Práctica 3b
Búsqueda por Trayectorias
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

6 de junio de 2021

Índice

Índice de figuras

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.

Cuando trabajemos con algoritmos poblacionales, usaremos la siguiente **nomenclatura**:

- Población: conjunto de soluciones
- Cromosoma: una solución individual
- Gen: cada uno de los elementos del vector de asignación punto → cluster que compone la solución

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.

Además, en todos los algoritmos, salvo copkmeans, debemos llevar la cuenta de las evaluaciones de *fitness* que consumimos. Para ello tenemos la función fitness_and_consumed. También tenemos funciones para invalidar la cache, para comprobar si tenemos el *fitness* cacheado o no, ...

2.3.2. FitnessEvaluationResult

Para mejorar el control sobre las evaluaciones del fitness, disponemos del struct FitnessEvaluationResult <T>. Guarda un tipo genérico T y las evaluaciones que consume la operación que genera dicho valor. Por ejemplo, si en un población (de la que hablaremos más adelante), queremos encontrar el individuo con mejor valor de fitness, devolvemos FitnessEvaluationResult<Solution> con dicho mejor individuo y las consumiciones de fitness **efectivas** que consume esta búsqueda.

2.4. Representación de la población

Una población, intuitivamente, es un conjunto de individuos, que en nuestro caso, serán soluciones válidas del problema que estamos intentando resolver. En código la representaremos como un vector de Solution. Sobre este struct podemos realizar operaciones comunes a los algoritmos genéticos, tanto generacionales como estacionarios, y a los algoritmos meméticos. Operaciones como generar una población inicial aleatoria, mutar con cierta probabilidad a la población, realizar torneos binarios para generar una población de selección de tamaño dado, ...

2.4.1. Representación en código y consideraciones

El struct viene dado por:

```rust
/// Representa una poblacion para los algoritmos geneticos
#[derive(Debug, Clone)]
pub struct Population<'a, 'b>{
    /// Individuos de la poblacion
    individuals: Vec<Solution<'a, 'b> >,
}
```

Los métodos implementados para el struct serán comentados a medida que vayamos describiendo el pseudocódigo de los distintos algoritmos.

Una consideración importante es que, en un primer momento, consideramos utilizar una estructura de datos del tipo *Cola con prioridad*. De esta forma, podríamos obtener de forma eficiente los elementos mejores y peores (respecto a su valor del *fitness*) de una población. Sin embargo, introducía mucha complejidad en el código, pues era difícil tener en cuenta las evaluaciones del fitness que se consumían al mantener la estructura de datos (por ejemplo, deberíamos haber implementado la interfaz *Ord*, que no permite devolver el tipo de dato *FitnessEvaluationResult*). Esto, junto a que el código corre en unos tiempos muy razonables, motiva nuestra decisión a no considerar una estructura de datos más compleja. Podría haber sido interesante implementar a mano un PriorityQueue para poder controlar las evaluaciones del fitness y también optimizar algo más el código.

En los pseudocódigos mostramos que, en la función select_best_indixes , usamos una cola con prioridad. Esto porque previamente evaluamos toda la población, y con ello, tenemos controladas las evaluaciones del fitness.

Notar también que muchos de los métodos devuelven el tipo FitnessEvaluationResult, para tener control de las evaluaciones del fitness consumidas, como ya hemos comentado previamente.

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,

    # El nuevo cluster al que asignamos el elemento
    new_cluster: u32,

# Funcion que genera todos los vecinos posibles de un elemento
# Los vecinos generados pueden ser no validos
def generate_all_neighbours(
    number_of_elements,
    number_of_clusters):
    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     }
```

3.2. Descripción y Pseudocódigo del algoritmos de comparación - *Copk-means*

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:

```
1 # 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()

    # Comprobamos que la nueva solucion calculada es correcta
    if valid_cluster_configuration(current_cluster_indixes) == false:
        # Esto hace que el el caller de la funcion copkmeans, se muestre un
        # mensaje por pantalla y se vuelva a realizar la busqueda, con lo que
        # partimos de unos centroides aleatorios nuevos. Como ya se ha comentado,
        # hay un maximo de reinicios en el caller para este metodo
        return None

