Práctica 1b
Técnicas de Búsqueda Local y Algoritmos Greedy
Problema de Agrupamiento con Restricciones

Sergio Quijano Rey - 72103503k
4º Doble Grado Ingeniería Informática y Matemáticas
Grupo de prácticas 2 - Viernes 17.30h a 19.30h
sergioquijano@correo.ugr.es

16 de abril de 2021

Índice

1. Descripción del problema

Vamos a trabajar el problema del agrupamiento con restricciones (***PAR***). Consiste en una generalización del problema de agrupamiento clásico, al que añadimos restricciones sobre los datos.

El problema de agrupamiento clásico consiste en, dados unos datos de entrada sin etiquetar X de tamaño n, agruparlos en k grupos (o en inglés, *clusters*) diferentes, formando una partición C de X, de forma que se optimice alguna métrica. Normalmente, se busca minimizar la distancia *intra_cluster* (que más tarde se definirá).

La diferencia con el problema de agrupamiento clásico, por tanto, es la inclusión de restricciones. En nuestro caso concreto, trabajemos con restricciones entre pares de puntos, que además serán de dos tipos:

- Restricción tipo *Must Link*: los dos puntos afectados por esta restricción deberán pertenecer al mismo cluster
- Restricción tipo *Cannot Link*: los dos puntos afectados por esta restricción no deben pertenecer al mismo cluster

Consideraremos de forma débil estas restricciones, es decir, podemos incumplir algunas restricciones. Pero supondrá que la solución será de peor calidad. Para especificar mejor esta noción, definimos la función de *fitness* que buscamos minimizar:

$$fitness(sol) := distancia_{intra-cluster}(sol) + \lambda * infeasibility(sol)$$

donde $infeasibility$ es el número de restricciones que se incumplen. Esta función de $fitness$ nos permite pasar de intentar optimizar dos funciones objetivo a solo tener que optimizar un objetivo. El valor de λ se especifica más adelante.

Como los datos no están etiquetados a priori, podríamos considerar este problema como un problema de aprendizaje no supervisado. Sin embargo, se puede considerar que las restricciones nos dan un tipo de etiquetado, por lo que es más correcto pensar que estamos ante una tarea de aprendizaje

semi-supervisado. La principal utilidad de resolver estos problemas es que normalmente estamos reduciendo la dimensionalidad de los datos a analizar, y de este modo, es más sencillo extraer conocimiento sobre dichos datos.

2. Descripción de la aplicación de los algoritmos empleados

2.1. Representación del conjunto de datos

Los datos vienen dados en una matriz de tamaño $n \times d$ donde n es el número de puntos y d es la dimensión de cada uno de los puntos.

2.1.1. Representación del conjunto de datos en código

El conjunto de datos viene representado en problem_datatypes::DataPoints que contiene un vector de otro tipo de dato: problem_datatypes::Point. El tipo de dato Point tiene un campo que es de tipo ndarray::Array1<f64> que representa un vector (usamos una librería para trabajar con matrices y vectores). Por tanto, hemos pasado de trabajar con una matriz de datos a trabajar con un vector de puntos. Esto nos permite trabajar de forma más expresiva y sencilla con el problema. Por ejemplo, podemos calcular con métodos del struct distancia entre dos puntos, centroide de un conjunto de puntos, ...

2.2. Representación de las restricciones

Las restricciones vienen dadas en un fichero de datos que representa una matriz que codifica las restricciones de la siguiente forma:

- El elemento en la fila i-ésima y columna j-ésima representa las restricciones que hay entre el punto i y el punto j
- Como la restricción que tenga el punto i con el punto j implica que el punto j tiene la misma restricción con el punto i, es claro que dicha matriz debe ser simétrica
- Un valor 0 significa que no hay restricciones. Un valor 1 significa que hay una restricción tipo *Must Link*. Un valor -1 implica una restricción *Cannot Link*
- Además, la matriz tiene la diagonal de 1s

2.2.1. Representación de las restricciones en código

El struct problem_datatypes::Constraints junto al enumerado problem_datatypes::ConstraintType representan en el código las restricciones. El código es el siguiente:

```
pub enum ConstraintType {
    MustLink,
    CannotLink,
}

pub struct Constraints{
    data: HashMap<(i32, i32), ConstraintType>,
}
```

Es claro que guardamos pares de enteros, que marcan los índices de los puntos, y la restricción entre el par de puntos representados, en un HashMap. Esta elección viene motivada por:

- Podemos acceder a las restricciones entre dos puntos en tiempo constante
- Podemos iterar sobre todas las restricciones, gracias a los métodos proporcionados por el lenguaje de programación, en un tiempo más que razonable. Así iteramos solo sobre una lista de r restricciones, en vez de sobre una matriz cuadrada de dimensión n^2
- En cierto modo, estamos combinando los beneficios de tener acceso directo a elementos concretos y los beneficios de poder iterar sobre una lista (aunque iterar sobre un Hash puede ser algo más lento que iterar sobre una lista o un array)
- Es fácil de implementar métodos para operar con restricciones con este tipo de dato

