UNIVERSIDAD
DE GRANADA

Grupo 2 – Miercoles 17.30 - 19.30

Metaheurísticas — Técnicas de Búsqueda Local y Algoritmos Greedy para el Problema del Agrupamiento con Restricciones

---

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

Marzo 23, 2020

Índice general

1. Descripción del problema **2**
1.1. Formalización de los datos . . . 2

2. Descripción de los algoritmos empleados **3**
2.1. Representación de los datos . . . 3
2.2. Operadores comunes . . . 3

3. Métodos de búsqueda **5**
3.1. Búsqueda Local . . . 5
3.1.1. Función objetivo . . . 5
3.1.2. Datos propios . . . 5
3.1.3. Descripción del algoritmo . . . 5
3.1.4. Pseudocódigo del algoritmo . . . 6
3.1.5. Operadores propios . . . 6

4. Descripción de los algoritmos de comparación **8**
4.1. K-medias Restringido Débil . . . 8
4.1.1. Función objetivo . . . 8
4.1.2. Descripción del algoritmo . . . 8
4.1.3. Pseudocódigo del algoritmo . . . 8
4.1.4. Operadores propios . . . 9

5. Procedimiento **10**
5.1. Estructura de datos . . . 10
5.2. Guía de Uso . . . 10

6. Experimentos y analisis de resultados **11**
6.1. Semillas . . . 11
6.2. Análisis . . . 11
6.3. Resultados . . . 13

Apartado 1:
Descripción del problema

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

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

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

§1.1: Formalización de los datos

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

Apartado 2:
Descripción de los algoritmos empleados

En esta práctica se han implementado dos algoritmos:

- **K-medias Restringido Débil**: Algoritmo de heurística greedy, que busca minimizar tanto la desviación general como la *infeasibility*. $F_{objetivo} = \bar{C} + infeasibility$
- **Búsqueda Local**: Como método de búsqueda por trayectorias utilizaremos la búsqueda local, en este caso, se busca minimizar la desviación general y la *infeasibility* multiplicado por un parámetro λ que se describirá mas adelante. $F_{objetivo} = \bar{C} + (infeasibility \cdot \lambda)$

§2.1: Representación de los datos

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

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

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

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

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

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

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

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

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

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

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

§2.2: Operadores comunes

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

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

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

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

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

```
Para todos los clusters:
   d_intracluster = 0

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

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

```
desv_gen = 0

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

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

```
incumplidas = 0

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

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

Devolver incumplidas
```

Apartado 3:
Métodos de búsqueda

§3.1: Búsqueda Local

§3.1.1: Función objetivo

Como ya he explicado anteriormente, el objetivo de la búsqueda local es minimizar tanto la desviación general de los elementos, como el número de restricciones incumplidas, si embargo se introduce un parámetro de escalado λ como una manera de dar relevancia a la infactibilidad.

$F_{objetivo} = \bar{C} + (infeasibility \cdot \lambda)$

En nuestro caso $\lambda = \frac{\lceil D \rceil}{|R|}$ donde D es la máxima distancia entre dos elementos de X

§3.1.2: Datos propios

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

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

§3.1.3: Descripción del algoritmo

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

§3.1.4: Pseudocódigo del algoritmo

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

solucion_inicial()
generar_vecindario()

f_objetivo_ant = f_objetivo
solucion_ant = solucion
infactibilidad_ant = infactibilidad

Hacer:
   generar_vecino()
   i++

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

   Si no:
      solucion = solucion_ant
      infactibilidad = infactibilidad_ant

   Si no quedan vecinos que generar:
      Leer solucion actual

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

§3.1.5: Operadores propios

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

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

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

Ordenar index aleatoriamente

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

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

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

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

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

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

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

lambda = d_max / restricciones
```

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

```
Vaciar vecindario

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

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

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

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

   Si Pareja (pos,n) existe en vecindario:
      salir = verdadero
      Borrar Pareja (pos,n) de vecindario
      c = solucion[pos]
      Marcar solucion[pos] como invalido

