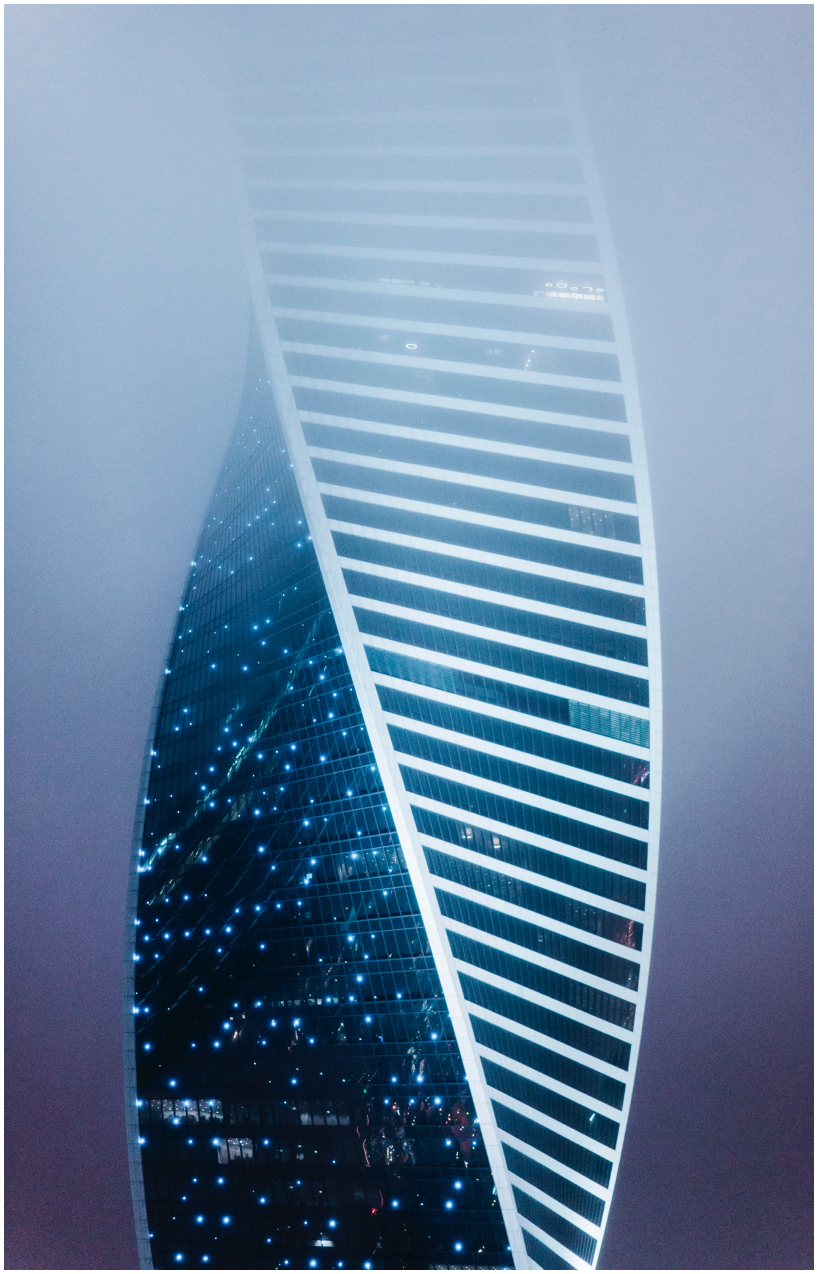
Algoritmos para resolver el problema de agrupación con restricciones



Autor: Andrés Millán
DNI:
Email: amilmun@correo.ugr.es
Grupo de prácticas: MH3

Tabla de contenidos

Sobre esta memoria

En esta memoria se recoge toda la información necesaria para resolver el problema del **agrupamiento con restricciones**, así como la documentación y el desarrollo de la práctica 2 de la asignatura Metaheurísticas.

Todo el código está subido en el repositorio de Github https://github.com/Asmilex/Metaheuristicas. Para ejecutarlo, es necesario tener instalado Rust. El comprimido de la entrega contendrá los mismos archivos que se encuentran en el repositorio. El único cambio será la fecha de la versión y la localización del PDF generado a partir de este archivo.

Se puede compilar y correr el proyecto con `cargo run --release`. Sin embargo, es necesario especificar ciertos parámetros de entrada, que dependerán de lo que se quiera hacer. Estas son las posibilidades:

Benchmark de uno o más algoritmos

Escribir en la línea de comandos `cargo run --release benchmark [algoritmos]`. donde `[algoritmos]` son uno o más elementos de la siguiente lista:

- `greedy`.
- `bl`.
- Algoritmos genéticos
  - Se puede introducir `geneticos` para ejecutarlos todos
  - Alternativamente, se pueden especificar a mano: `agg_un`, `agg_sf`, `age_un` o `age_sf`.
- Algoritmos meméticos:
  - Usando `memeticos` se ejecutarán todos los de esta categoría descritos en esta documentación
  - Alternativamente, se pueden especificar a mano: `am_10_1`, `am_10_01`, `am_10_01_mejores`.

