Práctica 2b
Técnicas de Búsqueda Basadas en Poblaciones
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

12 de mayo de 2021

Índice

Índice de figuras

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).

[4] "Espacio de búsqueda - wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Espacio_de_b%C3%BAsqueda. (Accessed on 12/05/2021).

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:

```
/// 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 de los algoritmos genéticos generacionales

Este algoritmo tendrá la siguiente estructura, que más adelante pasaremos a detallar con los pseudocódigos:

1. Generamos una población aleatoria. Recordar que los valores de los parámetros se desarrollan en *5.1.2. Parámetros de los algoritmos poblacionales*
2. Mientras no se hayan agotado las evaluaciones del fitness:
   a) Seleccionar elementos de la población, a partir de torneos binarios, generando una nueva población del mismo tamaño que la población anterior
   b) Cruzar con cierta probabilidad dicha población. Más adelante discutimos los distintos esquemas de cruce
   c) Mutar con cierta probabilidad genes de la población cruzada. Usaremos en todos los casos mutación uniforme
   d) Evaluar todos los elementos de la población (calcular el valor de *fitness*)
   e) Reemplazar la población anterior con esta nueva población. En caso de que el individuo mejor (el de menor valor de *fitness*) no haya sobrevivido, lo introducimos en la nueva población.
   f) **Consideración importante**: en el guión se nos indicaba que el mejor individuo reemplazase, en caso de ser necesario, al peor individuo de la población nueva. Sin embargo, por los problemas que tuve con la diversidad de la población, y por consejo del profesor de prácticas, este reemplazo se hace en la posición que ocupaba el mejor individuo en la población anterior
   g) Contabilizar las evaluaciones del fitness que se consumen en el proceso

Notar que a la hora de cruzar con una probabilidad la población, lo que hacemos por temas de eficiencia es, calcular el número esperado de genes a mutar, y mutar aleatoriamente ese número de individuos de la población. Y en este caso, también usaremos el número esperado de genes a mutar, usando de nuevo la esperanza matemática de la probabilidad dada.

Vamos a trabajar con dos esquemas de cruce:

- Cruce uniforme: por cada gen, se elige aleatoriamente el padre del que se copia dicho gen
- Segmento fijo: se copian todos los genes del primer padre en un segmento generado aleatoriamente. El resto de genes elegirlos aleatoriamente entre el primer y segundo padre. Notar que este esquema da mucho más peso al primer padre

Por cada dos padres escogidos, cruzamos generando dos hijos. Para hacer esto, cada cruce produce un único hijo. Lo que hacemos para generar, dados dos padres, dos hijos, es cruzar el primer padre con el segundo y después el segundo con el primero. Esto es claro cuando consideramos el cruce por segmento fijo, en el que hay una dependencia en el orden de los padres (da mucho más peso al primer padre).

También usaremos una optimización para el cruce. Los individuos a cruzar deberían ser aleatorios, pero como la población que cruzamos viene del proceso de selección, que ya introduce

aleatoriedad, podemos cruzar dos a dos los primeros elementos de la población de selección. Esto se verá detallado en el pseudocódigo correspondiente.

Los detalles sobre la implementación se desarrollan en los pseudocódigos.

3.4. Pseudocódigos

3.4.1. Bucle principal de ejecución

Comenzamos mostrando el pseudocódigo de mayor nivel, asociado al bucle principal de ejecución:

```
# Establecemos el valor de los parametros
#=====================================

max_fitness_evaluations = 100000;
population_size = 50;
crossover_probability = 0.7;

# Elegimos si usamos cruce uniforme o cruce por segmento fijo
# Depende de lo que pasemos por parametro
cross_uniform = True or False

# El tamaño de un gen sera el tamaño de la poblacion de datos a
   asignar a clusters
gen_size = data_points.len();

# Usamos la esperanza matematica de la probabilidad
mutation_probability_per_gen = 0.1 / gen_size;
individuals_to_mutate = mutation_probability_per_gen * gen_size *
   population_size;

# Bucle principal
#=====================================

# Poblacion inicial aleatoria
let mut current_population = new_random_population()

# Realizamos las iteraciones pertinentes
consumed_fitness_evaluations = 0;
while consumed_fitness_evaluations < max_fitness_evaluations:

    # Seleccionamos una poblacion por torneos binarios
    # Tamaño: el de la poblacion original
    let selection_population = current_population.
   select_population_binary_tournament(len(current_population))

    # Cruzamos la poblacion seleccionada
    # Segun el esquema de cruce pasado como parametro
    if cross_uniform == true:
```

```
36          crossed_population = selection_population.
    cross_population_uniform()
37      else:
38          crossed_population = selection_population.
    cross_population_segment()
39
40      # Mutamos la poblacion que ha sido cruzada
41      # Usamos el numero esperado de mutaciones
42      mutated_population = crossed_population.mutate_population(
    individuals_to_mutate);
43
44      # Comprobamos que el individuo mejor de la poblacion original
    se
45      # preserve en esta nueva poblacion
46      let final_population = mutated_population.
    preserve_best_past_parent(&current_population);
47
48      # Evaluamos la poblacion final
49      final_population.evaluate_all_individuals();
50
51      # Realizamos el cambio de poblacion
52      current_population = final_population.clone();
53
54      # Añadimos las evaluaciones de fitness consumidas en esta
    pasada
55      consumed_fitness_evaluations +=
    iteraciones_consumidas_en_esta_iteracion
56
57  # Devolvemos los resultados
58  return current_population.get_best_individual()
```

Si se lee el código de esta parte, que se encuentra en generational_genetic.rs, se puede ver cómo usamos FitnessEvaluationResult para controlar las evaluaciones del fitness consumidas. También se puede ver que estamos usando muchos debug_assert! para realizar comprobaciones de seguridad. Estas comprobaciones no se tienen en cuenta cuando ejecutamos el programa con ––release.

3.4.2. Solución inicial aleatoria