La implementación de los métodos que permiten manipular el struct aseguran que:

- No guardamos la restricción (i, j) y junto a la (j, i). Solo guardamos una de las dos restricciones, ahorrando memoria

- De hecho, el criterio es guardar como índices el par (i, j) donde $i \leq j$
- Tampoco guardamos las restricciones $(i, i), MustLink$ pues son restricciones triviales

2.3. Representación de la solución

Una representación de la solución será un vector de tamaño n con valores en $\{0, \ldots, k-1\}$ donde n es el número de puntos y k es el número de clusters en los que dividimos los datos. Este vector representa la partición de los datos en los k clusters. En la posición i-ésima del vector, guardamos el cluster al que pertenece el punto i-ésimo de nuestro conjunto de datos.

Las soluciones deben cumplir las siguientes restricciones:

- No pueden quedar clusters vacíos. Es decir, clusters a los que no haya ningún punto asignado. Esto puede verse viendo que $\forall i \in \{0, \ldots, k-1\}$ $\exists pos \in \{0, \ldots, n-1\}$ tal que $solution[pos] = i$, es decir, el vector de soluciones tiene al menos una vez cada valor posible de los clusters
- Cada punto solo puede pertenecer a un único cluster. Por la forma vectorial en la que representamos la partición, esta restricción se verifica forzosamente, y por tanto no nos tenemos que preocupar de realizar comprobaciones
- La unión de los puntos de los clusters debe ser todo el conjunto de datos, es decir, $X = \bigcup c_i$. De nuevo, nuestra representación vectorial fuerza a que esta restricción se verifique

Por ejemplo, si tenemos 5 puntos y 3 clusters, una posible solución sería $\{3, 1, 2, 3, 0\}$. Y por otro lado, la solución $\{3, 1, 2, 3, 2\}$ no es válida pues el cluster 0 está vacío.

Para cada solución podemos calcular algunas métricas necesarias para conocer el valor de $fitness$ de la solución que estamos representando. Para comenzar, por cada cluster podemos calcular el **centroide** del cluster:

$$\vec{\mu_i} := \frac{1}{|c_i|} \sum_{x_i \in c_i} \vec{x_i}$$

Definimos para cada cluster su **distancia media intra-cluster** como:

$$\bar{c_i} := \frac{1}{|c_i|} \sum_{x_i \in c_i} ||\vec{x_i} - \vec{\mu_i}||_2$$

Y con ello podemos definir la **desviación general de la partición** como:

$$\bar{c} := \frac{1}{k} \sum_{i \in 1,...k} \bar{c_i}$$

Definimos $infeasibility$ como el número de restricciones, tanto del tipo *Must Link* como del tipo *Cannot Link*, que se violan.

Con ello, ya podemos definir el valor de λ como $\lambda := \frac{D}{|R|}$ donde $|R|$ es el número total de restricciones y D la distancia máxima entre dos puntos de X.

2.3.1. Representación de la solución en código

La solución se representa en la clase problem_datatypes::Solution. El código de los campos del struct desarrollado es:

```
pub struct Solution<'a, 'b> {
    cluster_indexes: Vec<u32>,
    data_points: &'a DataPoints,
    constraints: &'b Constraints,
    number_of_clusters: i32,

    /// Representa el peso de infeasibility en el calculo de fitness
    /// Solo se calcula una vez al invocar a Solution::new
    lambda: f64,

    // Para cachear el valor de fitness pues es un calculo costoso de realizar
    // Como los datos del struct no cambian, podemos hacer el cacheo sin miedo
    // Usamos RefCell para tener un patron de mutabilidad interior
    fitness: RefCell<Option<f64>>,
}
```

Los campos del struct representan:

- cluster_indixes : el vector solución que representa la asignación de puntos a clusters
- data_points: referencia al conjunto de datos (sirve para calcular métricas como el $fitness$ de la solución que se representa)
- constraints: referencia al conjunto de restricciones sobre los datos (sirve para calcular métricas como el $fitness$ de la solución que se representa)
- number_of_clusters: número de clusters en los que se agrupan los datos (sirve para comprobar que una solución sea válida)
- lambda: valor de λ, necesario para calcular el $fitness$
- fitness : valor de $fitness$. Está incluida en un RefCell<Option<f64>> para poder cachear su valor, puesto que los atributos de una instancia nunca cambian y el cálculo del valor λ es muy costoso (implica calcular restricciones violadas y distancias entre puntos)

La comprobación de que no tenemos clusters sin puntos asignados se hace en el método Solution :: is_valid . La distancia media intracluster se calcula en Solution :: intra_cluster_distance . Mientras que la desviación general se calcula en Solution :: global_cluster_mean_distance. El valor de $infeasibility$ se calcula en Solution :: infeasibility . El cálculo de λ se realiza en el *constructor* del struct.

3. Descripción de los algoritmos empleados

3.1. Búsqueda Local

Usamos un pseudocódigo muy parecido a Python pues es muy expresivo y facilita traducir partes de nuestro código real a pseudocódigo.

