso anterior, pero si encontramos un cluster cuyo *infeasability* y la distancia a la entidad es menor al inicial, reemplazamos, así con cada uno de los posibles clusters. Finalmente, para cada elemento de RSI, vamos asociando al i-ésimo elemento, el k-ésimo mejor cluster encontrado.

Una vez realizado para todos los elementos de RSI, vamos a recalcular los centroides obtenidos utilizando la función previamente descrita *group_centroid*, así, el algoritmo para cada iteración, va a agrupar converger en una posición donde los centroides representan mejor a su población, mientras el vector asignado en dicha iteración sea distinto al asignado previamente, entonces, el algoritmo, continuará.

2.2. Algoritmo de Búsqueda Local, Primer Mejor (BL-PM).

El algoritmo de búsqueda local por primer mejor ("*BL_PM*"), debe devolver una solución válida, es decir, ningún cluster debe quedar vacío. Es por ello que inicialmente se va a generar un vecino central que cumpla con dicha restricción, por lo que se va a generar dicho vecino de forma aleatoria, hasta encontrar uno que la cumpla. Una vez obtenido, vamos a calcular su agregado utilizando "*f(...)*" y vamos a mantener un contador del total de ejecuciones de dicha función como una bandera que nos indique si se encontró una solución en dicho vecindario.

Este algoritmo intenta encontrar una solución rápida, viajando a través del vecindario, una vez encontrado un vecino cuyo agregado sea mejor que este, se sustituirá dicho vecino por el mejor y se repetirá el proceso.

Mientras el número de ejecuciones de “f(...)” sea menor a 100.000 iteraciones y no se haya encontrado un vecino mejor al mejor (en este caso, el inicial), vamos a generar el vecindario de este y poner la bandera en falso, para ello vamos a utilizar el método *generate_neighbourhood* explicado anteriormente, utilizaremos la lista de pares obtenidos para optimizar (Expuesto en la transparencia del Seminario de prácticas), en vez de generar un vector de asignación, vamos a modificar el vecino actual según el par obtenido y recalcular “f(...)” incrementando el contador y comprobamos que no hemos superado la tasa límite, si el obtenido es menor al vecino del que veníamos, entonces sustituimos el mejor “f(...)”, mantenemos dicho cambio y activamos la bandera, finalmente rompemos el bucle y pasamos a la siguiente iteración.

Si en caso contrario, no se encontró un mejor vecino o se sobrepasó el número máximo de iteraciones, entonces devolvemos la solución.

2.3. Algoritmo de Búsqueda Local, Mejor Vecino (BL-MV).

Este algoritmo, que en mi código recibe el nombre de “BL_PM”, es variación del anterior, en este caso, el algoritmo es computacionalmente más costoso ya que en vez de quedarse con el mejor vecino iterado hasta el momento, va a mirar todos los vecinos generados. La diferencia en código es simple, en vez de romper el código, vamos a mantener el mejor vecino en una variable, una vez acabado el bucle, vamos a sustituirlo si se ha encontrado un mejor vecino, si no existe, procedemos a terminar el algoritmo y devolver la solución.

Remarcar también que este algoritmo tiene las mismas restricciones que el anterior, un total de 100.000 ejecuciones de la función y los vecinos generados no pueden dejar ningún cluster vacío.

2.4. Optimizaciones.

Tras un análisis del código, el 62% del tiempo de ejecución estaba siendo dedicado a la función *infesibility*. Este código estaba realizado con un doble bucle anidado comprobando la submatriz superior, lo que no aprovechaba al máximo la capacidad de paralelizar este método, al ser trabajos separables. Para ello, en vez de recurrir a gestionar hebras, aposté por encontrar una técnica basada en matrices, aprovechando que **numpy** tiene implementado todos los cálculos matriciales eficientemente. Observé que si tuviéramos dos matrices tales que las restricciones fueran positivas, entonces se cumple que:

	k_1	k_2	...	k_n
k'_1	0	$r_{1,1}$	….	$r_{n,1}$
...	...	...	...	...
k'_n	$r_{n,1}$	$r_{n,2}$	....	0

Donde $r_{ij} \in \{0,\ 1\}$ y $k_m, k'_m \in \{0,\ ...,\ k-1\}$, es decir uno es el vector de asignación y la otra, la matriz de restricciones $R_{n,n}$.