```
1  # Del struct Population
2  def new_random_population(data_points, constraints,
    number_of_clusters, population_size):
3       # Vector de individuos vacio
4       rand_population = Self{individuals: vec![]};
5
6       # Añadimos las soluciones aleatorias a la poblacion
7       for _ in 0..population_size:
8           new_individual = Solution.generate_random_solution();
9           rand_population.individuals.push(new_individual);
10
```

```
    return rand_population

# Del struct Solution
def generate_random_solution():
    return Self::new(
        # Vector de tamaño data_points.len()
        # Elementos del vector aleatorios en [0, number_of_clusters)
        (0..data_points.get_points().len()).map(aleatorio(0.. number_of_clusters),

        data_points,
        constraints,
        number_of_clusters,
    )
```

3.4.3. Operador de selección

```
# self es la poblacion sobre la que realizamos la seleccion
def select_population_binary_tournament(self, new_population_size):
    # Partimos de una poblacion sin individuos
    new_pop = Self::new_empty_population();

    # Añadimos individuos usando el torneo binario
    for _ in 0..new_population_size:

        # Los dos individuos que van a competir en el torneo
        first_candidate = self.individuals.choose_randomly()
        second_candidate = self.individuals.choose_randomly()

        # Seleccionamos el ganador
        winner = Solution::binary_tournament(first_candidate, second_candidate);
        new_pop.individuals.push(winner.clone());

    return new_pop

# Metodo del struct Solution
pub fn binary_tournament(first, second):
    first_fitness = first.fitness();
    second_fitness = second.fitness();
    if first_fitness < second_fitness:
        return first
    else:
        return second
```

3.4.4. Operadores de cruce

Mostramos ahora el pseudocódigo de los dos operadores de cruce. Notar de nuevo que estamos cruzando los primeros individuos de la población de selección. El proceso de selección ya introduce la aleatoriedad necesaria, por lo que nos ahorramos tiradas de números aleatorios.

```
def cross_population_uniform(self, crossover_probability):
    # Partimos de una poblacion identica a la dada
    let mut new_population = self.clone();

    # Calculamos el numero de genes a cruzar
    let inidividuals_to_cross = crossover_probability * self.
    population_size()

    # Cruzamos los inidividuals_to_cross primeros individos
    index = 0;
    while index < inidividuals_to_cross - 1:

        # Tomamos los dos padres
        let first_parent = new_population.individuals[index].clone
    ();
        let second_parent = new_population.individuals[index + 1].
    clone();

        # Generamos los dos hijos usando los dos padres
        let first_child = Solution::uniform_cross(first_parent,
    second_parent);
        let second_child = Solution::uniform_cross(second_parent,
    first_parent);

        # Sustituimos los dos individuos
        new_population.individuals[index] = first_child;
        new_population.individuals[index + 1] = second_child;

        # Avanzamos la iteracion
        index = index + 2;

    return new_population

def cross_population_segment(self, crossover_probability):
    # Identico al cross_population_segment salvo por:
    first_child = Solution::cross_segment(first_parent,
    second_parent);
    second_child = Solution::cross_segment(second_parent,
    first_parent);

# Metodo de Solution
def uniform_cross(first, second):
    gen_size= first.cluster_indexes.len();
    half_gen_size = (gen_size as f64 / 2.0) as usize;

```

```
39      # Generamos aleatoriamente las posiciones de los genes del
    primer padre con las que nos
40      # quedamos. Para ello, tomamos una permutacion aleatoria de
    {0, ..., gen_size - 1} y nos
41      # quedamos con la primera mitad. La segunda mitad nos indicara
        las posiciones que usamos
42      # del segundo padre
43      positions_to_mutate: Vec<usize> = (0..gen_size as usize).
    collect();
44      positions_to_mutate.shuffle(rng);
45
46      # Nueva solucion a partir de la informacion de uno de los
    padres
47      # Tomamos este first.clone() porque hace que vaya mas rapido
    el algoritmo
48      let mut crossed_solution = first.clone();
49
50      # Tomamos los elementos aleatorios del primer padre
51      for index in 0..half_gen_size:
52          # Tenemos que usar el indice que indica de la permutacion
    aleatoria
53          curr_index = positions_to_mutate[index];
54          crossed_solution[curr_index] = first[curr_index];
55
56
57      # Tomamos los elementos aleatorios del segundo padre
58      for index in half_gen_size..gen_size:
59          # Tenemos que usar el indice que indica de la permutacion
    aleatoria
60          curr_index = positions_to_mutate[index];
61          crossed_solution[curr_index] = second[curr_index];
62
63
64      # No deberia ocurrir, pero reseteo el valor del fitness para
    evitar problemas
65      crossed_solution.invalid_fitness_cache();
66
67      # Reparamos la solucion en caso de que sea necesario
68      if crossed_solution.is_valid() == false:
69          crossed_solution.repair_solution(rng)
70
71
72      return crossed_solution;
73
74  # Metodo de Solution
75  def cross_segment(first, second):
76      # Nueva solucion a partir de la informacion de uno de los
    padres
77      crossed_solution = first.clone();
78      gen_size= first.cluster_indexes.len();
79
```

```
    # Seleccionamos el inicio y tamaño del segmento
    segment_start = aleatorio(0..gen_size);
    segment_size = aleatorio(0..gen_size);

    # Copiamos los valores del primer padre
    for i in 0..segment_size:
        # Calculamos la posicion actual en el segmento
        let index = (segment_start + i)  % gen_size;

        crossed_solution[index] = first[index];