Método de exploración de entorno:

```
# Estrategia el primero mejor
# Devuelve el primer vecino que mejora la solucion actual
def get_neighbour():
    # Tomamos el generador de vecinos que se describe mas adelante
    neighbours_generator = generate_all_neighbours()

    # Mezclo los generadores de vecinos
    neighbours_generator.shuffle()

    # Exploro los vecinos hasta encontrar uno mejor que esta solucion
    for current_generator in neighbours_generator:
        current_solution = self.generate_solution_from(current_generator)

        if
            current_solution.is_valid() and
            current_solution.fitness() < self.fitness():

            return current_solution

    # No hemos encontrado un vecino mejor
    return None;
}
```

Operador de generación de vecino:

```

# Struct que representa el generador de vecinos de
# forma eficiente
struct NeighbourGenerator:
    # El elemento que queremos mover de cluster
    element_index: i32,
```

```
7
8       # El nuevo cluster al que asignamos el elemento
9       new_cluster: u32,
10
11  # Funcion que genera todos los vecinos posibles de un elemento
12  # Los vecinos generados pueden ser no validos
13  def generate_all_neighbours(
14      number_of_elements,
15      number_of_clusters):
16      neighbours = []
17
18      for current_element in 0..number_of_elements:
19          for current_cluster in 0..number_of_clusters:
20              neighbours.append(NeighbourGenerator{
21                  current_element,
22                  current_cluster,
23              });
24
25      return neighbours;
```

Generación de soluciones aleatorias:

```
1   # Genera una solucion inicial aleatoria como punto de partida de las busquedas
2   # Puede dejar clusters vacios, por lo que el caller de esta funcion tiene que
3   # comprobar la validez de la solucion aleatoria, y en caso de invalidez, volver
4   # a llamar a esta funcion (es muy poco probable que con muchos puntos dejemos
5   # un cluster vacio)
6   def generate_random_solution(data_points, constraints, number_of_clusters):
7       # Vector con indices de clusters aleatorios, de tamaño el numero de puntos
8       # que trabajamos
9       random_cluster_indixes = [
10          random(0, number_of_clusters)
11          for _ in data_points.len()
12      ]
13
14      # En nuestro codigo, generamos el struct Solution a partir de los parametros
15      # de entrada y random_cluster_indixes
16      return solution_from(cluster_indexes)
17  }
```

4. Descripción y Pseudocódigo de los algoritmos de comparación

Como algoritmo de comparación estamos considerando una modificación del algoritmo clásico *K-means* al que añadimos la capacidad de considerar las restricciones: *copkmeans* o *Constrained K-means*. Por tanto, estamos ante un algoritmo *greedy*.

La idea general es:

1. Partir de una solución inicial aleatoria, que vendrá dada por una asignación de centroides de clusters aleatorios
2. Iterar sobre todos los datos en orden aleatorio, asignando a cada punto el mejor cluster en ese momento (siguiendo claramente un esquema greedy). Consideramos como mejor cluster el que menos restricciones violadas produzca, y en caso de empate, el cluster cuyo centroide sea más cercano al punto
3. Una vez acabada la asignación de todos los puntos, calcular los centroides de los clusters con la asignación actual de los puntos
4. Repetir el proceso desde 2. si los centroides han cambiado respecto de la anterior iteración

A la hora de ejecutar el algoritmo, en algunos *datasets* el algoritmo se encuentra con problemas, pues los centroides pueden oscilar infinitamente entre dos soluciones muy cercanas (debido entre otros factores a la configuración de los datos de entrada). Esta configuración de los datos también puede provocar que haya clusters que se queden sin puntos asignados, generando así una solución no válida. Por tanto, el algoritmo admite un parámetro de entrada para indicar si queremos que sea *robusto* o no. En caso de que indiquemos que queremos que sea robusto se tendrán las siguientes diferencias:

- Los centroides aleatorios no se tomarán como puntos aleatorios, sino como puntos del *dataset* aleatorios, por lo que en una primera iteración no podrán quedar clusters vacíos, aunque si podrán quedar clusters vacíos en iteraciones posteriores. Con esto se buscas evitar el problema de los clusters vacíos

- Se tendrá un máximo de iteraciones. Este máximo lo hemos establecido como 50 iteraciones sobre el bucle principal. Teniendo en cuenta que cuando no cicla infinitamente, en menos de 10 iteraciones el algoritmo encuentra solución, consideramos que es un máximo mucho más que aceptable para asegurar que la solución devuelta sea la mejor (o la segunda mejor) que el *greedy* puede calcular con esa semilla aleatoria. Con esto se busca evitar el problema del ciclado infinito
- Aún con estos cambios, en ciertas ocasiones no podemos evitar que dejemos un cluster vacío en iteraciones posteriores a la primera. Por tanto, también colocaremos un máximo de reinicios del algoritmo en la parte del código que llama al método de búsqueda.