Se incumplía las restricciones (o se cumplían, según la matriz) si $r_{ij} k_i k'_j = k^2{}_i \neq 0$. Podría transformar $k'_j = \frac{1}{k_j}$, así $r_{ij} k_i k'_j = 1 \leftrightarrow r_{ij} \neq 0$ y aquellos que sean exactamente 1, incumplirían o no restricción. Hay de remarcar que tenemos que desplazar el interval $k_m, k'_m \in \{1,\ ...,\ k\}$ sumando 1 para no tener divisiones entre 0. Si el vector contiene -1 como en el caso de greedy, cuando obtengamos *infinity*, sustituimos por 0.

El método *infeasibility* además de tener el vector de entidades y de asignación tiene 2 matrices obtenidas a partir de la matriz de restricción (en la función *algoritm*), eliminando la diagonal principal. Estas matrices son *positive_restrictions* (P_r) que mantiene las celdas de la matriz de restricciones en 1 si eran 1 y 0 si eran distinto de 1, como sabemos que las celdas están formadas por 0, -1 y 1, basta con, a cada celda aplicar $c'_{i,j} = c_{i,j}(c_{i,j} + 1)\frac{1}{2}$ y *negative_restrictions* (N_r) que convierte a 1 aquellas celdas con -1 y 0 las demás, de manera similar, $c'_{i,j} = c_{i,j}(c_{i,j} - 1)\frac{1}{2}$, este cálculo se hace para evitar “branching”.

Para realizar la multiplicación escalar de $r_{ij} k_i k'_j$, basta con poner k_i, k'_j en la diagonal principal de una matriz NxN, denotamos para k_i, $M_{n,n}$ y para k'_j $M'_{n,n}$:

$$P(X) = M_{n,n}(M'_{n,n}X)^t,\ infeasibility = \sum_{c \in P(P_r) \mid c \neq 1,0} 1 + \sum_{c \in P(N_r) \mid c = 1} 1$$

3. Análisis de los resultados

3.1 Dataset “Rand”.

El factor obtenido es $\lambda = 0.0072$, está definido el problema para 3 clusters.

3.1.1. Análisis Tabla de Resultados Greedy.

10% COPKM	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	1.00	280.00	3.01	1.15s
Ejecución 2	1.00	226.00	2.63	0.77s
Ejecución 3	0.92	158.00	2.05	1.12s
Ejecución 4	1.00	200.00	2.44	1.53s
Ejecución 5	1.21	238.00	2.92	1.91s
Media	1.02	220.40	2.61	1.30s

20% COPKM	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.76	68.00	1.01	1.16s
Ejecución 2	1.02	408.00	2.49	1.17s
Ejecución 3	0.86	302.00	1.95	1.59s
Ejecución 4	0.77	156.00	1.33	1.52s
Ejecución 5	0.85	256.00	1.77	1.57s
Media	0.85	238.00	1.71	1.40s

Como podemos ver, la tasa de infeasibility es bastante alta, lo que nos dice que greedy nos ofrece una solución bastante mala. Aún así, el agregado es bastante bajo debido a que el factor tiende a darle poca importancia a la infeasibility, debido a que los datos están poco dispersos.

3.1.2. Análisis Tabla de Resultados Búsqueda Local, Primer Mejor.

10% BL-PM	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	3.02s (1522 its)
Ejecución 2	0.72	0	0.72	3.28s (1671 its)
Ejecución 3	0.72	0	0.72	3.17s (1613 its)
Ejecución 4	0.72	0	0.72	2.63s (1353 its)
Ejecución 5	0.72	0	0.72	2.90s (1484 its)
Media	0.72	0	0.72	3.00s (1529 its)

20% BL-PM	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	3.62s (1818 its)
Ejecución 2	0.72	0	0.72	3.25s (1618 its)
Ejecución 3	0.72	0	0.72	3.25s (1640 its)
Ejecución 4	0.72	0	0.72	3.03s (1575 its)
Ejecución 5	0.72	0	0.72	2.27s (1123 its)
Media	0.72	0	0.72	3.08s (1555 its)

Vemos que el tiempo de ejecución es superior a greedy, pero la solución parece ser la óptima, debido a que no se han inclumplido ninguna restricción, haciendo que el agregado coincida con la distancia media intra-cluster. Cabe destacar que el algoritmo ha necesitado para los distintos números de restricciones, aproximadamente ~3s, lo que hace que en un problema duro, sea una solución muy buena en un tiempo rápido. Con respecto a Greedy, esta técnica tarda un 60% más, pero ofrece la solución óptima.