    # Copiamos, con cruce uniforme, el resto de valores
    for i in 0..(gen_size - segment_size):
        # Calculamos el indice de la misma forma que antes, partiendo de donde nos quedamos
        let index = (segment_size + segment_start + i)  % gen_size;

        # Padre del que queremos tomar la informacion
        let choose_parent = elige_aleatorio{1, 2}
        if choose_parent == 0:
            crossed_solution[index] = first[index];
        else:
            crossed_solution[index] = second[index];


    # No deberia ocurrir, pero reseteo el valor del fitness para evitar problemas
    crossed_solution.invalid_fitness_cache();

    # Reparamos la solucion en caso de que sea necesario
    if crossed_solution.is_valid() == false:
        crossed_solution.repair_solution();

    return crossed_solution;
```

3.4.5. Operador de mutación usando número esperado de mutaciones

Tenemos dos funciones para mutar una población. Una se usa en el algoritmo genético generacional, y la otra en el estacionario. Por tanto, aquí mostramos la función de mutación para el generacional, y el operador de mutación uniforme, común para todos los algoritmos poblacionales.

En este caso, consideramos la esperanza matemática de la probabilidad, para calcular el número de genes a mutar. Este cálculo se especificó ya en el bucle principal.

```
def mutate_population(self, individuals_to_mutate):
    # Copiamos la poblacion original
    new_pop = self.clone();

    # Posiciones sobre las que podemos elegir aleatoriamente
```

```
    let positions: Vec<usize> = (0..self.population_size())

    for _ in 0..individuals_to_mutate:
        # Elegimos un individuo aleatorio de la poblacion
        random_index = positions.choose_random()

        # Mutamos
        new_pop.individuals[random_index] = new_pop.individuals[
    random_index].mutated()

    return new_pop;

# Metodo de Solution
def mutated():
    # Copiamos la solucion para realizar una modificacion
    mutated_sol = self.clone();

    # Escogemos una posicion a mutar (el gen del cromosoma)
    mut_position_candidates = (0..mutated_sol.data_points.len())
    mut_position = mut_position_candidates.choose_random()

    # Podemos elegir como nuevo valor aquellos que esten en el
    itervalo adecuado y que no sean
    # el cluster original que ya teniamos, pues estariamos
    perdiendo una mutacion efectiva
    new_cluster_candidates = (0..mutated_sol.number_of_clusters)
    new_cluster_candidates.filter(|x| x != valor original
    cromosoma)
    mut_value = new_cluster_candidates.choose_random()

    # Mutamos el valor
    mutated_sol.cluster_indexes[mut_position as usize] = *
    mut_value

    // Reseteamos el fitness, porque estamos haciendo un cambio a
    la solucion que devolvemos
    mutated_sol.invalid_fitness_cache();

    # Comprobamos que la solucion sea valida. En caso de que no lo
    sea, la reparamos
    if mutated_sol.is_valid() == false:
        mutated_sol.repair_solution(rng)

    # Comprobamos que la solucion no sea la misma (puede ocurrir
    al reparar). En caso de que
    # sea la misma solucion, volvemos a mutar
    if mutated_sol.cluster_indexes == self.cluster_indexes
        return self.mutated(rng)

    return mutated_sol;
```

Notar que un mimso individuo puede mutar más de una vez. En cuyo caso, no evitamos que pueda mutar el mismo gen más de una vez.

Una cosa que nos ha dado muchos problemas ha sido comprobar que solo mutásemos genes asociados a cluster con más de un punto asignado. De esa forma, no se podrían generar soluciones no válidas. Pero esto nos restaba mucha variabilidad en nuestra población. Por ejemplo, si toda la población tiene el mismo valor en un gen concreto, y este representa el único punto asignado a un cluster, no podemos modificar esa asignación.

Por tanto, permitimos soluciones no válidas que, antes de ser devueltas, se reparan en caso de ser necesario. De esta forma, en ciertos casos, en vez de modificarse un gen, se pueden modificar más de uno (en todas las pruebas realizadas, no hemos modificado más de dos genes).

3.4.6. Reemplazamiento de la población

El método que evalúa toda la población no lo mostramos por no tener interés. Lo que sí que es interesante es mostrar cómo preservamos el mejor individuo de la población pasada en caso de que no sobreviva en la nueva población.

```
def preserve_best_past_parent(self, original_population):
    # Para realizar el cambio sin mutar la sol. original
    new_pop = self.clone();

    # Tomamos el mejor individuo de la poblacion original
    let best_individual_at_original_pop= original_population.
   get_best_individual();

    # Comprobamos si esta dentro de la poblacion
    found = self.search_individual_with_same_cluster_indixes(
   best_individual_at_original_pop)
    if found:
        # El mejor individuo ha sobrevivido
        # Devolvemos la poblacion original
        return self.clone()

    # No ha sobrevivido
    new_pop.individuals[best_individual_index_original_pop] =
   best_individual_at_original_pop
    return new_pop
```

Como ya se ha comentado, en el guión se especifica que debemos colocar al individuo en la posición del peor individuo de la población nuevo. Pero esto nos provocaba problemas con la diversidad en nuestra población, y siguiendo el consejo del profesor, decidimos colocar al individuo en la posición que ocupaba en la población pasada.

3.5. Descripción y pseudocódigo de los algoritmos genéticos estacionarios

Los algoritmos genéticos estacionarios siguen el siguiente esquema:

1. Generamos una población aleatoria. Recordar que los valores de los parámetros se desarrollan en *5.1.2. Parámetros de los algoritmos poblacionales*
2. Mientras no se hayan agotado las evaluaciones del fitness:
   a) Seleccionar dos elementos de la población. Esto, con dos torneos binarios que involucran a cuatro individuos (aunque en el proceso los individuos que compiten podrían repetirse).
   b) Cruzar los dos individuos seleccionados, dos veces, generando dos hijos. De nuevo, tenemos los mismos dos esquemas de cruce
   c) Mutar con cierta probabilidad genes de la población cruzada.
   d) Los dos individuos nuevos compiten para entrar en la población. Entran en la población cuando el fitness del individuo es mejor que el fitness del peor individuo de la pbolación
   e) Contabilizar las evaluaciones del fitness que se consumen en el proceso

En este caso, solo indicamos el pseudocódigo del bucle principal, la mutación usando la probabilidad y no el número esperado, y la competición para entrar en la población, pues el resto es común con el algoritmo genético generacional.

3.5.1. Bucle principal