    # Calculamos los nuevos centroides y comprobamos si han cambiado
    new_centroids = calculate_new_centroids(new_cluster_indixes)
    centroids_have_changed = centroids_are_different(
        current_centroids,
        new_centroids
    )

```

```
    # Cambiamos a la nueva asignacion de clusters y los nuevos centroides
    current_cluster_indixes = new_cluster_indixes
    current_centroids = new_centroids


    # En caso de que robust = true, acotamos el numero de iteraciones de forma
    # efectiva aumentando el contador. En otro caso, al no tocar el contador
    # no estamos teniendo en cuenta este parametro
    if robust == true:
        curr_iteration = curr_iteration + 1;

# Devolvemos la solucion en la estructura de datos correspondiente
return solution_from(current_cluster_indixes)
```

Desarrollamos el código de assign_points_to_clusters por su importancia:

```
def assign_points_to_clusters():
    # Realizamos una nueva asignacion de clusters
    # -1 para saber que puntos todavia no han sido asignados a un cluster
    new_cluster_indixes= [-1, -1, ..., -1]

    # Recorremos aleatoriamente los puntos para irlos asignando a cada cluster
    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
```

3.3. Descripción y pseudocódigo del algoritmo de búsqueda local multiarranque básica

Este algoritmo consiste en lanzar, con distintas soluciones iniciales aleatorias, la búsqueda local fuerte implementada en la práctica 1. Tras lanzar el número dado de veces esta búsqueda local, nos quedamos con la solución que mejor *fitness* ha alcanzado. En *3.1. Búsqueda Local*, se detalla el proceso de generación de una solución aleatoria, que se usa en esta metaheurística. El pseudocódigo de este algoritmo, por tanto, es el siguiente:

```
1
2  # Maximo de evaluaciones del fitness
3  max_fitness_evaluations = 10000
4
5  # Numero de veces que realizamos la busqueda local
6  number_of_local_searchs = 10
7
8  # Vectores para guardar los resultados de las busquedas locales
9  solutions = []
10 fitness_evolutions = []
11
12 # Lanzamos las busquedas locales
13 for i in 0..number_of_local_searchs:
14     solucion_local, fitness_evolution= local_search::run(
       max_fitness_evaluations)
15     solutions.append(solucion_local)
16     fitness_evolutions.append(fitness_evolution)
17
18 # Indice de la solucion con mejor fitness
19 best_index = select_best_solution(solutions)
20
21 # Recuperamos la mejor solucion y la mejor evolucion del fitness
22 best_solution = solutions[best_index]
23 best_fit_ev = fitness_evolutions[best_index]
24
25 def select_best_solution(solutions):
26     best_index = 0
27     best_fitness = solutions[best_index].fitness()
28
29     for index in 0..solutions.len():
30         if solutions[index].fitness() < best_fitness:
31             best_fitness = solutions[index].fitness()
32             best_index = index
33
34     return best_index
```

Notar también que el valor de los dos parámetros fundamentales (número de repeticiones de la búsqueda local y número de evaluaciones del fitness máximo por búsqueda local) queda especificado en *5.1.2. Parámetros de la búsqueda local multiarranque básica*. Se verifica que el producto de repeticiones de la búsqueda local por el número de evaluaciones del fitness asignadas a cada búsqueda local es igual al número máximo de evaluaciones del fitness asignados a los algoritmos que hemos empleado en prácticas anteriores, para poder realizar comparaciones.

3.4. Descripción y pseudocódigo del algoritmo *Iterative Local Search*

En este algoritmo, realizaremos una búsqueda local sobre una solución inicial aleatoria. Una vez hecho esto, mutamos fuertemente la solución con la mutación por segmento fijo(especificado en la función hard_mutated) y aplicamos búsqueda local o enfriamiento simulado como intensificación a esta solución mutada. Si esta nueva solución es mejor que la original, se sustituye. Se repite este proceso de mutación de la mejor solución, intensificación de la mutación y reemplazo en caso de obtener mejores resultados un número dado de veces. El pseudocódigo queda tal que:

```
# Numero maximo de repeticiones y maximo de evaluaciones del fitness
# en el algoritmo intensificador
max_fitness_evaluations = 10000
number_of_repetitions = 10