El pseudocódigo (en notación muy parecida a Python) de nuestra implementación del algoritmo quedaría tal que:

```
# Generamos los centroides aleatorios. Dependiendo de si es robust o no
# consideramos puntos aleatorios en [0,1] x [0,1] o puntos del dataset
# de entrada aleatorios
current_centroids = generate_random_centroids()

# Solucion inicial aleatoria que se va a modificar en la primera iteracion
# Notar que no es valida porque deja todos los clusters menos uno vacíos
current_cluster_indixes = [0, 0, ..., 0]

# Para comprobar que los centroides cambien
centroids_have_changed = true


# Si robust == true, acotamos el numero maximo de iteraciones
max_iterations = 50
mut curr_iteration = 0

while centroids_have_changed == true and curr_iteration < max_iterations{

    # Realizamos una nueva asignacion de clusters. Recorremos los puntos
    # aleatoriamente y asignamos el cluster que menos restricciones viole
    # en esa iteracion. En caso de empate, se toma el cluster con centroide
    # mas cercano al punto
    new_cluster_indixes = assign_points_to_clusters()
```

```
25
26      # Comprobamos que la nueva solucion calculada es correcta
27      if valid_cluster_configuration(current_cluster_indixes) == false:
28          # Esto hace que el el caller de la funcion copkmeans, se muestre un
29          # mensaje por pantalla y se vuelva a realizar la busqueda, con lo que
30          # partimos de unos centroides aleatorios nuevos. Como ya se ha comentado,
31          # hay un maximo de reinicios en el caller para este metodo
32          return None
33
34      # Calculamos los nuevos centroides y comprobamos si han cambiado
35      new_centroids = calculate_new_centroids(new_cluster_indixes)
36      centroids_have_changed = centroids_are_different(
37          current_centroids,
38          new_centroids
39      )
40
41      # Cambiamos a la nueva asignacion de clusters y los nuevos centroides
42      current_cluster_indixes = new_cluster_indixes
43      current_centroids = new_centroids
44
45
46      # En caso de que robust = true, acotamos el numero de iteraciones de forma
47      # efectiva aumentando el contador. En otro caso, al no tocar el contador
48      # no estamos teniendo en cuenta este parametro
49      if robust == true:
50          curr_iteration = curr_iteration + 1;
51
52  # Devolvemos la solucion en la estructura de datos correspondiente
53  return solution_from(current_cluster_indixes)
```

Desarrollamos el código de assign_points_to_clusters por su importancia:

```
1  def assign_points_to_clusters():
2      # Realizamos una nueva asignacion de clusters
3      # -1 para saber que puntos todavia no han sido asignados a un cluster
4      new_cluster_indixes= [-1, -1, ..., -1]
5
6      # Recorremos aleatoriamente los puntos para irlos asignando a cada cluster
7      point_indexes = (0..data_points.len())
```

```
    point_indexes.shuffle();

    for index in point_indexes:
        # Calculo el cluster al que asignamos el punto actual
        new_cluster_indixes[index] = select_best_cluster(
            current_cluster_indixes,
            current_centroids,
        )

    # Devolvemos los indices que representan la solucion
    return new_cluster_indixes
```

5. Explicación del procedimiento considerado para desarrollar la práctica

Hemos desarrollado todo el código desde prácticamente cero usando el lenguaje de programación Rust. Para el entorno de desarrollo, solo hace falta instalar Cargo, que permite compilar el proyecto, correr los ejecutables cómodamente, descargar las dependencias o correr los tests que se han desarrollado.

Las librerías externas que hemos empleado pueden consultarse en el fichero Cargo.toml en el que se incluye un pequeño comentario sobre la utilidad de cada librería.

A continuación describimos el proceso de instalación y un pequeño manual de usuario para compilar y ejecutar el proyecto en Linux (pues es el sistema operativo que nuestro profesor de prácticas, Daniel Molina, nos ha indicado que usa).

5.1. Instalación del entorno

Lo más sencillo y directo es instalar rustup [1]que se encargará de instalar todos los elementos necesarios para compilar y ejecutar la práctica. Para ello podemos ejecutar:

- En cualquier distribución linux: curl ––proto '=https' ––tlsv1.2 –sSf https://sh.rustup.rs | sh y seguir las instrucciones
- Por si acaso no tenemos actualizado el entorno: rustup update

Una vez tengamos instalado rustup, podemos ejecutar órdenes como cargo check, rustc, ...