```
# Parametros del algoritmo
#============================
max_fitness_evaluations = 100000
population_size = 50

# El tamaño de un gen sera el tamaño de la poblacion de datos a
   asignar a clusters
gen_size = data_points.len()
mutation_probability_per_gen = 0.1 / gen_size

# Poblacion inicial aleatoria
#============================
current_population = Population::new_random_population()

# Realizamos las iteraciones pertinentes
#============================
consumed_fitness_evaluations = 0
while consumed_fitness_evaluations < max_fitness_evaluations:

    # Tomamos dos individuos con dos torneos binarios
    selection_population = current_population.
   select_population_binary_tournament(2);

```

```
    # Cruzamos los dos individuos de la poblacion
    # Los cruzamos forzosamente <- Prob == 1.0
    crossover_probability = 1.00;
    if cross_uniform == true:
        crossed_population = selection_population.
    cross_population_uniform(crossover_probability)
    else:
        crossed_population_result = selection_population.
    cross_population_segment(crossover_probability)

    # Mutamos con cierta probabilidad los genes de los individuos
    mutated_population = crossed_population.
    mutate_population_given_prob(mutation_probability_per_gen)

    # Los dos individuos compiten para entrar a la poblacion
    let final_population = current_population.
    compete_with_new_individuals(mutated_population);

    # Por seguridad, evaluamos a la poblacion
    # En el codigo se comprueba que esto no consume evaluaciones
    del fitness
    final_population.evaluate_all_individuals();

    # Cambio de poblacion
    current_population = final_population.clone();

    # Añadimos las evaluaciones de fitness consumidas en esta
    pasada
    consumed_fitness_evaluations += evaluaciones consumidas en
    esta iteracion

return current_population.get_best_individual()
```

3.5.2. Mutación usando la probabilidad

```
def mutate_population_given_prob(mutation_probability_per_gen):
    # Copiamos la poblacion original
    new_pop = self.clone();

    # Iteramos sobre los individuos
    for (index, _individual) in self.individuals.enumerate():
        # Iteramos sobre los genes de los individuos
        for _ in 0..individual.num_genes:
            do_mutation = random(0.0, 1.0) <=
    mutation_probability_per_gen;
            if do_mutation == true:
                new_pop.individuals[index] = new_pop.individuals[
    index].mutated();

    return new_pop;
```

Notar que por cada individuo, iteramos sobre sus genes lanzando un número aleatorio que indica si mutamos un gen o no. En caso de mutar, no necesariamente mutamos el gen correspondiente a la iteración en la que nos encontramos. Con ello, es posible mutar más de una vez el mismo gen, aunque con las probabilidades tan bajas con las que trabajamos esto ocurrirá muy pocas veces.

Aquí tuvimos otro problema. El segundo bucle sobre los genes no lo realizábamos. Por ello, apenas mutaban las soluciones, lo que disminuía enormemente la diversidad de la población, generando soluciones muy malas.

3.5.3. Competición para entrar en la población

```
def compete_with_new_individuals(self, candidate_population):
    # Copiamos la poblacion original para hacer el cambio
    new_pop = self.clone();

    for candidate in candidate_population.individuals:
        # Tomamos el peor individuo de la poblacion original
        # Aquel con mayor valor de fitness
        worst_individual = new_pop.get_index_worst_individual();

        # Decidimos si el candidato entra o no en la poblacion
        if candidate.fitness() < worst_individual.fitness() {
            new_pop.individuals[worst_individual_index] =
   candidate.clone();

    return new_pop
```

3.6. Descripción y pseudocódigo de los algoritmos meméticos

Los algoritmos meméticos que vamos a desarrollar son una hibridación de los algoritmos genéticos que ya hemos visto, junto a una búsqueda local. El esquema general de los algoritmos meméticos es el siguiente:

1. Generamos una población inicial aleatoria. Los valores de los parámetros se pueden consultar en *5.1.3. Parámetros de los algoritmos memeticos.*
2. Mientras no se hayan agotado las evaluaciones del fitness
   a) Aplicar una iteración del algoritmo genético base. Podemos elegir entre el algoritmo genético generacional o estacionario
   b) Cada diez iteraciones del algoritmo genético, aplicamos una búsqueda local suave sobre la población

Como algoritmo genético base hemos escogido el generacional. Tenemos el algoritmo memético implementado con las dos bases genéticas, y el generacional es el que mejores resultados a proporcionado. Como operador de cruce en el algoritmo genético base, hemos empleado en ambos casos el cruce uniforme. De nuevo, por haber generado mejores resultados al compararlo con el cruce de segmento fijo.

A la hora de aplicar la búsqueda local suave, tenemos tres opciones referentes a qué parte de la población queremos aplicar dicha intensificación:

- Aplicar la búsqueda local a toda la población
- Aplicar la búsqueda local a un porcentaje (*5.1.3. Parámetros de los algoritmos memeticos*) aleatorio de la población
- Aplicar la búsqueda local al porcentaje dado de la población con mejor valor de *fitness*

Estas variantes suponen distintas elecciones entre exploración y explotación.

3.6.1. Bucle principal del algoritmo memético

El código es prácticamente igual que en los genéticos (dependiendo de si elegimos una base generacional o una estacionaria). La diferencia es que dentro del for, cada diez iteraciones aplicamos el tipo de búsqueda local suave especificada

3.6.2. Búsqueda local suave

Dado un cromosoma, recorremos aleatoriamente y sin repetición sus genes, asignando en cada momento el mejor cluster a cada punto (esta asignación es lo que representa cada gen). En este recorrido, paramos de iterar cuando cometemos ξ fallos. El valor de ξ se especifica en *5.1.3. Parámetros de los algoritmos memeticos.*

Consideramos un fallo cuando no conseguimos, dado un gen del cromosoma, encontrar otro que mejore su valor del fitness. Esto se hace para intentar evitar desbalancear el equilibrio entre exploración y explotación. A continuación mostramos el código de la búsqueda local suave:

```

def soft_local_search(self, max_fails):
    # Clonamos la solucion para devolver la solucion alterada
    new_solution = self.clone();

    # Recorreremos las posiciones de los puntos en orden aleatorio
    let mut indixesi = (0..self.data_points.len()).shuffle();

    # Valores iniciales para el algoritmo
    let mut fails = 0;
    let mut i = 0;

    while fails < max_fails && i < self.data_points.len():
        # Posicion a cambiar en esta iteracion
        let index = indixes[i];

        # Seleccionamos el mejor cluster para este punto en la posicion index
        new_cluster = new_solution.select_best_cluster(index);

        # Realizamos el cambio, guardando el valor original de la asignacion
        past_cluster = new_solution.cluster_indexes[index];
        new_solution.cluster_indexes[index] = new_cluster;

        # Comprobamos si hemos fallado
        if new_cluster == past_cluster:
            fails += 1;

        i += 1;

    return new_solution
```

3.6.3. Distintos esquemas de búsqueda local suave

Mostramos las distintas estrategias empleadas para realizar la búsqueda local suave sobre los individuos de la población:

```

def soft_local_search_all(self, max_fails):
    new_pop = self.clone();

    # Aplicamos la busqueda local suave a todos los individuos de la poblacion
    for index, _ in self.individuals.enumerate():
        new_individual = new_pop.individuals[index].soft_local_search(max_fails);
```

```
8           new_pop.individuals[index] = new_individual
9
10      return new_pop
11
12  def soft_local_search_random(self, max_fails, search_percentage):
13      new_pop = self.clone();
14
15      # Numero de individuos sobre los que vamos a realizar la
    busqueda local suave
16      number_of_individuals_to_intensify = self.individuals.len() *
    search_percentage
17
18      # Indices de todos los individuos ordenados aleatoriamente
19      individuals_indixes = (0..self.individuals.len()).shuffle()
20
21      # Aplicamos la busqueda local solo a un numero dado de los
    individuos. Usando los indices
22      # en orden aleatorio, escogemos aleatoriamente a dichos
    individuos
23      for i in 0..number_of_individuals_to_intensify:
24          # Escogemos aleatoriamente al individuo
25          index = individuals_indixes[i]
26
27          # Aplicamos la busqueda local a ese individuo
28          let new_individual = new_pop.individuals[index].
    soft_local_search(max_fails);
29          new_pop.individuals[index] = new_individual
30
31      return new_pop
32
33  def soft_local_search_elitist(self, max_fails, search_percentage):
34      new_pop = self.clone();
35
36      # Numero de individuos sobre los que vamos a realizar la
    busqueda local suave
37      number_of_individuals_to_intensify = self.individuals.len()
    search_percentage
38
39      # Seleccionamos los indices del mejor porcentaje de la
    poblacion
40      best_indixes = self.select_best_indixes(
    number_of_individuals_to_intensify);
41
42      # Aplicamos la busqueda local a este porcentaje mejor de
    individuos
43      for index in best_indixes:
44
45          # Aplicamos la busqueda local a ese individuo
46          new_individual= new_pop.individuals[index].
    soft_local_search(max_fails)
47          new_pop.individuals[index] = new_individual
```