3.1.3. Análisis Tabla de Resultados Búsqueda Local, Mejor Vecino.

10% BL-MV	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	51.17s (30900 its)
Ejecución 2	0.72	0	0.72	57.59s (35100 its)
Ejecución 3	0.72	0	0.72	46.18s (28200 its)
Ejecución 4	0.72	0	0.72	44.76s (27300 its)
Ejecución 5	0.72	0	0.72	42.79s (26100 its)
Media	0.72	0	0.72	48.50s (29520 its)

20% BL-MV	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.72	0	0.72	51.08s (30600 its)
Ejecución 2	0.72	0	0.72	53.82s (32100 its)
Ejecución 3	0.72	0	0.72	46.61s (27900 its)
Ejecución 4	0.72	0	0.72	44.65s (26700 its)
Ejecución 5	0.72	0	0.72	42.35s (25200 its)
Media	0.72	0	0.72	47.70s (28500 its)

Este problema computacionalmente es mucho más costoso, como habíamos comentado anteriormente, explora más posibles soluciones, pero es 15 veces más lenta sobre la técnica anterior, además, de ofrecer la misma solución, añadiendo cómputo innecesario. Vemos que el número medio de iteraciones necesarias es innecesario.

3.1.4. Análisis Tabla de Resultados Medios.

10%	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
COPKM	1.02	220.40	2.61	1.30s
BL-PM	0.72	0	0.72	3.00s (1529 its)
L-MV	0.72	0	0.72	48.50s (29520 its)

20%	**RAND (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
COPKM	0.85	238.00	1.71	1.40s
BL-PM	0.72	0	0.72	3.08s (1555 its)
BL-MV	0.72	0	0.72	47.70s (28500 its)

Como podemos observar, Greedy tiene una tasa-intracluster baja, al igual que las búsquedas locales, este ofrece una solución rápida aunque no muy positiva.

Las búsquedas locales en este dataset se caracterizan por ofrecer ambos la solución óptima[2], pero gana con creces **primer mejor** ya que aunque explora mucho menos, lo que resulta un ahorro en tiempo de ejecución además de la cantidad de iteraciones necesarias.

Vemos que este problema realmente es muy sencillo, tiene una dimensionalidad de 2, con 3 clusters y el que menos puntos a categorizar, pero aún así, vemos que explorar el entorno por completo, es muy costoso.

3.2 Dataset “Iris”.

El factor obtenido es $\lambda = 0.0063$, también definido para 3 clusters.

3.2.1. Análisis Tabla de Resultados Greedy.

10% COPKM	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	1.16	522.00	4.47	2.26s
Ejecución 2	1.10	690.00	5.47	1.87s
Ejecución 3	0.78	454.00	3.66	1.13s
Ejecución 4	0.91	414.00	3.54	1.50s
Ejecución 5	0.94	384.00	3.38	1.89s
Media	0.98	492.8	4.10	1.73s

20% COPKM	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	1.13	1044.00	4.44	1.59s
Ejecución 2	0.93	866.00	3.68	2.35s
Ejecución 3	0.70	632.00	2.70	1.20s
Ejecución 4	0.85	598.00	2.74	2.41s
Ejecución 5	0.88	498.00	2.46	2.14s
Media	0.90	727,6	3.20	1.94s

El problema sigue careciendo de una solución óptima, vemos que el factor además, da aún menos importancia al *infeasibility*, además, podemos observar que al pasar de 10% a 20% de restricciones, casi hemos duplicado en algunos casos la cantidad de restricciones violadas.

Aunque el tiempo de cómputo requerido, es bastante bajo, tenemos una proporción muy elevada de restricciones violadas.