Si no se especifica ninguno, se usarán todos. Cada algoritmo se ejecuta 5 veces por dataset (por lo que cada uno se realiza 30 veces). La información resultante se exportará al archivo `./data/csv/[dataset]_[número de restricciones]/[nombre del algoritmo].csv`, el cual contendrá las medidas necesarias para el posterior análisis que realizaremos. Por ejemplo, un archivo sería `/data/csv/bupa_10/age_sf.csv`.

Tras ejecutar esta opción, se analizarán automáticamente los resultados, produciendo los archivos correspondientes indicados en la sección Analizar los archivos csv

Ejecutar uno o varios algoritmos para un dataset en particular

Para ejecutar un único algoritmo para un cierto dataset, se debe introducir en la línea de comandos `cargo run --release [dataset] {10, 20} [algoritmos]`, donde `[dataset]` puede valer `bupa`, `glass` o `zoo`, eligiendo qué conjunto de restricciones usar (`10` o `20`). La lista de algoritmos funciona de la misma manera que en el apartado anterior.

Analizar los archivos `.csv`

Una vez se hayan ejecutado los benchmarks correspondientes, se puede extraer automáticamente la información de los resultados. Para ello, se debe introducir `cargo run --release analyze`. Esto exportará un archivo llamado `analisis.csv` en las carpetas de los datasets con la media de cada algoritmo.

Descripción del problema

A lo largo de estas prácticas se resolverá el problema del **agrupamiento con restricciones**. Este es una modificación del clásico problema del *clustering*, el cual se describe de la siguiente forma:

Se nos presenta una lista de elementos con un cierto número de atributos. Los representaremos como vectores en $[0, 1]^d$. Debemos agruparlos en un cierto número de categorías, llamados **clústers**, de forma que se minimice la distancia entre estos vectores.

Nuestro matiz consiste en que les pondremos restricciones a los elementos a analizar. De esta forma, forzaremos a que dos vectores deban localizarse en en mismo clúster, o lo contrario; que deban estar en clústers distintos. Por tanto, no solo debemos conseguir una denominada *distancia intraclúster* baja, sino que se debe violar el mínimo número de restricciones posibles.

Durante estas prácticas propondremos diferentes algoritmos para resolver este problema.

En la práctica 1, presentaremos soluciones sencillas basadas en algoritmos simples como **Greedy** o **Búsqueda Local**.

En la práctica 2, la cual es la que vamos a tratar, implementaremos algunas versiones de los algoritmos **genéticos** y **meméticos**.

Procedimiento seguido para resolver la práctica

Todo el código está escrito en **Rust**. Es un lenguaje moderno, eficiente y fiable en cuestiones relativas a accesos a memoria. Su elección ha sido puro interés personal.

Crates usadas

Para facilitar la implementación, se han utilizado una serie de *crates* (nombre que reciben las librerías por parte de los *rustáceos*). Estas son:

- Naglebra: una librería de álgebra lineal. Utilizada para operaciones con matrices y vectores.
- Rand: para los generadores de números aleatorios.
- Multimap: para el almacenamiento eficiente de las listas de restricciones.
- Csv: para exportar los resultados a `.csv`.
- Serde: Para facilitar el análisis de resultados. Me permite leer fácilmente los archivos `.csv`.
- Colored: para hacer más bonitas y legibles las salidas a consola.

Estructura del programa

Se han dividido las funcionalidades clave del programa en distintos ficheros. Estos son:

- `main.rs`: contiene el código relacionado con la realización de una ejecución simple y de un benchmark, así como el *parseo* de argumentos.
- `file_io.rs`: aquí se ubican las funciones relacionadas con entrada/salida de archivos. En particular, la lectura de los ficheros de restricciones y la salida a los archivos csv.
- `utils.rs`: se definen las diferentes estructuras relacionadas con el problema. Por ejemplo, `ParametrosDataset` agrupa los datos necesarios para operar un cierto dataset.
- `algorithm.rs`: todos los algoritmos implementados se encuentran aquí.
- `cluster.rs`: las principales estructuras necesarias para resolver el problema se localizan en este fichero. Específicamente, la clase `Clusters`. Estudiaremos a fondo sus elementos.
- `operator.rs`: en este fichero se definen e implementan los operadores utilizados a partir de la práctica 2.

La clase `Clusters`

Esta estructura supondrá el grueso de nuestro programa. *Agrupará* toda la información pertinente a la resolución del problema. En las siguientes secciones, describiremos sus elementos. No obstante, omitiremos las funciones de poco interés didáctico

Sobre las dimensiones

Necesitaremos tres medidas para generar una solución:

- `num_clusters` representa el número de clústers fijado por el problema.
- `dim_vectores` es el número de atributos del dataset.
- `num_elementos` es el número de vectores o muestras del dataset.