# Tamaño del segmento de mutacion fuerte que consideramos
mutation_segment_size = 0.1 * data_points.len()

# Generamos una solucion inicial aleatoria
# Current solution sera la mejor solucion hasta el momento
current_solution = Solution::generate_random_solution(data_points, constraints, number_of_clusters, rng);

# Realizamos las repeticiones dadas
for _ in 0..number_of_repetitions:

    # Mutamos fuertemente la mejor solucion encontrada hasta el momento
    new_solution = current_solution.hard_mutated(mutation_segment_size)

    # Aplicamos busqueda local o enfriamiento simulado a esta solucion mutada fuertemente
    # En el caso de enfriamiento simulado, establecemos los mismos parametros que en este
    # algoritmo
    local_solution = intensificar(max_fitness_evaluations, new_solution)

    # Comprobamos si esta solucion es mejor que la que ya teniamos
    if new_solution.fitness() < current_solution.fitness():
        current_solution = new_solution;

return (current_solution, fitness_evolution);

# Funcion de mutacion por segmento fijo
def hard_mutated(self, segment_size):
    # Copiamos para no modificar la solucion actual
    mutated = self.clone()

    # Seleccionamos el inicio del segmento
    let gen_size = self.cluster_indexes.len();
```

```
37      let segment_start = rng.gen_range(0..gen_size);
38
39      # Mutamos los valores el el segmento. El resto de valores son
        automaticamente copiados del
40      # padre porque mutated es clone de self
41      for i in 0..segment_size:
42          # Indice que debemos mutar segun los valores del segmento
43          index = (segment_start + i)  % segment_size
44
45          # Mutamos dicho valor
46          let new_cluster = rng.gen_range(0..mutated.
        number_of_clusters)
47          mutated.cluster_indexes[index] = new_cluster as u32;
48      }
49
50      # Reparamos la solucion si la solucion mutada acaba por no ser
        valida
51      if mutated.is_valid() == false:
52          mutated.repair_solution(rng)
53
54      return mutated;
```

Los parámetros fundamentales, como el número de repeticiones, máximo de evaluaciones del fitness en el algoritmo intensificador o tamaño del segmento para la mutación, se especifica en *5.1.3. Parámetros de la búsqueda local iterativa.*

3.5. Descripción y pseudocódigo del algoritmo Enfriamiento Simulado

El enfriamiento simulado se parece a la búsqueda local. Partimos de una solución aleatoria. El proceso de búsqueda consiste en ir generando soluciones en el vecindario de la solución actual. Sin embargo, mientras que en búsqueda local solo aceptamos soluciones que mejoren a la actual, en enfriamiento simulado existe la posibilidad de escoger vecinos peores, con la esperanza de que esto permita escapar de malos óptimos locales.

Cuando un vecino es mejor, siempre hacemos el cambio. Cuando un vecino es peor, podemos aceptarlo siguiendo una probabilidad que depende de la temperatura actual. Esta condición probabilística viene dada por $U(0,1) \leq e^{\frac{-\Delta f}{k*T}}$, donde U es la distribución continua uniforme, k es una constante física que ignoramos haciendo $k = 1$, y T es la temperatura actual.

Para seguir el esquema diversificación al principio, intensificación al final, partimos de una temperatura inicial relativamente alta, que poco a poco va tendiendo a cero (en nuestro caso, seguimos el esquema de enfriamiento de Cauchy). En el pseudocódigo se mostrará el esquema de enfriamiento, y en *5.1.4. Parámetros del enfriamiento simulado* se mostrarán los valores iniciales.

Además, guardamos la mejor solución que se ha encontrado hasta el momento. A diferencia de la búsqueda local, no generamos todos los posibles vecinos en formato (*pos_cambiar*, *nuevo_valor*), sino que generamos un vecino aleatorio. Por tanto, algunos vecinos pueden repetirse.

En el pseudocódigo queda claro que tenemos otra condición de parada: cuando el bucle interno produce cero éxitos, tras generar un número dado de vecinos.

Así, el pseudocódigo queda:

```
55 # Parametros iniciales del algoritmo
56 max_fitness_evaluations = 100000
57 mu = 0.3
58 final_tmp = 0.001
59 max_neighbours = 10.0 * data_points.len()
60 max_successes = 0.1 * max_neighbours
61 M = max_fitness_evaluations / max_neighbours
62
63 # Necesitamos la solucion inicial para establecer la temperatura
      inicial
64 init_solution = Solution::generate_random_solution()
65
66 # Con ello, computamos la temperatura inicial
67 # Usamos que en este problema, mu = phi, asi que solo usamos una
      variable
68 initial_tmp = (mu * init_solution.fitness()) / (-mu.ln())
69
70 # Valores iniciales para empezar a iterar
71 current_evaluations = 0
72 current_tmp = initial_tmp
73 current_solution = init_sol.clone()
74 best_solution = current_solution.clone()
75 best_fitness = best_solution.fitness()
76
```

```
77 # Necesitamos el valor de beta para el enfriamiento
78 beta (initial_tmp - final_tmp) / (M * initial_tmp * final_tmp)
79
80 # Bucle externo
81 while current_evaluations < max_fitness_evaluations && current_tmp
       >= final_tmp:
82
83     # Bucle interno
84     # Solo generamos max_neighbours a lo sumo. Tambien paramos
       cuando se ha alcanzado un
85     # numero maximo de exitos
86     current_successes = 0
87     for _ in 0..max_neighbours:
88         current_neighbour = current_solution.one_random_neighbour(
       rng);
89
90         # Calculamos el delta del fitness
91         current_solution_fitness = current_solution.fitness()
92         current_neighbour_fitness = current_neighbour.fitness()
93         delta_fitness = current_solution_fitness -
       current_neighbour_fitness
94
95         # Aceptamos el vecino si es mejor o con la probabilidad
96         # que depende de la temperatura
97         if delta_fitness < 0.0 && rng.gen::<f64>() > (-
       delta_fitness * current_tmp):
98             # La solucion es peor y ha fallado la probabilidad
99             continue
100
101
102         # Hemos aceptado la solucion
103         current_solution = current_neighbour.clone()
104         current_successes += 1
105
106         # Comprobamos si hemos mejorado a la mejor solucion
107         if current_solution_fitness < best_fitness:
108             best_fitness = current_solution_fitness
109             best_solution = current_solution.clone()
110
111
112         # Si hemos alcanzado el numero maximo de exitos, salimos
       del bucle interno
113         if current_successes >= max_successes:
114             break
115
116         # Si hemos consumido las evaluaciones maximas en el bucle
       interno, debemos salir
117         if current_evaluations >= max_fitness_evaluations:
118             break
119
120     # Computamos el siguiente valor de la temperatura
```