3.2.2. Análisis Tabla de Resultados Búsqueda Local, Primer Mejor.

10% BL-PM	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	2.69s (1344 its)
Ejecución 2	0.67	0.00	0.67	2.86s (1440 its)
Ejecución 3	0.67	0.00	0.67	2.93s (1450 its)
Ejecución 4	0.67	0.00	0.67	3.29s (1742 its)
Ejecución 5	0.67	0.00	0.67	3.42s (1804 its)
Media	0.67	0.00	0.67	3.04s (1556 its)

20% BL-PM	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	3.63s (1781 its)
Ejecución 2	0.67	0.00	0.67	3.51s (1734 its)
Ejecución 3	0.67	0.00	0.67	3.00s (1486 its)
Ejecución 4	0.67	0.00	0.67	3.46s (1704 its)
Ejecución 5	0.67	0.00	0.67	3.55s (1766 its)
Media	0.67	0.00	0.67	3.63s (1781 its)

De manera similar al problema anterior, de nuevo, nos ofrece una solución al parecer óptima, esta solución tiene una baja desviación media intracluster, con ninguna restricción violada, además el tiempo de ejecución es bastante rápido, aunque de la misma forma, duplica el tiempo que a greedy.

Ofrece una solución buena en un tiempo razonable, vemos que a mayor número de restricciones, el tiempo de ejecución aumenta levemente, pero aún así, podríamos considerarlo un buen resultado.

3.2.3. Análisis Tabla de Resultados Búsqueda Local, Mejor Vecino.

10% BL-MV	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	62.02s (37200 its)
Ejecución 2	0.67	0.00	0.67	53.33s (32100 its)
Ejecución 3	0.67	0.00	0.67	51.58s (30900 its)
Ejecución 4	0.67	0.00	0.67	45.38s (27300 its)
Ejecución 5	0.67	0.00	0.67	42.88s (25800 its)
Media	0.67	0.00	0.67	51.04s (30660 its)

20% BL-MV	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	0.67	0.00	0.67	52.45s (30000 its)
Ejecución 2	0.67	0.00	0.67	49.78s (28800 its)
Ejecución 3	0.67	0.00	0.67	47.08s (27300 its)
Ejecución 4	0.67	0.00	0.67	45.13s (26100 its)
Ejecución 5	0.67	0.00	0.67	44.68s (25500 its)
Media	0.67	0.00	0.67	47.82s (27540 its)

Vemos que obtenemos la misma agregación que en la búsqueda primer mejor, el tiempo computacional como en el anterior dataset, es similar, una cantidad de cálculo innecesario, ~15.5 veces más iteraciones que en algoritmo anterior, aunque la solución parece factible, estamos estresando más a nuestra máquina de manera innecesaria.

3.2.4. Análisis Tabla de Resultados Medios.

10%	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
COPKM	0.90	492.8	3.20	1.94s
BL-PM	0.67	0.00	0.67	3.04s (1556 its)
BL-MV	0.67	0.00	0.67	51.04s (30660 its)

20%	**IRIS (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
COPKM	0.98	727,6	4.10	1.73s
BL-PM	0.67	0.00	0.67	3.63s (1781 its)
BL-MV	0.67	0.00	0.67	47.82s (27540 its)

Como podemos observar, está claro que Greedy en ambos casos, incumple cada vez más restricciones a mayor número de restricciones impuestas haya. Podríamos intuir, pero no afirmar, que el tiempo de ejecución de Greedy es menor cuanto más número de restricciones ya que ayudaría a la conducir a la convergencia del problema.

En ambas propuestas de búsqueda, el resultado es el mismo, aunque de igual forma al dataset anterior, requiere de mucho cómputo innecesaria la búsqueda del mejor vecino.

Al tener una dimensionalidad baja y un bajo número de clusters, el dataset es similar a “rand” y por tanto, los tiempo de ejecución son similares

3.3 Dataset “Ecoli”.

El factor obtenido es $\lambda = 0.0268$, un problema con 8 clusters.

3.3.1. Análisis Tabla de Resultados Greedy.

10% COPKM	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	23.54	1326.00	59.12	208.91s
Ejecución 2	28.54	1526.00	69.49	201.19s
Ejecución 3	26.73	1622.00	70.25	447.20s
Ejecución 4	32.74	1316.00	68.05	252.68s
Ejecución 5	34.22	1470.00	73.66	145.05s
Media	29.16	1452.00	68.11	251.01s

20% COPKM	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	26.07	2574.00	60.60	139.98s
Ejecución 2	29.19	2760.00	66.22	198.87s
Ejecución 3	32.94	3356.00	77.96	332.54s
Ejecución 4	30.97	2294.00	61.75	258.97s
Ejecución 5	33.18	2704.00	69.45	252.50s
Media	30.47	2737.60	67.20	236.57s

Comenzamos a incrementar la complejidad del problema, los tiempos de ejecución son 100 veces mayores, la *infeasibility* al igual que el anterior, se ve duplicado y casi triplicado al aumentar las restricciones, además, ahora el factor es mayor, lo que hace que dé más importancia a este, por lo que el agregado es más alto.