[1]Información actualizada sobre el proceso de instalación puede consultarse en [1], [2] o [3]

5.2. Compilación y ejecución del binario

Además del binario aportado en la estructura de directios que se especifica en el guión, podemos generar fácilmente el binario con cargo y ejecutarlo tal que:

- Para compilar: cargo build ––release lo que nos generará un binario en ./target/release/PracticasMetaheuristicas
- Podemos usar ese binario para ejecutar el programa o podemos usar cargo run ––release <parametros entrada> para correr el programa de forma más cómoda
- Es muy importante la opción ––release porque de otra forma el binario no será apenas optimizado, lo que supondrá tiempos de ejecución mucho mayores
- Para correr los tests programados podemos hacer cargo test

Si ejecutamos el binario sin parámetros, veremos por pantalla un mensaje indicando los parámetros que debemos pasar al programa. Estos parámetros son:

- Fichero de datos: el fichero donde guardamos las coordenadas de los puntos
- Fichero de restricciones
- Semilla para la generación de números aleatorios
- Número de clusters en los que queremos clasificar los datos
- Tipo de búsqueda: para especificar el tipo de algoritmo que queremos ejecutar. Los posibles valores son:
  - copkmeans: búsqueda greedy
  - copkmeans_robust: búsqueda greedy con los cambios ya indicados para que sea algo más robusto
  - local_search: búsqueda local

5.3. Compilación y ejecución usando el script

El proceso de compilación y ejecución del programa, sobre los distintos conjuntos de datos y restricciones, usando las distintas semillas que más adelante se especifican, se puede lanzar de forma cómoda invocando el script ./launch_all_programs.

6. Experimentos y análisis realizados

6.1. Descripción de los casos del problema empleados

ara los cinco **valores de las semillas**, hemos elegido los siguientes: 123456789, 234567891, 3456789, 456789123 y 567891234, sin ningún criterio en concreto (podríamos haber buscado, por ejemplo, semillas que no diesen problemas a la hora de lanzar el algoritmo greedy).

Tenemos tres problemas distintos. Cada problema, tiene dos ficheros de restricciones, uno con el 10 % de los datos con restricciones, y otro con el 20 % de los datos con restricciones. Los problemas son:

- Zoo: problema de agrupación de animales. Debemos clasificar datos de 16 dimensiones (o atributos) en 7 clusters distintos. Hay 101 instancias o puntos.
- Glass: problema de agrupación de vidrios. Debemos clasificar datos de 9 dimensiones en 7 clusters. Hay 214 instancias de datos
- Bupa: agrupar personas en función de hábitos de consumo de alcohol. Datos de dimensión 9 agrupados en 16 clusters. Hay 345 instancias de datos

Por tanto, haciendo cuentas, tenemos $3*2*5 = 30$ ejecuciones de búsquedas. De nuevo, estas 30 ejecuciones se pueden lanzar cómodamente usando ./launch_all_programs

Otros parámetros que hemos introducido en el programa son:

- Máximo de iteraciones de la búsqueda local: 100.000
- Máximo de iteraciones del algoritmo greedy cuando forzamos la robustez: 50
- Máximo de repeticiones del algoritmo greedy cuando este deja clusters vacíos: 100

6.2. Resultados obtenidos según el formato especificado

El formato especificado viene dado en una tabla de excel que hemos rellenado con los resultados mostrados por pantalla por el programa. Para obtener estos resultados, basta con lanzar ./launch_all_programs > salida_datos.txt y consultar el fichero creado. Ese fichero se entrega con la práctica para que pueda consultarse los datos con los que hemos generado las tablas.

6.2.1. Resultados Greedy

Tabla 6.1: Resultados obtenidos por el algoritmo Greedy en el PAR con 10% de restricciones

	Zoo				Glass				Bupa			
	Infeasable	*Error_Dist*	*Agr.*	T	*Infeasable*	*Error_Dist*	*Agr.*	T	*Infeasable*	*Error_Dist*	*Agr.*	T
Ejecución 1	23.00	4.4E-02	1.1E+00	0.21	4.00	1.0E-02	3.8E-01	0.93	0.00	4.0E-03	2.5E-01	7.88
Ejecución 2	12.00	6.5E-02	1.1E+00	0.21	1.00	-1.7E-02	3.5E-01	0.92	86.00	1.1E-02	2.5E-01	5.99
Ejecución 3	43.00	1.2E-01	1.4E+00	0.21	22.00	1.1E-02	4.0E-01	0.93	53.00	3.3E-03	2.4E-01	5.23
Ejecución 4	26.00	1.7E-01	1.3E+00	0.21	43.00	1.7E-02	4.2E-01	0.93	166.00	1.9E-02	2.8E-01	7.99
Ejecuión 5	4.00	5.4E-02	9.9E-01	0.24	10.00	2.7E-02	4.0E-01	0.93	91.00	1.2E-02	2.6E-01	7.13
Media	21.60	9.1E-02	1.2E+00	0.22	16.00	9.7E-03	3.9E-01	0.93	79.20	9.8E-03	2.6E-01	6.84

Tabla 6.1: Resultados obtenidos por el algoritmo Greedy en el PAR con 20% de restricciones