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

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

      infactibilidad -= restricciones_incumplidas(pos,c)
      infactibilidad += restricciones_incumplidas(pos,n)
      solucion[pos] = n
      Añadir pos al cluster n
      desviacion_general()
      f_objetivo = desv_gen + (infactibilidad*lambda)
```

Apartado 4:
Descripción de los algoritmos de comparación

§4.1: K-medias Restringido Débil

§4.1.1: Función objetivo

El objetivo del algoritmo K-medias Restringido Débil — de heurística greedy — Es minimizar tanto la desviación general del problema como el numero de restricciones que no se satisfacen (infactibilidad).

$F_{objetivo} = \bar{C} + infeasibility$

§4.1.2: Descripción del algoritmo

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

§4.1.3: Pseudocódigo del algoritmo

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

Guardar en solucion_ant la matriz cluster

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

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

   Si cambio_c:
      Vaciar los clusters
```

```
   i++
Mientras: cambio_c

desviacion_general()
f_objetivo = desv_gen + infactibilidad
```

§4.1.4: Operadores propios

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

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

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

Ordenar r en orden ascendente

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

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

Apartado 5: Procedimiento

§5.1: Estructura de datos

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

```cpp
int n_cluster;
std::vector<std::vector<double>> posiciones;
std::vector<std::vector<double>> centroides;
std::map<std::pair<int,int>,int> restricciones;
std::set<std::pair<int,int>> vecindario;
std::vector<std::vector<int>> clusters;
std::vector<double> d_intracluster;
std::vector<int> solucion;
double desv_gen;
double infactibilidad;
double lambda;
double f_objetivo;
```

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

Los operadores de los algoritmos descritos anteriormente se implementan como métodos de la clase CCP.

§5.2: Guía de Uso

El programa el muy sencillo de compilar. Con la orden **make** el programa se compila y se ejecuta. Los datos se cargan automáticamente desde los ficheros.
La estructura de ficheros es la siguiente:

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

Apartado 6:
Experimentos y analisis de resultados

§6.1: Semillas

Para la ejecución de los algoritmos se han seleccionado las siguientes semillas, utilizando un algoritmo para probar que no producen ciclos en ninguna de las ejecuciones del algoritmo greedy para ninguno de los conjuntos de datos. Toda semilla que en menos de 1000 iteraciones del algoritmo greedy no obtenga resultado es rechazada. La búsqueda de las semillas ha sido relativamente compleja ya que la mayoria de las primeras semillas que probé no obtenían resultados para el data set *Ecoli*.

- 1584565171
- 1584764782
- 1584565259
- 1584564539
- 1522565615

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

§6.2: Análisis

Primero empezare analizando los resultados de los data sets *Iris* y *Rand*, porque considero que son similares entre ellos y ambos difieren bastante de *Ecoli*.

Para un conjunto de restricciones del 10 % del data set *Iris* se puede notar que las soluciones que aporta el algoritmo greedy no convergen a la solución óptima — debido a las semillas utilizadas — sin embargo la desviación general es bastante homogénea, nunca superando un valor de 1. La infactibilidad de estas soluciones si es bastante elevada sobre todo en la segunda ejecución, cosa que perjudica enormemente a la función objetivo, que en greedy se calcula como la suma de la desviación general y la infactibilidad.
Podemos notar así que cuando comparamos las soluciones greedy contra las soluciones BL —para *Iris* 10 %— el agregado de las solucione BL es mucho menor, cuando la infactibilidad de las mismas es muy elevada. Aquí entra en juego el parámetro de escalado λ. Las soluciones BL para *Iris* 10 % obtienen una desviación general menor y mas homogénea que las soluciones greedy. Como resultado la BL obtiene clara ventaja sobre greedy, porque en BL a coste de disminuir la desviación general se ha aumentado la infactibilidad, que al ser escalada por λ disminuye considerablemente su efecto en la función objetivo. Notar también, que los resultados de greedy dependen enormemente de como de buena sea la colocación inicial de los centroides, cosa que depende de la semilla escogida, para un ajuste mejor de los centroides, es probable que greedy supere o iguale a los resultados ofrecidos por BL.
Para un conjunto de restricciones del 20 % si vemos que algunas soluciones convergen a la optima —probablemente por lo que ya hemos comentado de la buena colocación inicial de los centroides— o se quedan muy cerca de estas, aunque hay ejecuciones con un agregado muy alto, debido a la infactibilidad elevada. Para BL los resultados de desviación general y sobre todo de infactibilidad empeoran notablemente (casi se duplican) pero ya vemos que el agregado no se resiente en gran medida por el efecto escalado de λ
Como conclusión para ambos conjuntos de restricciones, las soluciones BL superan bastante a las soluciones greedy, no tanto porque obtengan un buen reparto de datos si no por como se calcula la función objetivo de

ambos algoritmos.