La distancia media intracluster es al parecer bastante dispersa y alta en algunos casos, al igual que la infactibilidad y por ello la agregación.

3.3.2. Análisis Tabla de Resultados Búsqueda Local, Primer Mejor.

10% BL-PM	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	22.57	32.00	23.43	248.91s (22842 its)
Ejecución 2	22.72	36.00	23.69	210.88s (19171 its)
Ejecución 3	22.56	42.00	23.68	289.42s (27032 its)
Ejecución 4	22.58	34.00	23.50	292.90s (26797 its)
Ejecución 5	26.29	172.00	30.91	210.31s (19129 its)
Media	23.35	63.20	25.04	250.48s (22994 its)

20% BL-PM	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	22.49	84.00	23.58	256.63s (22880 its)
Ejecución 2	21.98	134.00	23.78	253.50s (23528 its)
Ejecución 3	22.39	90.00	23.60	185.05s (16341 its)
Ejecución 4	22.36	116.00	23.92	222.64s (20570 its)
Ejecución 5	22.27	86.00	23.43	186.59s (16803 its)
Media	22.29	102.0	23.66	220.88s (20024 its)

Curiosamente, los tiempos de ejecución del algoritmo Greedy y de búsqueda local son muy similares, esto puede ocurrir ya que en problemas de alta dimensionalidad, existe un hipervolumen mayor donde los centroides pueden posicionarse, además al haber 8 de ellos, deben estar bien dispersos, por lo que Greedy necesite más tiempo para encontrar una solución y mejor primero, consiga converger rápidamente viajando por el mejor vecino a la vez que optimiza la distancia intra-cluster y las restricciones.

Podemos observar además que las tasas de *infeasibility* son muy inferiores a greedy, además de ser capaz de encontrar unos centroides homogéneos que agrupen reduciendo la distancia intra-cluster, apoyando la hipótesis de lo expuesto anteriormente.

3.3.3. Análisis Tabla de Resultados Búsqueda Local, Mejor Vecino.

10% BL-MV	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	44.78	3040	126.25	901.13s (MAX its)
Ejecución 2				s (its)
Ejecución 3				s (its)
Ejecución 4				s (its)
Ejecución 5				s (its)
Media				s (its)

20% BL-MV	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1				s (its)
Ejecución 2				s (its)
Ejecución 3				s (its)
Ejecución 4				s (its)
Ejecución 5				s (its)
Media				s (its)

Puesto que la primera iteración, el algoritmo no encontró una solución aceptable ya excedió el número de llamadas a la función “f(...)”, la duración de la primera ejecución fue de alrededor de 15 minutos. Esto se puede ver ya que comparándolo con el problema anterior, la tasa de *infeasibility* es muy alta, lo que no tiene ningún sentido puesto que al menos, la solución debería converger a la del mejor vecino.

Se intentó realizar otra prueba de un millón de iteraciones máximas, después de 1h y 30min, todavía no había encontrado solución, por lo que me rendí.

3.3.4. Análisis Tabla de Resultados Medios.

10%	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
COPKM	29.16	1452.00	68.11	251.01s
BL-PM	23.35	63.20	25.04	250.48s (22994 its)
BL-MV				

20%	**ECOLI (k = 8)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
COPKM	30.47	2737.60	67.20	236.57s
BL-PM	22.29	102.0	23.66	220.88s (20024 its)
BL-MV				

Como podemos observar, Greedy nos ofrece una solución en tiempos de ejecución similares a los de búsqueda local, su explicación, los expuesto en la hipótesis anterior. Vemos que búsqueda local en problemas más complejos, sigue dando mejor resultado que Greedy, una baja agregación además de una baja *infeasibility.*

Además, sigue cumpliendo la duplicidad de la violación de restricciones según el aumento de restricciones, algo que ya habíamos observado anteriormente en los diferentes datasets.

Este dataset ya es caracterizado como complejo, su dimensionalidad es de 7, donde vamos a agrupar las entidades utilizando 8 clusters (Para más información sobre la complejidad del dataset, véase [2]).

3.4 Dataset “Newthyroid”.

El factor obtenido es $\lambda = 0.0365$, Con una agrupación de 3 clusters.