	Zoo				Glass				Bupa			
	Infeasable	*Error_Dist*	*Agr.*	T	*Infeasable*	*Error_Dist*	*Agr.*	T	*Infeasable*	*Error_Dist*	*Agr.*	T
Ejecución 1	6.00	1.0E-01	1.0E+00	0.21	0.00	-1.5E-02	3.5E-01	0.21	Sin dato	Sin dato	Sin dato	Sin dato
Ejecución 2	11.00	3.8E-02	9.9E-01	0.22	1.00	-1.7E-02	3.5E-01	0.14	Sin dato	Sin dato	Sin dato	Sin dato
Ejecución 3	3.00	1.1E-01	1.0E+00	0.22	21.00	2.2E-02	4.0E-01	0.96	Sin dato	Sin dato	Sin dato	Sin dato
Ejecución 4	5.00	8.9E-02	1.0E+00	0.22	1.00	3.5E-04	3.7E-01	0.29	Sin dato	Sin dato	Sin dato	Sin dato
Ejecuión 5	12.00	1.3E-01	1.1E+00	0.23	0.00	-6.3E-02	3.0E-01	0.17	Sin dato	Sin dato	Sin dato	Sin dato
Media	7.40	9.3E-02	1.0E+00	0.22	4.60	-1.5E-02	3.5E-01	0.35	Sin dato	Sin dato	Sin dato	Sin dato

6.2.2. Resultados Búsqueda local

Tabla 6.1: Resultados obtenidos por el algoritmo BL en el PAR con 10% de restricciones

	Zoo				Glass				Bupa			
	Infeasable	*Error_Dist*	*Agr.*	T	*Infeasable*	*Error_Dist*	*Agr.*	T	*Infeasable*	*Error_Dist*	*Agr.*	T
Ejecución 1	10.00	-3.0E-01	6.8E-01	0.12	11.00	-1.2E-01	2.6E-01	0.57	92.00	-1.0E-01	1.4E-01	5.68
Ejecución 2	16.00	-2.9E-01	7.4E-01	0.12	38.00	-1.5E-01	2.6E-01	0.73	114.00	-1.1E-01	1.4E-01	4.53
Ejecución 3	8.00	-2.1E-01	7.6E-01	0.06	37.00	-1.5E-01	2.5E-01	0.47	114.00	-1.0E-01	1.5E-01	13.27
Ejecución 4	9.00	-1.8E-01	7.9E-01	0.09	40.00	-1.5E-01	2.5E-01	0.40	104.00	-1.0E-01	1.5E-01	6.88
Ejecuión 5	7.00	-1.5E-01	8.1E-01	0.08	27.00	-1.2E-01	2.7E-01	0.42	88.00	-1.1E-01	1.3E-01	8.22
Media	10.00	-2.2E-01	7.6E-01	0.09	30.60	-1.4E-01	2.6E-01	0.52	102.40	-1.1E-01	1.4E-01	7.72

Tabla 6.1: Resultados obtenidos por el algoritmo BL en el PAR con 20% de restricciones

	Zoo				Glass				Bupa			
	Infeasable	*Error_Dist*	*Agr.*	T	*Infeasable*	*Error_Dist*	*Agr.*	T	*Infeasable*	*Error_Dist*	*Agr.*	T
Ejecución 1	30.00	-1.9E-01	8.4E-01	0.06	40.00	-1.1E-01	2.7E-01	0.74	251.00	-1.1E-01	1.5E-01	6.07
Ejecución 2	26.00	-1.9E-01	8.2E-01	0.08	31.00	-1.2E-01	2.6E-01	0.37	221.00	-1.0E-01	1.5E-01	6.82
Ejecución 3	54.00	-2.8E-01	8.5E-01	0.07	41.00	-1.1E-01	2.7E-01	0.56	204.00	-1.0E-01	1.4E-01	6.13
Ejecución 4	43.00	-1.8E-01	9.0E-01	0.08	35.00	-1.2E-01	2.7E-01	0.66	219.00	-1.1E-01	1.4E-01	6.51
Ejecuión 5	26.00	-1.9E-01	8.2E-01	0.06	44.00	-1.1E-01	2.7E-01	0.48	168.00	-1.0E-01	1.4E-01	7.61
Media	35.80	-2.0E-01	8.5E-01	0.07	38.20	-1.1E-01	2.7E-01	0.56	212.60	-1.0E-01	1.5E-01	6.63

6.2.3. Comparación de resultados

Tabla 6.2: Resultados globales en el PAR con 10% de restricciones

	Zoo				Glass				Bupa			
	Infeasable	***Error_Dist***	***Agr.***	**T**	***Infeasable***	***Error_Dist***	***Agr.***	**T**	***Infeasable***	***Error_Dist***	***Agr.***	**T**
COPKM	21.60	0.09	1.16	0.22	16.00	0.01	0.39	0.93	79.20	0.01	0.26	6.84
BL	10.00	-2.2E-01	7.6E-01	0.09	30.60	-1.4E-01	2.6E-01	0.52	102.40	-1.1E-01	1.4E-01	7.72

Tabla 6.2: Resultados globales en el PAR con 20% de restricciones

	Zoo				Glass				Bupa			
	Infeasable	***Error_Dist***	***Agr.***	**T**	***Infeasable***	***Error_Dist***	***Agr.***	**T**	***Infeasable***	***Error_Dist***	***Agr.***	**T**
COPKM	7.40	0.09	1.03	0.22	4.60	-0.01	0.35	0.35	Sin dato	Sin dato	Sin dato	Sin dato
BL	35.80	-2.0E-01	8.5E-01	0.07	38.20	-1.1E-01	2.7E-01	0.56	212.60	-1.0E-01	1.5E-01	6.63