El data set *Rand* sin embargo es todo lo contrario a *Iris*.
Las soluciones greedy convergen —Para conjuntos de 10 % y 20 % de restricciones— a la solucion optima casi en todas las ejecuciones, podría deberse quizá a que exista una mayor distribución de los datos y los centroides se posicionen favorablemente con mayor posibilidad en *Rand* que en *Iris*. El agregado en *Rand* es muy bueno, porque al converger a la solución optima o quedarse muy cerca, este no aumenta debido a la infactibilidad nula.
En las soluciones ofrecidas por BL sin embargo, podemos ver que tanto para conjuntos de 10 % y 20 % de restricciones, la solución parece estancarse en un mínimo local, ya que ni la desviación general ni la infactibilidad mejoran a las aportadas por greedy. Las soluciones aportadas por BL depende en gran medida del punto de partida, es decir, de como de buena o mala sea la solución inicial aleatoria —lo lejos que se encuentre de un optimo local o global—.
Parece que en esta ocasión —por las semillas escogidas— la solución inicial de BL hace que las soluciones finales no converjan en la optima, si no en un optimo local. El parámetro λ suaviza mucho los malos resultados obtenidos, pero en este caso, las soluciones greedy superan en gran medida a las soluciones aportadas por BL.

Pasemos ahora a analizar los resultados del data set *Ecoli*, sin duda el data set que mas problemas ha dado a la hora de seleccionar semillas que no produzcan ciclos infinitos en greedy.
En primer lugar notar que las desviaciones generales en el data set *Ecoli* son mucho mas elevadas que en los data set *Iris* y *Rand*. Esto es debido a que para calcular la distancia euclídea entre dos elementos, prescindo de la operación raíz cuadrada, debido a lo costosa que es y que en esencia es innecesaria para comparar distancias —si aplico la misma operación a ambos elementos estos varían de igual forma—. En *Ecoli* la distancia entre los elementos es considerablemente mayor.
En las soluciones greedy tanto para conjuntos de 10 % y 20 % de restricciones, se puede observar que la desviación general de las soluciones es bastante homogénea y la infactibilidad es algo elevada, con respecto a otros data sets, pero esto es porque en *Ecoli* al existir mas elementos, existen mas restricciones posibles. Como resultado, el agregado de los datos es bastante homogéneo y no demasiado superior a la desviación general de las soluciones.
Sin embargo si nos fijamos en las soluciones que consigue BL, podemos hacer dos distinciones:
Para un conjunto de restricciones del 10 % vemos que las soluciones obtenidas mejorar enormemente a las greedy en cuanto a desviación general se refiere, esto indica que los elementos en los clusters están mas homogéneamente repartidos, aunque debido a esto, también aumenta (prácticamente duplica) la infactibilidad. Es sorprendente con esto datos, notar que el agregado de estas soluciones es muy similar al que aportan las soluciones greedy. Este resultado se produce porque λ para este conjunto de restricciones duplica el valor de la infactibilidad, consiguiendo que el agregado obtenga un valor mucho mayor. Esto indica que para este conjunto de restricciones, prima mucho mas el reducir el valor de la infactibilidad que el encontrar un mejor reparto de los elementos x_i.
Para un conjunto de restricciones del 20 % esto no sucede, ya que el parámetro λ no afecta demasiado al peso de la infactibilidad en la solucion. Podemos ver que la desviación general obtenida es similar a la obtenida para un 10 % de restricciones e inferior a la obtenida para soluciones greedy, sin embargo, se obtiene una infactibilidad muy elevada, que perjudica en gran medida al calculo del agregado.
Para ambos algoritmos las soluciones obtenidas para ambos conjuntos de restricciones son muy similares, aunque para los resultados empíricos obtenidos, BL supera a greedy.

Tras analizar los resultados, podemos notar como aunque *Iris* y *Rand* sean en un principio bastante parecidos —ya que ambos agrupan los elementos en 3 clusters— obtiene resultados diametralmente opuesto, por como se reparten los datos en el espacio n-dimensional y como de buenas son las posiciones iniciales de los centroides y las soluciones de partida de las búsqueda local.
El data set *Ecoli* obtiene soluciones que difieren bastante de los otros data sets, aunque estas soluciones son muy similares entre las obtenidas por greedy y las obtenidas por BL.