3.4.1. Análisis Tabla de Resultados Greedy.

10% COPKM	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	13.21	1034.00	51.03	8.67s
Ejecución 2	12.84	750.00	40.27	5.48s
Ejecución 3	13.37	782.00	41.97	5.43s
Ejecución 4	12.91	720.00	39.24	4.33s
Ejecución 5	10.32	1398.00	61.45	4.51s
Media	12.53	936.80	46.79	5.68s

20% COPKM	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	12.95	1802.00	45.89	9.33s
Ejecución 2	12.99	1366.00	37.96	4.58s
Ejecución 3	13.07	1448.00	39.55	5.61s
Ejecución 4	12.84	1510.00	40.45	4.60s
Ejecución 5	13.36	2506.00	59.17	4.73s
Media	13.04	1726.40	44.60	5.77s

Este dataset se introdujo con posterioridad a los que ya teníamos, es un dataset intermedio entre los dos primero y el último, este dataset se diferencia porque tiene pocos cluster pero una alta dimensionalidad, en este caso, es 5.

Podemos ver que la dimensionalidad del espacio no afecta significativamente al tiempo, podríamos afirmar que el tiempo de ejecución se ha duplicado con respecto de los anteriores, pero sí que afecta la cantidad de clusters, ya que en el apartado anterior, este afectaba considerablemente a los tiempos.

3.4.2. Análisis Tabla de Resultados Búsqueda Local, Primer Mejor.

10% BL-PM	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	14.09	4.00	14.23	9.42s (2740 its)
Ejecución 2	14.09	4.00	14.23	9.87s (2883 its)
Ejecución 3	10.09	194.00	17.99	11.14s (3466 its)
Ejecución 4	14.09	4.00	14.23	8.84s (2703 its)
Ejecución 5	10.87	192.00	17.90	15.37s (4929 its)
Media	12.81	79.6	15.72	10.93s (3344 its)

20% BL-PM	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	14.29	0.00	14.29	8.43s (2499 its)
Ejecución 2	10.81	462	19.26	7.95s (2413 its)
Ejecución 3	10.86	462	19.30	11.16s (3374 its)
Ejecución 4	14.29	0.00	14.29	8.13s (2481 its)
Ejecución 5	14.29	0.00	14.29	7.30s (2190 its)
Media	12.91	184.8	16.28	8.59s (2591 its)

Vemos que los tiempos de ejecución se han duplicado con respecto del anterior. Además, vemos que es muy importante la configuración inicial ya que en algunos casos, el *infeasibility* es muy alto mientras que en otros, cero. Podemos observar una homogeneidad en la desviación de los centroides, en aquellos que su desviación es baja, las violaciones aumentan.

El número de iteraciones a converger se ve incrementado aproximadamente 1.3 veces por dimensión, pero aún así, podríamos considerar ~10s como aceptables para encontrar una solución.

3.4.3. Análisis Tabla de Resultados Búsqueda Local, Mejor Vecino.

10% BL-MV	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	14.09	4.00	14.23	180.81s (62780 its)
Ejecución 2	14.09	4.00	14.23	204.91s (69660 its)
Ejecución 3	14.09	4.00	14.23	215.48s (70520 its)
Ejecución 4	14.09	4.00	14.23	181.88s (60630 its)
Ejecución 5	14.09	4.00	14.23	196.49s (66220 its)
Media	14.09	4.00	14.23	195.91s (65962 its)

20% BL-MV	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
Ejecución 1	14.29	0.0	14.29	183.64s (62350 its)
Ejecución 2	14.29	0.0	14.29	201.02s (66650 its)
Ejecución 3	14.29	0.0	14.29	187.91s (61490 its)
Ejecución 4	14.29	0.0	14.29	178.54s (58480 its)
Ejecución 5	14.29	0.0	14.29	184.76s (60200 its)
Media	14.29	0.0	14.29	187.17s (61834 its)

Claramente podemos observar una homogeneidad en los resultados, viendo que la búsqueda por el primer vecino no los ofrecía y este sí, pero dando a cambio un incremento de peso de cómputo, haciendo que los tiempos de ejecución se multipliquen por 20, casi tocando el techo del límite máximo de iteraciones.

Aunque el resultado es factible, su tiempo no, suponiendo por la tabla anterior que tenemos ½ de probabilidad de obtener un resultado como los obtenidos aquí, resulta más eficiente ejecutar varias veces la búsqueda local por mejor primero antes que realizar una búsqueda exhaustiva por todos los vecinos.