6.3. Análisis de resultados

Las comparaciones más directas se pueden hacer con las dos últimas tablas en las que comparamos valores medios. Lo primero que notamos es que el algoritmo greedy, sobre el *dataset* de *Bupa* con el 20 % de los datos, no ha sido capaz de encontrar solución con ninguna de las semillas. Es decir, en todos los casos ha necesitado más de 100 repeticiones del algoritmo desde cero. Podríamos haber decidido devolver la última solución válida obtenida en todos los procesos, pero esto prácticamente es lo mismo que devolver una solución aleatoria que sea válida. Por tanto, consideramos que es más representativo dejar la marca de que el algoritmo no ha conseguido obtener una solución.

En segundo lugar, es de esperar que el algoritmo greedy sea más rápido que el de búsqueda local. Sin embargo hay conjuntos de datos y restricciones en los que esto no pasa (por ejemplo, Zoo con 10 % y 20 % y Glass al 10 %). Esto puede deberse a que la búsqueda local caiga demasiado rápido en un óptimo local a partir de la solución aleatoria de partida. También influye que estemos usando la versión robusta del algoritmo greedy en la que acotamos por 50 iteraciones, pudiendo haber probado una cota menor a vista de las iteraciones requerridas en otros casos del problema en los que greedy converge sin ayudas.

En tercer lugar, también cabe esperar que greedy produzca soluciones con menores violaciones de restricciones, pues es el primer criterio voraz que estamos considerando (y en caso de empate según violaciones incumplidas, ya se considera el cluster con centroide más cercano). Esto se cumple en todos los *datasets* salvo en Zoo al 10 %. Puede ser que al ser un espacio de

búsqueda más pequeño, la búsqueda local consiga encontrar un buen óptimo que respete muchas más restricciones manteniendo una buena desviación general de la partición.

Además, es claro que la búsqueda local genera soluciones cuya desviación general de cluster está por debajo de los valores de referencia. Esto gracias a que estamos considerando a la vez en la función de fitness el número de restricciones violadas (ponderado por el valor de λ) y la desviación general del cluster. Del mismo modo, estamos obteniendo mejores valores del fitness en prácticamente todos los casos. Esto era de esperar pues en búsqueda local estamos optimizando este valor mientras que en el algoritmo greedy estamos simplemente siguiendo una estrategia voraz que pensamos que puede producir buenos resultados. Cuando no ocurre esto, podríamos pensar que el algoritmo greedy ha obtenido una muy buena solución por casualidad, o lo que parece más probable, que nuestra búsqueda local cae en un mal óptimo local en contadas ocasiones.

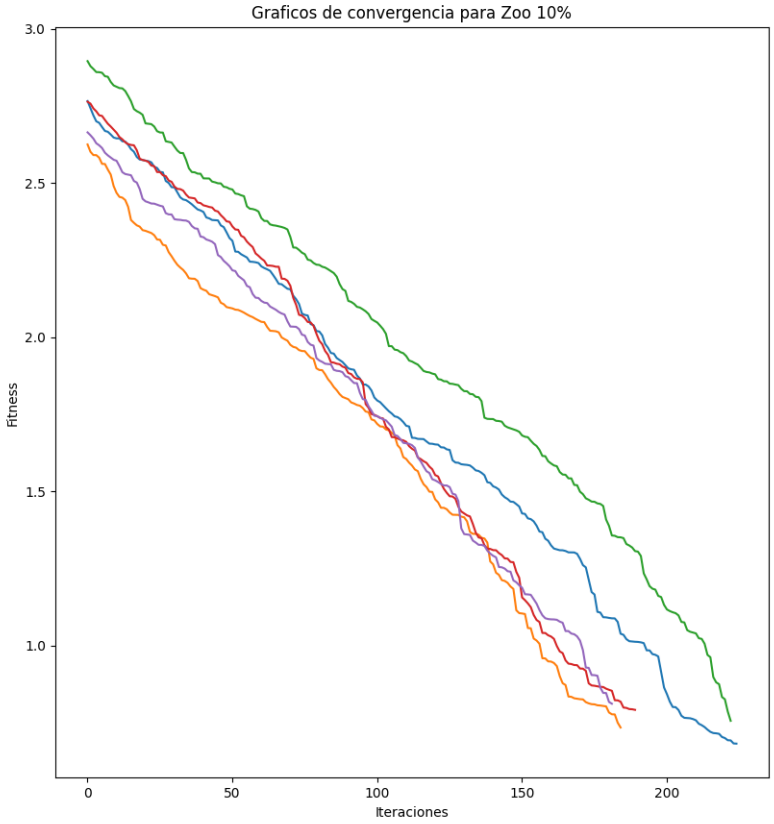
Pasamos ahora a comentar las tablas generales, no solo las de comparaciones. Podríamos hacer comentarios sobre el comportamiento del algoritmo greedy, como por ejemplo, su dependencia de la semilla inicial a la hora de obtener buenos o malos resultados, pero al ser un algoritmo de comparación base, no parece relevante que hagamos un profundo estudio sobre su comportamiento.