```
48
49      return new_pop
50
51  def select_best_indixes(self, number_of_individuals):
52      best_indixes = []
53
54      # Necesitamos que toda la poblacion este evaluada para poder
        ordenar a sus individuos
55      self.evaluate_all_individuals();
56
57      # Usamos una cola con prioridad para tomar facilmente a los
        mejores individuos
58      # Cola de la forma: indice - fitness del indv asoc. al indice
59      priority_queue = PriorityQueue::new()
60      for (index, individual) in self.individuals.enumerate():
61          priority_queue.push(index, individual.fitness());
62
63      # Tomamos los mejores elementos
64      for (it, index) in priority_queue.into_sorted_iter().enumerate
        ():
65          best_indixes.push(index)
66
67          if it >= number_of_individuals:
68              break
69      return best_indixes
```

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

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 los algoritmos poblacionales

- Máximo de evaluaciones de la función fitness en búsqueda local: 100.000
- Tamaño de la población: 50 cromosomas
- Probabilidad de cruce AGG: 0.7
- Probabilidad de cruce AGE: 1.0
- Probabilidad mutación por gen: $\frac{0,1}{numero\ de\ genes}$

5.1.3. Parámetros de los algoritmos memeticos

- Máximo de evaluaciones de la función fitness en búsqueda local: 100.000
- Tamaño de la población: 50 cromosomas

- Probabilidad de cruce AGG: 0.7
- Probabilidad de cruce AGE: 1.0
- Probabilidad mutación por gen: $\frac{0,1}{numero\ de\ genes}$
- Número de fallos: $\xi = 0,1 * n$ donde n es el número de genes por cromosoma

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.

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	21.60	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
Genético Generacional - Cruce Uniforme	5.4	0.623351081	0.66427	1.427320604	50	0.193264172	0.242466219	3.47	50	0.193264172	0.24247	3.468223541
Genético Generacional - Cruce Segmento	5.4	0.600100044	0.64102	1.443166992	41.8	0.214575288	0.255708199	3.47	41.8	0.214575288	0.25571	3.473923843
Genético Estacionario - Cruce Uniforme	6.25	0.695857393	0.74322	1.873449363	30.8	0.2239915	0.254299961	4.07	519.8	0.155250598	0.29455	7.945436751
Genético Estacionario - Cruce Segmento	6.4	0.651889009	0.70039	1.835720023	43.4	0.220647027	0.263354404	4.02	544.8	0.156466938	0.30247	7.890710235
Memético - Todos	8.4	0.691463508	0.75512	1.289209223	281	0.320698432	0.610204019	3.31	799.6	0.224365101	0.53967	11.36137977
Memético - Aleatorio	9.6	0.628737286	0.70148	1.366496409	35.2	0.20186861	0.236506851	3.28	342.2	0.164208689	0.25591	6.680055605
Memético - Elitista	10	0.687337882	0.76311	1.361656647	154.4	0.266851972	0.418787892	3.25	884.8	0.230909814	0.46803	6.731610198

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	7.40	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
Genético Generacional - Cruce Uniforme	17.8	0.684204501	0.75608	1.669733918	17.8	0.684204501	0.756082345	1.67	17.8	0.684204501	0.75608	1.669733918
Genético Generacional - Cruce Segmento	19.4	0.664598662	0.74294	1.680982231	19.4	0.664598662	0.742937435	1.68	929.8	0.149768671	0.27942	9.902319878
Genético Estacionario - Cruce Uniforme	19.6	0.689226374	0.78395	2.072718815	67.4	0.241035309	0.276124392	5.17	1002.6	0.152430511	0.29223	10.82754189
Genético Estacionario - Cruce Segmento	18.8	0.674065452	0.74998	2.076220715	56.4	0.234590301	0.26395268	5.12	1012.4	0.15191929	0.29309	10.7759186
Memético - Todos	18.8	0.688420927	0.76434	1.530783596	45.1	0.307479149	0.560089632	4.5	1623.8	0.216993242	0.54093	17.39531764
Memético - Aleatorio	15.2	0.686788359	0.74817	1.659262742	58.8	0.222764647	0.253376488	4.35	562.8	0.164004551	0.24324	9.554149861
Memético - Elitista	10.8	0.730631364	0.77424	1.591715102	22.5	0.259889999	0.377027147	4.35	1650.6	0.213226954	0.44339	9.642406734

Figura 2: Tablas Globales - 20 % de restricciones

5.4. Análisis de resultados

Por las correcciones en la búsqueda local y el algoritmo *greedy*, se puede ver que los resultados respecto a estas dos búsquedas han cambiado respecto la práctica anterior. Estas correcciones se comentan en *6. Correcciones respecto la práctica pasada.* Es notorio que, en *Bupa 20 %*, tengamos un tiempo de ejecución tan pequeño. Esto se justifica en la corrección al ver lo que hemos tenido que hacer para poder asignar valor a este *dataset*.

El algoritmo memético elitista está generando resultados malos de forma consistente, en comparación a la búsqueda local y el resto de genéticos y meméticos. A compañeros nuestros les ofrecen resultados mucho mejores, lo que nos hace pensar que hemos cometido un error en la práctica. Este error puede venir dado por:

- Durante toda la práctica hemos tenido problemas con la diversidad en la población. Si este problema ha persistido sin tener mucho impacto en el resto de algoritmos, en el elitista se puede ampliar. Esto porque estamos haciendo más grande la diferencia entre individuos de buen *fitness* e individuos de mal *fitness*
- En la selección del mejor individuo, usamos una cola con prioridad. Puede ser que nos esté devolviendo los peores elementos, en vez de los mejores.

El problema era el segundo. Cambiamos el código y repetimos las ejecuciones, para Mem ético Elitista. No podemos guardar la traza y depurarla de nuevo con el poco tiempo del que disponemos, así que directamente mostramos los resultados en la siguientes tablas. Además, aprovechamos para marcar el mejor y peor resultado, que será de utilidad a la hora de realizar los análisis:

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	21.60	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	0.7559	0.094328311	30.6	0.227662417	0.257774069	0.5	116	0.115242325	0.1463	5.900788302
Genético Generacional - Cruce Uniforme	5.4	0.623351081	0.6643	1.427320604	50	0.193264172	0.242466219	3.47	50	0.193264172	0.2425	3.468223541