```
121     current_tmp = current_tmp / (1.0 + beta * current_tmp);
122 
123     # Si se obtuvieron 0 exitos en el bucle interno, paramos de
    iterar
124     if current_successes == 0:
125         break
126 
127 return best_solution
128 
129 
130 def one_random_neighbour(self):
131     # Usamos la funcion de mutacion para realizar el cambio
132     let mutated = self.mutated(rng)
133 
134     # Si hay mas de una diferencia, es porque el operador de
    reparacion ha reparado provocando
135     # mas cambios. En este caso, esto no es lo que queremos
136     if mutated.number_of_discrepancies(self) != 1:
137         return self.mutated(rng);
138 
139     return mutated;
```

4. 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).

4.1. 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, ...

4.2. Compilación y ejecución del binario

Además del binario aportado en la estructura de directorios 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. Además todas las comprobaciones con debug_assert! no serán ignorados, lo que relentecerá muchísimo las ejecuciones
- 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:

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

- 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
  - gguniform
  - ggsegment
  - gsuniform
  - gssegment
  - memeall
  - memerandom
  - memeelitist
  - multistartlocalsearch
  - iterative_local_search
  - iterative_local_search_annealing
  - simulated_annealing

4.3. 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.

5. Experimentos y análisis realizados

5.1. Descripción de los casos del problema empleados

Para 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

Podemos lanzar las $5*3*2 = 30$ ejecuciones por algoritmo cómodamente con launch_all_programs .

5.1.1. Parámetros de la búsqueda local y Greedy

- Máximo de evaluaciones de la función fitness en 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

5.1.2. Parámetros de la búsqueda local multiarranque básica

- Número de repeticiones de la búsqueda local: 10 repeticiones
- Máximo de evaluaciones del fitness por cada búsqueda local: 10000 evaluaciones

5.1.3. Parámetros de la búsqueda local iterativa

- Número de repeticiones de la búsqueda local / enfriamiento simulado: 10 repeticiones
- Máximo de evaluaciones del fitness por cada búsqueda local / enfriamiento simulado: 10000 evaluaciones
- Longitud del segmento en la mutación: $0{,}1 * n$ donde es el número de puntos (o la longitud del vector de asignación)

5.1.4. Parámetros del enfriamiento simulado

- Máximo de evaluaciones del fitness: 100000 evaluaciones
- Temperatura inicial: $T_0 = \frac{\mu * C(S_0)}{-ln(\phi)}$ donde $C(S_0)$ es el fitness de la solución inicial, y $\mu = \phi = 0{,}3$
- Temperatura final: $T_f = 10^{-3}$
- Máximo de vecinos generados en el bucle interno: $10 * n$ donde n es el tamaño del caso del problema
- Esquema de enfriamiento:
  - $T_{k+1} = \frac{T_k}{1+\beta T_k}$
  - $\beta = \frac{T_0 - T_f}{M * T_0 * T_f}$
  - $M = \frac{max\_iteraciones}{max\_vecinos}$

5.2. 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.

En esta práctica hemos cambiado ligeramente el formato de la tabla *Excel*, pero manteniendo el contenido y su estructura. Todo esto para hacer más sencillo el recopilar los datos en estas tablas tras al ejecución del script ./launch_all_programs. Esta recopilación se ha realizado en varias etapas usando *macros* de VIM. Estas fases pueden ser consultadas en los archivos salida .txt, convertida a salida .csv y refinada en salida_depurada.csv.

Los datos guardados en un formato especificado se encuentra en la tabla de *excel*. Esta tabla se encuentra colgando de la carpeta FUENTES/analisis/Segunda Memoria/. Los resultados de la ejecución en nuestro ordenador (en el ordenador de los profesores, se puede obtener una traza distinta a pesar de usar las mismas semillas) se encuentran en FUENTES/results. También tenemos ficheros *.csv* más depurados que he usado para cargar los datos en el *Excel*.

Considerar también que las ejecuciones provocan que se guarden ficheros tipo numpy en la carpeta FUENTES/fitness_evolution_data

5.3. Tablas con los resultados globales