§6.3: Resultados

Tabla 5: Resultados globales en el PAR con 10 % de restricciones

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

Tabla 6: Resultados globales en el PAR con 20 % de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**
K-medias Restringido Débil	0,60	31	31,60	0,005	1.659,37	211,60	1.870,97	0,351	0,85	4	4,45	0,004
Búsqueda Local	1,37	511	7,11	0,036	715,42	1.015	1.705,59	1,403	3,50	509	10,89	0,044

Tabla 1: Resultados obtenidos por el algoritmo greedy en el PAR con 10 % de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**
Ejecución 1	0,77	25	25,77	0,004	1.644,41	232	1.876,41	0,239	0,95	19	19,95	0,017
Ejecución 2	0,78	70	70,78	0,005	1.583,88	206	1.789,88	0,068	0,85	0	0,85	0,006
Ejecución 3	0,71	54	54,71	0,005	1.574,02	133	1.707,02	0,192	0,85	0	0,85	0,006
Ejecución 4	0,66	29	29,66	0,005	1.666,63	152	1.818,63	0,102	0,84	11	11,84	0,004
Ejecución 5	0,59	4	4,59	0,005	1.528,26	225	1.753,26	0,165	0,85	0	0,85	0,004
Media	0,70	36	37,10	0,005	1.599,44	190	1.789,04	0,153	0,87	6	6,87	0,007

Tabla 2: Resultados obtenidos por el algoritmo greedy en el PAR con 20 % de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**
Ejecución 1	0,60	0	0,60	0,004	1.742,61	511	2.253,61	0,877	0,85	0	0,85	0,004
Ejecución 2	0,60	33	33,60	0,006	1.519,48	75	1.594,48	0,087	0,85	0	0,85	0,004
Ejecución 3	0,62	80	80,62	0,004	1.717,16	180	1.897,16	0,328	0,85	0	0,85	0,004
Ejecución 4	0,60	10	10,60	0,006	1.633,66	162	1.795,66	0,321	0,84	18	18,84	0,004
Ejecución 5	0,59	32	32,59	0,006	1.683,96	130	1.813,96	0,140	0,85	0	0,85	0,004
Media	0,60	31	31,60	0,005	1.659,37	211,60	1.870,97	0,351	0,85	4	4,45	0,004

Tabla 3: Resultados obtenidos por el algoritmo BL en el PAR con 10 % de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**
Ejecución 1	0,60	268	6,63	0,034	644,71	474	1.604,84	1,149	1,97	259	9,49	0,036
Ejecución 2	2,71	231	7,90	0,033	656,85	481	1.631,16	1,259	1,97	259	9,49	0,035
Ejecución 3	0,60	268	6,63	0,040	694,25	470	1.646,27	1,710	1,97	259	9,49	0,031
Ejecución 4	0,60	268	6,63	0,030	718,13	459	1.647,88	1,264	3,99	241	10,99	0,037
Ejecución 5	0,60	268	6,63	0,033	644,71	474	1.604,84	1,370	1,97	259	9,49	0,037
Media	1,03	261	6,88	0,034	671,73	472	1.627,00	1,350	2,37	255	9,79	0,036

Tabla 4: Resultados obtenidos por el algoritmo greedy en el PAR con 20 % de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**
Ejecución 1	0,56	517	6,36	0,033	640,20	1.059	1.673,09	1,295	3,77	500	11,03	0,055
Ejecución 2	2,59	502	8,23	0,042	737,17	999	1.711,54	1,159	3,78	499	11,02	0,042
Ejecución 3	2,59	500	8,21	0,038	736,22	1.006	1.717,42	1,528	1,97	543	9,85	0,029
Ejecución 4	0,56	517	6,36	0,031	739,16	1.003	1.717,44	1,182	4,11	502	11,40	0,042
Ejecución 5	0,56	517	6,36	0,037	724,361	1.009	1.708,49	1,850	3,87	501	11,14	0,052
Media	1,37	511	7,11	0,036	715,42	1.015	1.705,59	1,403	3,50	509	10,89	0,044