Genético Generacional - Cruce Segmento	5.4	0.600100044	0.641	1.443166992	41.8	0.214575288	0.255708199	3.47	41.8	0.214575288	0.2557	3.473923843
Genético Estacionario - Cruce Uniforme	6.25	0.695857393	0.7432	1.873449363	30.8	0.2239915	0.254299961	4.07	519.8	0.155250598	0.2946	7.945436751
Genético Estacionario - Cruce Segmento	6.4	0.651889009	0.7004	1.835720023	43.4	0.220647027	0.263354404	4.02	544.8	0.156466938	0.3025	7.890710235
Memético - Todos	8.4	0.691463508	0.7551	1.289209223	281	0.320698432	0.610204019	3.31	799.6	0.224365101	0.5397	11.36137977
Memético - Aleatorio	9.6	0.628737286	0.7015	1.366496409	35.2	0.20186861	0.236506851	3.28	342.2	0.164208689	0.2559	6.680055605
Memético - Elitista	7	0.590481008	0.6435	1.389591232	39	0.193526683	0.231904279	3.3	230	0.140826183	0.2025	6.718178838

Figura 3: Tablas Globales - 10 % de restricciones, **Corregidas**

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	7.40	0.09	1.03	0.22	4.6	-0.01	0.35	0.35	2163	0.255368309	0.557	0.003718075
BL	69.47142857	0.348900604	0.4207	2.644544083	53.39081633	0.389219617	0.458977195	1.75	212.6	0.357252727	0.4227	2.328629045
Genético Generacional - Cruce Uniforme	17.8	0.684204501	0.7561	1.669733918	17.8	0.684204501	0.756082345	1.67	17.8	0.684204501	0.7561	1.669733918
Genético Generacional - Cruce Segmento	19.4	0.664598662	0.7429	1.680982231	19.4	0.664598662	0.742937435	1.68	929.8	0.149768671	0.2794	9.902319878
Genético Estacionario - Cruce Uniforme	19.6	0.689226374	0.7839	2.072718815	67.4	0.241035309	0.276124392	5.17	1002.6	0.152430511	0.2922	10.82754189
Genético Estacionario - Cruce Segmento	18.8	0.674065452	0.75	2.076220715	56.4	0.234590301	0.26395268	5.12	1012.4	0.15191929	0.2931	10.7759186
Memético - Todos	18.8	0.688420927	0.7643	1.530783596	45.1	0.307479149	0.560089632	4.5	1623.8	0.216993242	0.5409	17.39531764
Memético - Aleatorio	15.2	0.686788359	0.7482	1.659262742	58.8	0.222764647	0.253376488	4.35	562.8	0.164004551	0.2432	9.554149861
Memético - Elitista	29	0.603946567	0.7211	1.641801474	29	0.251091159	0.262367417	4.43			m	9.777568744

Figura 4: Tablas Globales - 20 % de restricciones, **Corregidas**

Tras este cambio, vemos que el algoritmo memético elitista tiene el comportamiento deseado, con lo que podemos pasar a analizar los resultados.

5.4.1. Análisis Global

Los únicos algoritmos que han sido vencedores más de una vez han sido el memético elitista y la búsqueda local. Respecto al memético elitista, era en parte esperable, pues es el que conlleva mayor explotación, y para ciertos problemas esto puede ser lo deseado. Destaca que, a pesar de estar usando algoritmos mucho más avanzados que la búsqueda local, en ciertos problemas es el mejor algoritmo. Y en general, podemos apreciar que es bastante estable *entre distintos tipos de problemas*, no llegando nunca a ser el peor algoritmo. Con esto no queremos decir que sea estable *con distintos valores de la semilla*, porque como ya hemos visto en la práctica anterior, esto no es cierto.

Debemos sospechar en cierta medida que los conjuntos de datos no sean lo suficientemente complejos como para que la búsqueda local se vea afectada por caer en malos óptimos locales. Un *dataset* con muchísimos óptimos locales, debería mostrar que los algoritmos poblacionales generan consistentemente mejores soluciones. Por ejemplo, el que se visualiza en la siguiente imagen [2]:

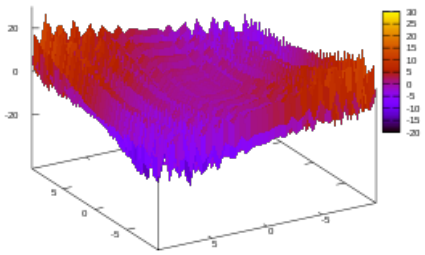

Figura 5: Espacio de búsqueda con muchos óptimos locales

Podemos ver que en el resto de algoritmos, dependemos muchísimo del *dataset* a la hora de tener un buen o mal comportamiento respecto de los otros algoritmos. Cada algoritmo establece un equilibrio exploración-explotación distinto. Por tanto, a la hora de enfrentarnos a un problema concreto, deberemos estudiar distintos enfoques e intentar usar el conocimiento experto disponible para tomar una decisión en cuanto a la hora de fijar el algoritmo que vamos a emplear. Estamos justificando algo que ya sabíamos, por ejemplo, del *Teorema Free Lunch*: no disponemos de un algoritmo muy bueno que funcione para todo tipo de problemas.

El algoritmo memético que aplica búsqueda local a todos los miembros de la población genera resultados que son o bien regulares o bien malos. Intuitivamente, podemos pensar que estamos desperdiciando evaluaciones del fitness en intensificar soluciones malas. Es más, estamos perdiendo evaluaciones en intensificar soluciones que son muy parecidas de base, a soluciones que son todavía más parecidas al intensificar. Por tanto, disminuimos la diversidad de la población. Estas soluciones mejoradas de una solución mala a una solución menos mala, pero no buena, se podrían haber obtenido a partir de cruces y mutaciones, sin malgastar evaluaciones del fitness.

También podemos ver que en la mitad de los casos, un algoritmo memético ha sido el que mejores resultados ha generado. Exceptuando los resultados regulares del memético que aplica búsqueda a todos los individuos, podemos observar resultados consistentes en los otros dos

[2]Imagen extraída de: [4]

meméticos. Por tanto, a la hora de enfrentarnos a un problema sin demasiado conocimiento, trataríamos de empezar explorando la búsqueda local, y la búsqueda memética. Aunque para ello primero debemos implementar un genético, por lo que también podríamos tenerlo en cuenta. Visto desde otra perspectiva, si disponemos de un algoritmo genético que funcione medianamente bien, siempre trataríamos de aplicar una mejora memética, pues hemos visto que suelen mejorar los resultados.

Para conjuntos de datos grandes, vemos que existe una tendencia hacia que los estacionarios funcionen mejor. Y al revés, con conjuntos de datos de menor tamaño, y con menos restricciones, los algoritmos genéticos generacionales parecen funcionar mejor. Esto introduce otro posible criterio a la hora de escoger un algoritmo u otro, en base a los conjuntos de datos con los que trabajemos.