El valor del fitness se ve afectado por la semilla inicial (que determina la solución inicial de la que partimos), pero no hay variaciones demasiado bruscas (como si ocurre, por ejemplo, en el greedy con los Zoo de 10 % y 20 % de restricciones). Sin embargo, si que podemos observar una variabilidad mucho más grande en la cantidad de restricciones violadas. En zoo 10 % podemos pasar de 7 violaciones a 16 (más del doble). En zoo 20 % podemos pasar de 26 a 54 (de nuevo, un poco más del doble). En bupa 20 %, podemos pasar de 168 a 251. Por tanto podemos pensar que es esta métrica es más sensible a la aleatoriedad que el fitness (en el que esta métrica compone el primer sumando de forma ponderada). Por ejemplo, para Zoo 10 %, las restricciones violadas tienen un 3.54 de desviación típica, mientras que el fitness tiene un 0.05 de desviación típica.

6.3.1. Gráficas de evolución de fitness

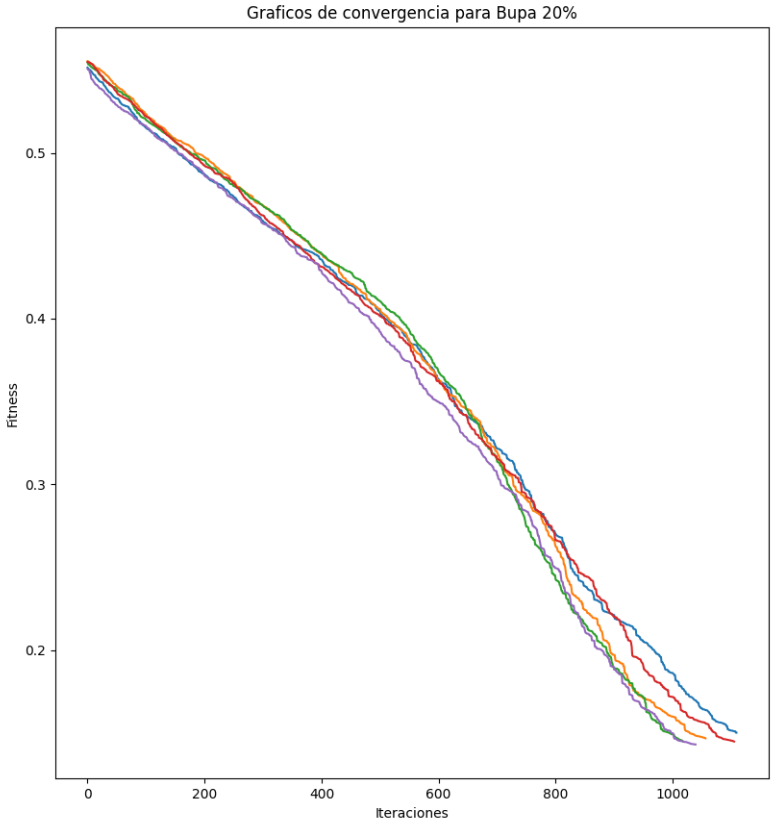
Estas gráficas han sido generadas a partir de la salida del programa y un script de Python que emplea la librería matplotlib para la generación de las gráficas. El código de generación de las gráficas se puede consultar en el fichero graficas.py.

Podemos mirar las gráficas de evolución del error (o gráficas de convergencia) para Zoo 10 %:



Con esta gráfica queda claro el impacto que tiene la semilla inicial, tanto para la calidad de la solución obtenida, como para el número de iteraciones

necesarias para conseguir alcanzar un óptimo local. Podemos mostrar la misma gráfica para Bupa 20 %:



Esta gráfica refuerza de nuevo todo lo que venimos desarrollando en relación a la influencia de la semilla aleatoria. Ambas gráficas también dejan claro que la evolución del fitness es estrictamente decreciente. Esto porque estamos haciendo una búsqueda el primero mejor, por tanto, siempre nos movemos hacia soluciones de mejor fitness que la solución actual.

6.3.2. Posibles mejoras

Para comenzar, sería interesante realizar un estudio sobre el valor de λ, buscando métodos o valores para que produzca soluciones con un mejor valor del $fitness$. En segundo lugar, hemos visto la sensibilidad de la búsqueda local a la semilla aleatoria. Esto puede movitar el interés por otros métodos como el de realizar búsquedas múltiples partiendo de distintas soluciones iniciales.

7. Bibliografía

Referencias

[1] "Getting started - rust programming language." `https://www.rust-lang.org/learn/get-started`. (Accessed on 04/11/2021).

[2] "Install rust - rust programming language." `https://www.rust-lang.org/tools/install`. (Accessed on 11/04/2021).

[3] "rustup.rs - the rust toolchain installer." `https://rustup.rs/`. (Accessed on 11/04/2021).