3.4.4. Análisis Tabla de Resultados Medios.

10%	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
COPKM	12.53	936.80	46.79	5.68s
BL-PM	12.81	79.6	15.72	10.93s (3344 its)
BL-MV	14.09	4.00	14.23	195.91s (65962 its)

20%	**NEWTHYROID (k = 3)**			
	Tasa_C	***Tasa_inf***	***Agr.***	***T***
COPKM	13.04	1726.40	44.60	5.77s
BL-PM	12.91	184.8	16.28	8.59s (2591 its)
BL-MV	14.29	0.0	14.29	187.17s (61834 its)

Como hemos comentado anteriormente, el algoritmo Greedy nos ofrece una solución en poco tiempo, aunque no respete mucho la infactibilidad, si lo hace con la distancia intra-cluster. Vemos que búsqueda local es muy bueno en encontrar soluciones aceptables en un tiempo que aunque superior a Greedy, aceptable.

Remarcar que entre ambas búsqueda, con este dataset, la búsqueda por todos los vecinos da resultados más homogéneos que primer mejor, pero nos ofrece una desventaja muy importante, el tiempo de cómputo es exponencialmente superior, es por eso que aunque los resultados no sean los más homogéneos, sin duda optaría por la versión primer mejor.

3.5 Análisis Global de los resultados.

Una vez analizados los diferentes datasets, podemos sacar conclusiones claras, Greedy es muy rápido para darnos una solución factible, aunque no el mejor, Véase la imagen inferior (Imagen 1) para “Greedy con 20% de restricciones”, esta imagen muestra el mejor de las iteraciones. Este resultado muestra cómo en pocos segundos, se puede obtener una solución interesante, aunque no la mejor.

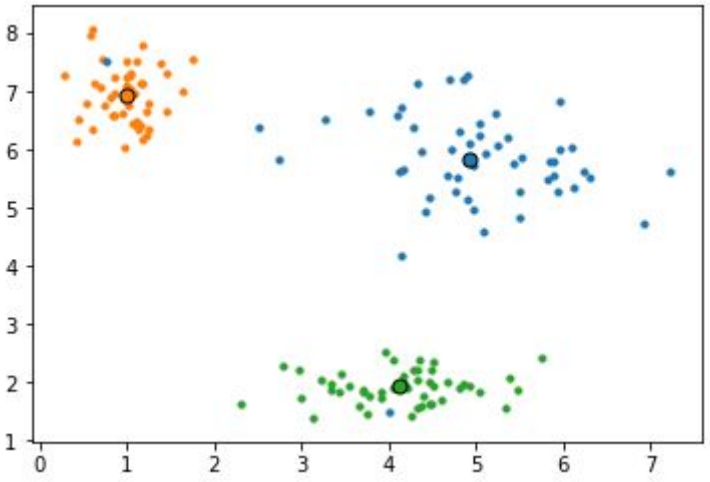

Imagen 1: Greedy 20% restricciones Agregado 1,01

Hemos podido observar que los tiempos de ejecución de los algoritmos han sido muy dependientes de la cantidad de clusters utilizados “k” y de la dimensionalidad. Como hemos pasado de problemas que tardaron pocos segundos a otros, minutos. Pero en todos ellos, obtener resultados válidos para los datasets. De obtener resultados con una gran violación de restricciones como en el caso de Greedy a encontrar una posible solución con Búsqueda Local, estudiar diferentes restricciones a la búsqueda local, como primer mejor o mejor vecino.

Búsqueda Local para primer mejor ha resuelto el problema en tiempos aceptables en comparación con Greedy, dando resultados muy superiores, minimizando el Agregado, explorando únicamente lo necesario. Como hemos visto, nos ha dado en algunos casos, resultados homogéneos, como en “iris” o en “rand”, mientras que en otros casos, no tanto, es el caso de “newthyroid”. Mientras que la Búsqueda Local por el Vecindario, ofrece resultados homogéneos, pero con un altísimo tiempo de cómputo.

El motivo principal por el cual Búsqueda Local es mejor que Greedy es debido a que es capaz de dar marcha atrás, mientras que Greedy, una vez tomado un cluster,

este no vuelve a cambiarlo. Por ello, con Greedy hemos ido arrastrando cada vez más violaciones a las restricciones.