Por la inmensa cantidad de datos, mostramos en esta memoria solamente las tablas de resultados globales. Notar que las tablas que dejamos fuera nos dan información adicional sobre como varían los algoritmos respecto al cambio de semilla. Información sobre estabilidad, desviaciones típicas, ...

Y además, como ya se ha comentado, en la carpeta FUENTES/results tenemos la traza de ejecución y, además, los ficheros .csv que hemos generado para tener información más depurada de los resultados.

Las tablas con los resultados son las siguientes:

Resultados globales en el PAR con 10% de restricciones

	Zoo				Glass				Bupa			
	Infeasable	*Distancia_Intra*	*Agr.*	T	*Infeasable*	*Distancia_Intra*	*Agr.*	T	*Infeasable*	*Distancia_Intra*	*Agr.*	T
COPKM	21.6	0.09	1.16	0.22	16	0.01	0.39	0.93	79.2	0.01	0.26	6.84
BL	10	0.680084933	0.75586	0.094328311	30.6	0.227662417	0.257774069	0.5	116	0.115242325	0.14633	5.900788302
Búsqueda Multiarranque	10.4	0.625113828	0.70392	0.960462896	48.2	0.202113598	0.249544371	3.56	446.4	0.141852424	0.26148	7.753531674
ILS - Local Search	12.4	0.66371837	0.75768	0.182339879	63.8	0.210087368	0.272869179	0.83	458.8	0.146026316	0.26898	4.504243213
ILS - Simulated Annealing	8.2	0.579451431	0.64159	0.963671273	39.8	0.210067735	0.249232564	2.81	123.2	0.111239019	0.14426	7.156380782
Enfriamiento Simulado	17.8	0.805727536	0.94061	0.211094227	48.8	0.202743959	0.250765157	0.52	105.4	0.120902129	0.14915	6.164183478

Figura 1: Tablas Globales - 10 % de restricciones

Resultados globales en el PAR con 20% de restricciones

	Zoo				Glass				Bupa			
	Infeasable	*Distancia_Intra*	*Agr.*	T	*Infeasable*	*Distancia_Intra*	*Agr.*	T	*Infeasable*	*Distancia_Intra*	*Agr.*	T
COPKM	7.4	0.09	1.03	0.22	4.6	0.01	0.35	0.35	2163	0.255368309	0.55698	0.003718075
BL	69.47142857	0.348900604	0.42071	2.644544083	53.39081633	0.389219617	0.458977195	1.75	212.6	0.357252727	0.42267	2.328629045
Búsqueda Multiarranque	21.6	0.694971751	0.78219	0.84453839	62.4	0.231206511	0.263692547	4.64	648.68	0.162239922	0.25745	9.646450653
ILS - Local Search	25.6	0.713150609	0.81653	0.222826753	52.6	0.250473059	0.277857121	1.15	52.6	0.250473059	0.27786	1.145811257
ILS - Simulated Annealing	22.8	0.642745126	0.73481	0.915192725	48.2	0.239914336	0.265007716	3.68	176.2	0.118877258	0.14345	10.18210313
Enfriamiento Simulado	24.6	0.762735844	0.86207	0.205368	65	0.242169271	0.276008891	0.75	249.8	0.119301216	0.15413	6.024784638

Figura 2: Tablas Globales - 20 % de restricciones

5.4. Análisis de resultados

5.4.1. Análisis general de los resultados

Para este análisis va a resultar muy útil tener en cuenta los algoritmos que mejor y peor resultado han dado en cada uno de los casos del problema:

Resultados globales en el PAR con 10% de restricciones

	Zoo				Glass				Bupa			
	Infeasable	*Distancia_Intra*	*Agr.*	T	*Infeasable*	*Distancia_Intra*	*Agr.*	T	*Infeasable*	*Distancia_Intra*	*Agr.*	T
COPKM	21.6	0.09	1.16	0.22	16	0.01	0.39	0.93	79.2	0.01	0.26	6.84
BL	10	0.680084933	0.75586	0.094328311	30.6	0.227662417	0.257774069	0.5	116	0.115242325	0.14633	5.900788302
Búsqueda Multiarranque	10.4	0.625113828	0.70392	0.960462896	48.2	0.202113598	0.249544371	3.56	446.4	0.141852424	0.26148	7.753531674
ILS - Local Search	12.4	0.66371837	0.75768	0.182339879	63.8	0.210087368	0.272869179	0.83	458.8	0.146026316	0.26898	4.504243213
ILS - Simulated Annealing	8.2	0.579451431	0.64159	0.963671273	39.8	0.210067735	0.249232564	2.81	123.2	0.111239019	0.14426	7.156380782
Enfriamiento Simulado	17.8	0.805727536	0.94061	0.211094227	48.8	0.202743959	0.250765157	0.52	105.4	0.120902129	0.14915	6.164183478

Figura 3: Mejor y peor algoritmo, 10 % de las restricciones

Resultados globales en el PAR con 20% de restricciones

	Zoo				Glass				Bupa			
	Infeasable	*Distancia_Intra*	*Agr.*	T	*Infeasable*	*Distancia_Intra*	*Agr.*	T	*Infeasable*	*Distancia_Intra*	*Agr.*	T
COPKM	7.4	0.09	1.03	0.22	4.6	0.01	0.35	0.35	2163	0.255368309	0.55698	0.003718075
BL	69.47142857	0.348900604	0.42071	2.644544083	53.39081633	0.389219617	0.458977195	1.75	212.6	0.357252727	0.42267	2.328629045
Búsqueda Multiarranque	21.6	0.694971751	0.78219	0.8453839	62.4	0.231206511	0.263692547	4.64	648.68	0.162239922	0.25745	9.646450653
ILS - Local Search	25.6	0.713150609	0.81653	0.222826753	52.6	0.250473059	0.277857121	1.15	52.6	0.250473059	0.27786	1.145811257