Usando el *excel* que entregamos con la práctica, podemos ver las desviaciones típicas de cada tipo de búsqueda, para comparar la robustez de los algoritmos frente a los factores aleatorios:

Algoritmo	Desviacion tipica Fitness
Busqueda local	0.277134172
Genético Generacional Uniform	0.218058378
Genético Generacional Segment	0.206175669
Genético Estacionario Uniforme	0.234904797
Genético Estacionario Segmento	0.216223187
Memético - Todos	0.111547428
Memético - Aleatorio	0.232461809
Memético - Elitista	0.180510447

Figura 6: Desviaciones típicas de los *fitness* de los algoritmos

Como era de esperar, el algoritmo con desviación más alta es el de búsqueda local. Era intuitivo pensar esto, pues es el que pensamos que está más afectado por la semilla inicial. En un principio es sorprendente lo poco que se desvía el memético sobre todos los individuos. Creemos que es debido a que la intensificación provoca que todo el conjunto avance rápidamente hacia una solución muy similar (pérdida de diversidad), y por ello, es más estable respecto a las distintas semillas aleatorias.

Del mismo modo, también es de esperar que el algoritmo elitista, que hace que las mejores soluciones avancen aún más, tenga una desviación tan pequeña. Los cuatro generacionales tiene desviaciones similares. No sabemos si las diferencias en desviación son significativas, sobre todo teniendo en cuenta que solo hemos lanzado 5 veces los algoritmos.

De los meméticos, el que más desviación tiene es el de porcentaje aleatorio. De nuevo, esto también era esperable, pues es el que mayor aleatoriedad introduce a la hora de seleccionar los individuos a intensificar.

Respecto a las restricciones violadas, el algoritmo *greedy* lógicamente es el que mejores resultados genera. Gracias a que el único criterio que emplea es el de minimizar este valor. Y sin embargo, ya hemos comentado que no genera soluciones de calidad. Es destacable que los algoritmos meméticos han sido capaces de obtener soluciones buenas, con más restricciones violadas que el resto de algoritmos. Por ejemplo, en *Bupa 10 %*.

Por tanto, concluimos que la mejor estrategia es minimizar tanto las restricciones violadas como la distancia intra-cluster, según el equilibrio dado por el valor de λ.

Respecto a los tiempos de ejecución, es de esperar que el más rápido sea *greedy*. Aunque en casos en los que la búsqueda local converja prematuramente a la solución, puede ser que

esta última sea más rápida. Esto pasa, por ejemplo, en *zoo 10 %* y *Bupa 10 %*. También, que *greedy* cicle sobre soluciones muy parecidas hasta que forzosamente paramos el algoritmo puede favorecer a este comportamiento.

En general, los algoritmos genéticos estacionarios son algo más lentos que los genéticos generacionales. Al realizar cambios de población de tamaño completo, en generacional estamos consumiendo las evaluaciones del fitness en menos iteraciones. Por lo tanto, podemos pensar que en generacional estamos realizando las operaciones en *batch*, mientras que en estacionario tenemos que repetir todos los pasos del algoritmo de dos en dos individuos. En *Bupa 20 %* estas diferencias de tiempo son notables, difiriendo en más de 7 segundos.

El algoritmo memético sobre toda la población, hace que los tiempos sean aún más grandes. Sin embargo, los otros dos algoritmos meméticos mejoran algo los tiempos de ejecución respecto a los genéticos estacionarios. En algunas ocasiones mejoramos los tiempos de los algoritmos generacionales, pero no vemos suficiente consistencia en los datos para sacar conclusiones de ello.

Por tanto, cuando trabajemos con conjuntos de datos mucho más grandes, el tiempo de ejecución puede ser un factor a tener en cuenta cuando elijamos el algoritmo a emplear. Teniendo en cuenta que el memético sobre un porcentaje aleatorio y el memético elitista, cuando no mejora al genético estacionario, se queda muy cerca del genético generacional, y a vista de las mejoras de los resultados, en muchas ocasiones podemos considerar emplear un memético como mejora a un genético sin tener demasiado miedo al impacto en tiempos de ejecución.

A la hora de ejecutar los algoritmos, viendo cómo avanzaban las poblaciones, hemos observado un comportamiento que nos ha parecido algo problemático. Tras no demasiadas iteraciones, la población de individuos se estabilizaba alrededor de unas pocas soluciones repetidas muchas veces. Por tanto, sería interesante usar otros parámetros del programa para introducir más diversidad en fases tardías de los algoritmos.

6. Correcciones respecto la práctica pasada

- En la búsqueda local, en vez de considerar iteraciones, consideramos el máximo de evaluaciones del fitness
- En la búsqueda local, en vez de considerar iteraciones, consideramos el máximo de evaluaciones del fitness
- Hemos puesto finalmente la solución de *Copkmeans* para *Bupa 20 %* en las tablas

Estas correcciones no las plasmamos en el pseudocódigo, pues pensamos que es lo lógico al no ser esta parte evaluada de nuevo en esta práctica. Realizamos las correcciones para que las comparaciones entre distintos algoritmos sean lo más justas posbile.

El problema que teníamos con *greedy*, no es que en algún momento alcanzásemos una solución no válida. En ese caso, sería sencillo devolver la última solución válida alcanzada. El problema es que no éramos capaces de generar ni una sola solución valida. Para solucionar esto, en el caso de *Bupa 20 %*, y **únicamente en ese caso**, hemos empleado el siguiente código para obtener al menos unos valores:

```
for i in 0..data_points.len(){
    current_cluster_indixes[i as usize] = (i % number_of_clusters
   as usize) as u32;
}
let curr_sol = Solution::new(
    current_cluster_indixes.clone(),
    data_points,
    constraints,
    number_of_clusters,
);

return (Some(curr_sol), fitness_evolution);
```

Construimos una solución cíclica y devolvemos esa como resultado, que claramente será válida.