Para el dataset de “rand”, con Búsqueda Local Primer mejor, hemos optimizado la función de distancia intra-cluster y el número de restricciones violadas, quedándonos con el primer mejor vecino, dando más riqueza y modificaciones a mismas entidades que ya tenían un cluster asociado.

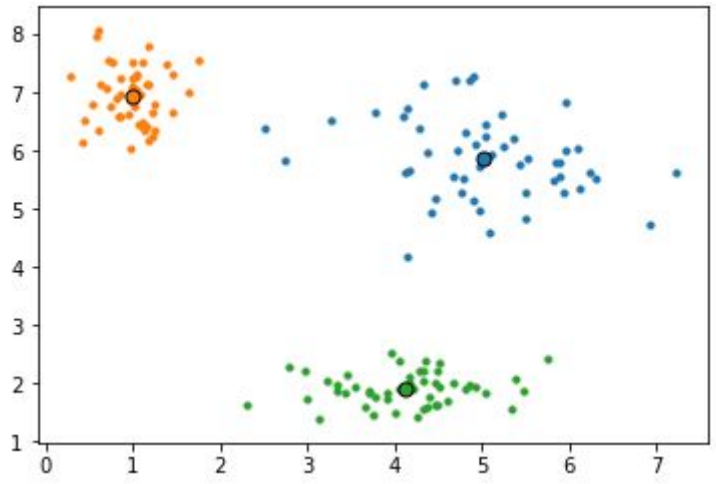

Imagen 2: Búsqueda Local Primer Mejor 20% restricciones Agregado 0,72

Además, podemos observar que para la mayoría de problemas, la relación calidad/tiempo de cómputo de la solución entre las dos Búsquedas Locales, está muy por encima la búsqueda del primer mejor, ya que explorar todo el vecindario se hace una tarea costosa y como hemos visto, los resultados no están muy alejados entre las diferentes búsquedas. Una estrategia probabilística, sería realizar diferentes Búsquedas Locales con primer mejor y quedarnos con la mejor, ya que en la mayoría de los casos, el tiempo de cómputo de 100 veces inferior.

Finalmente, remarcar que este tipo de problemas aún teniendo una gran complejidad algorítmica, hemos sido capaces de utilizar diferentes métodos para encontrar soluciones, que en la mayoría de los casos reducen exponencialmente el tiempo necesitado y cuya solución es más que aceptable.

4. Bibliografía

1. *K. Alsabti, S. Ranka, V. Singh*. (**1998**). **An Efficient K-Means Clustering Algorithm**. "*Inthispaper, we present a novel algorithm for performing k-means clustering. It organizes all the patterns in a k-d tree structure such that one can find all the patterns which are closest to a given prototype efficiently. The main intuition behind our approach is as follows. All the prototypes are potential candidates for the closest prototype at the root level.However, for the children of the root node, we may be able to prune the candidate set by using si...*". Obtenido de https://www.cs.utexas.edu/~kuipers/readings/Alsabti-hpdm-98.pdf
2. *P. Krömer, H. Zhang, Y. Liang, J-S. P. S*. (**2018**) .**Proceedings of the Fifth Euro-China Conference on Intelligent Data Analysis**. *"This volume of Advances in Intelligent Systems and Computing highlights papers presented at the Fifth Euro-China Conference on Intelligent Data Analysis and Applications (ECC2018), held in Xi'an, China from October 12 to 14 2018. The conference was co-sponsored "* Obtenido de https://books.google.es/books?id=GzKBDwAAQBAJ&pg=PA513&lpg=PA513&dq=newthyroid+dataset+kmean&source=bl&ots=04b14UkELW&sig=ACfU3U1C4JwYT8QMCkFrC79eNn-ShPEDVA&hl=es&sa=X&ved=2ahUKEwiQjomnm67oAhWcA2MBHVgVCAwQ6AEwCnoECAkQAQ#v=onepage&q=newthyroid%20dataset%20kmean&f=false
3. *C. Blum, A. Roli* (**2003**). **Metaheuristics in Combinatorial Optimization: Overviewand Conceptual Comparison**. "*The field of metaheuristics for the application to combinatorial optimization problems isa rapidly growing field of research. This is due to the importance of combinatorialoptimization problems for the scientific as well as the industrial world...*" Obtenido de https://www.iiia.csic.es/~christian.blum/downloads/blum_roli_2003.pdf