ILS - Simulated Annealing	22.8	0.642745126	0.73481	0.915192725	48.2	0.239914336	0.265007716	3.68	176.2	0.118877258	0.14345	10.18210313
Enfriamiento Simulado	24.6	0.762735844	0.86207	0.205368	65	0.242169271	0.276008891	0.75	249.8	0.119301216	0.15413	6.024784638

Figura 4: Mejor y peor algoritmo, 20 % de las restricciones

Como no hemos modificado ni corregido nada respecto de la práctica anterior, los resultados de la búsqueda local y búsqueda *copkmeans* son los mismos que en la práctica anterior.

Fijándonos en los resultados en los *datasets* con el 10 % de las restricciones, cabe destacar que el peor algoritmo siempre ha sido *copkmeans*. Por tanto, los nuevos algoritmos introducidos en esta práctica han tenido un comportamiento como mínimo decente sobre estos conjuntos con el 10 % de las restricciones. Recordar que en la práctica anterior, algunos de los algoritmos tenían un comportamiento nefasto según la configuración del problema, obteniendo resultados peores que *copkmeans*. Y en los tres problemas con el 10 % de los resultados, el mejor algoritmo ha sido *ILS-Enfriamiento Simulado*.

Esto puede ser porque, a pesar de que la intensificación usando enfriamiento simulado sea menos inteligente que la intensificación con búsqueda local, en muchas ocasiones sea útil salir de óptimos locales. Sobre todo en repeticiones tardías de la búsqueda iterativa, donde la intensificación por búsqueda local puede desperdiciar muchas evaluaciones del fitness al quedar atascada en óptimos locales demasiado rápido. Además, la búsqueda con enfriamiento acepta resultados peores con mayor probabilidad al principio, luego en iteraciones más tardías de la intensificación acabaremos con un comportamiento más o menos similar a la búsqueda local.

En los conjuntos con el 20 % de las restricciones tenemos algo más de variedad. Se sigue manteniendo el hecho de que en ninguno de los casos los nuevos algoritmos sean peores que *copkmeans*. Solo en *Glass* la búsqueda local es peor que *Copkmeans*. Y en ese caso, los cuatro algoritmos restantes siguen siendo mejores que *copkmeans*.

En el caso de *Zoo* el mejor algoritmo es la búsqueda local. Esto puede venir motivado porque, al no tener que repartir las evaluaciones entre 10 repeticiones distintas, tenemos 10 veces más iteraciones para intensificar. Con lo que, si no caemos en malos óptimos locales, es lógico pensar que vamos a encontrar mejores soluciones. Esta es la explicación mas sensata a lo que ha pasado

en este caso concreto.

La búsqueda local multiarranque ha sido la ganadora en el caso de *Glass*, seguida en segundo lugar de *ILS-Enfriamiento Simulado*. No es sorprendente, pues enfriamiento simulado es una búsqueda menos inteligente que la búsqueda local el primero mejor.

En el caso de *Bupa*, vuelve a ganar *ILS-Enfriamiento Simulado*, seguido por enfriamiento simulado. Con esta configuración de tamaño funcionan muy bien las técnicas basadas en el enfriamiento simulado, el permitir soluciones peores ha sido clave para este problema. Esto se puede ver también el *Bupa 10 %* donde hemos acabado en la misma situación que en este caso.

Todos los algoritmos han tenido comportamientos decentes. El único que no ha destacado en ningún momento ha sido la búsqueda local iterativa usando como intensificador la búsqueda local, obteniendo consistentemente resultados mediocres. El enfriamiento simulado tampoco ha sido el mejor nunca, pero en las dos variantes del problema *Bupa* ha tenido resultados muy buenos, parecidos a los que obtuvo *ILS-Enfriamiento Simulado*, cayendo en la segunda posición. Este comportamiento puede ser debido a lo que en parte ya hemos comentado. En repeticiones tardías de la búsqueda iterativa, a pesar de la fuerte mutación, la intensificación por búsqueda local puede caer en un mal óptimo demasiado rápido, desperdiciando muchas evaluaciones del fitness sin consumir en esa repetición.

Una mejora a este mal comportamiento de *ILS-Local Search* podría ser repetir hasta consumir las evaluaciones del fitness, no hasta un número prefijado de iteraciones. Con ello, si el problema de este algoritmo es efectivamente el que hemos comentado, tendría un menor impacto en el rendimiento del algoritmo.

Cabe destacar los buenos resultados que hemos obtenido con unos algoritmos que han sido muy rápidos de programar. Una vez que hemos programado la búsqueda local, el enfriamiento simulado y los operadores auxiliares, programar estos métodos ha sido muy sencillo, requiriendo muy poco esfuerzo de ingeniería. Por tanto, una vez hecha la base de código que ya hemos comentado, parece sensato siempre probar con estas variantes que son muy sencillas de introducir y mejoran los resultados. Y entre estos, el más destacable es la búsqueda local multiarranque, que prácticamente no ha supuesto esfuerzo de implementación.

En resumen de este primer análisis, el algoritmo que se ha impuesto sobre el resto ha sido *ILS-Enfriamiento simulado*. El resto de los algoritmos han tenido un comportamiento decente, siendo la búsqueda local iterativa con búsqueda local como intensificador el único con resultados mediocres.

Respecto a los tiempos de ejecución no hay demasiado que comentar. En media, el algoritmo que más ha tardado en ejecutarse ha tomado poco más de 10 segundos en *Bupa*. En los problemas con el 10 % de las restricciones, el más lento ha sido *ILS-Enfriamiento Simulado*. En los problemas con el 20 % de las restricciones, los más lentos han sido búsqueda local multiarranque e *ILS-Enfriamiento Simulado*. Sin embargo, no han diferido en distintos órdenes de magnitud, por lo tanto, no parece relevante realizar un análisis de estas diferencias de tiempo. Factores difíciles de capturar, como los cambios de contexto que realizamos en nuestro código, optimizaciones del compilador o fallos de caché pueden ser los causantes de estas diferencias, viendo lo parecidos que son los tiempos de ejecución.

5.4.2. Algunas comparaciones

La comparación más directa es la de búsqueda local con búsqueda local multiarranque. La búsqueda local tiene la bondad de que emplea todas las evaluaciones en un único proceso de búsqueda, por lo que, si no cae en óptimos locales de mala calidad, tiene 10 veces más evaluaciones del fitness para intensificar más la solución. Por el otro lado, la búsqueda local multiarranque tiene 10 veces menos evaluaciones del fitness, pero las 10 repeticiones debieran dar mucha más estabilidad y robustez.

En los conjuntos con 10 % de restricciones, solo en *Bupa* la búsqueda local es mejor. Esto tiene sentido, pues es un problema más grande en el que pueden ser necesarias más iteraciones de las que otorgamos a cada una de las búsquedas en la búsqueda multiarranque. En los conjuntos con el 20 % de las restricciones tan solo en *Zoo* la búsqueda local le gana a la búsqueda local multiarranque. De hecho, en *Glass* el peor algoritmo es la búsqueda local y el mejor es la búsqueda local multiarranque.

Mostramos las desviaciones típicas para justificar lo que hemos comentado sobre la robustez:

Algoritmo	Desviacion tipica Fitness
Busqueda local	0.277134172
Búsqueda local multiarranque	0.235206483

Figura 5: Desviaciones en el *fitness* de la búsqueda local y *BLM*

Por tanto, argumentamos estadísticamente que la búsqueda local es más sensible a la semilla aleatoria respecto al *fitness* de la solución obtenida que la búsqueda local multiarranque, pues su desviación típica es menor.

Ya hemos comparado la búsqueda local iterativa en sus dos variantes, concluyendo que el mejor comportamiento lo otorga el uso de enfriamiento simulado como intensificador. Esto, de nuevo, creemos que es porque la búsqueda local cae demasiado rápido en óptimos locales, desperdiciando evaluaciones del fitness sin consumir.

5.4.3. Posibles mejoras

Como ya hemos comentado, sería interesante que, en las búsquedas iterativas, en vez de fijar el parámetro de repeticiones, repitiésemos hasta agotar las evaluaciones del fitness, previo fijado del máximo de evaluaciones en cada intensificación. Con esto seguramente *ILS-Búsqueda local* habría sido más competitivo.

Por otro lado, en enfriamiento simulado, hemos dejado fijo el valor $k = 1$. Podría haber sido interesante realizar algún proceso de búsqueda como *grid search* para fijar un mejor valor de dicho parámetro k.

6. 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).