Elementos del espacio

La clase conoce en todo momento el conjunto de elementos del dataset que estamos tratando, así como sus distancias respectivas. Los miembros que se encargan de guardar esta información son `espacio` y `distancias` respectivamente. El cálculo de la variable λ, de la cual hablaremos más tarde, se guarda cuál es el máximo de las distancias al calcular la matriz `distancias`.

Para almacenar los vectores, hemos utilizado un vector de `Nalgebra::DVector`, un tipo de array dinámico con funciones de álgebra lineal. Esto nos será de gran ayuda, pues simplificará las operaciones del espacio vectorial con el que tratamos.

Representación de las soluciones

Las soluciones se representan con una lista de enteros, `lista_clusters`, de forma que, para una cierta entrada i de dicha lista:

- Si su valor es 0, entonces, ese elemento no tiene clúster asignado
- En otro caso, su valor está en el conjunto $\{1, \dots, num\_clusters\}$.

Una solución solo se considerará válida si todo clúster tiene al menos un elemento asignado. Si algún elemento ha modificado su clúster, la clase automáticamente lo registra en la estructura `recuento_clusters`, por lo que nos resultará sencillo comprobar cuántos elementos hay en cada uno.

Restricciones

Representaremos las restricciones de dos formas distintas:

1. La primera de ellas es mediante una matriz dinámica de *Nalgebra* (`restricciones`) con entradas que toman valores en $0, 1, -1$. Para una cierta entrada $[(i, j)]$, si su valor es 0, no hay ninguna restricción aplicada del vector con posición i y el vector j. Si es 1, entonces es una restricción del tipo `Must-Link`; esto es, deben ir agrupadas en el mismo clúster. Si su entrada es -1, ocurre lo contrario al caso anterior: estos dos vectores tienen una restricción del tipo `Cannot-Link`, y deben ir en clústers distintos. Esta estructura de datos nos resultará útil cuando queramos calcular el infeasibility de todo el sistema.
2. La segunda es un *hashmap* para cada tipo de restricción. Dado un cierto índice i, los hashmaps `restricciones_ML` y `restricciones_CL` devuelven todos los índices con los que tienen restricciones. Aceleran muchísimo el cálculo del infeasibility generado por la asignación de un clúster a un cierto elemento.

El número de restricciones se guarda al crear la matriz de restricciones.

Estadísticos

El interés de este problema reside en ser capaces de crear clústers lo más verosímiles posibles entre sí. Por tanto, para determinar cómo de buena es una solución, necesitamos algún tipo de estadístico que nos informe de ello. Debido a la naturaleza del problema, vamos a considerar dos: El **infeasibility** y el **fitness**.

- **Infeasiblity** es una medida de cuántas restricciones han sido violadas en conjunto; es decir, cuántos elementos con restricción *Cannot-Link* han caído en el mismo clúster, y cuántos vectores con restricción del tipo *Must-Link* se encuentran en clústers separados. La función `infeasibility()` nos permite conocer esto.

Sin embargo, no siempre nos interesa saber cuál es el estado de todo el sistema, sino cómo de malo sería meter un elemento en un cierto clúster. Para esto sirve la función `infeasibility_esperada(indice, clúster)`. Es una forma mucho más rápida de comprobar incrementos y decrementos en el sistema.

- En este problema, tanto las restricciones incumplidas como la distancia entre los elementos son importantes. Por ello, el **fitness** considera ambas. Éste se define de la siguiente manera:

donde la desviación general de la partición es la media de la suma de las distancias medias intraclúster, y λ se define como el cociente entre el máximo de las distancias en el sistema y el número de restricciones totales del sistema. Se puede conocer gracias a la función `fitness()`.

El pseudocódigo de las funciones que calculan estos valores es el siguiente:

Infeasibility

```
infeasibility():
    infeasibility = 0
    M = Matriz de restricciones
    n = tamaño de la matriz de restricciones

    Para i en 0 .. n
        Para j en 0 .. n
            Si M[i, j] == 1
                Si solucion[i] != solucion[j]
                    infeasibility++
            Si M[i, j] == -1
                Si solucion[i] == solucion[j]
                    infeasibility++

    infeasibility
```

Fitness

```
fitness():
    desviacion_general_particion + lambda() * infeasibility()
```

Semillas

Dado que todos los métodos de resolución que programemos requieren aleatoriedad, fijaremos unas semillas para todos los generadores del programas. Se encuentran almacenadas en la clase `utils.rs/Semillas`. Estas son: `328471273`, `1821789317287`, `128931083781`, `1802783721873`, `9584985309`. El generador usado es el que recomienda la documentación del crate `rand`: `StdRng`. El algoritmo que utiliza es el Chacha block cipher de 12 rondas.

Práctica 1: Greedy K-medias y búsqueda local

Greedy

El algoritmo **Greedy K-medias aplicado a clustering con restricciones** es capaz de proporcionarnos una solución relativamente buena en muy pocos milisegundos. Su implementación es muy sencilla, así como la idea que hay tras éste.

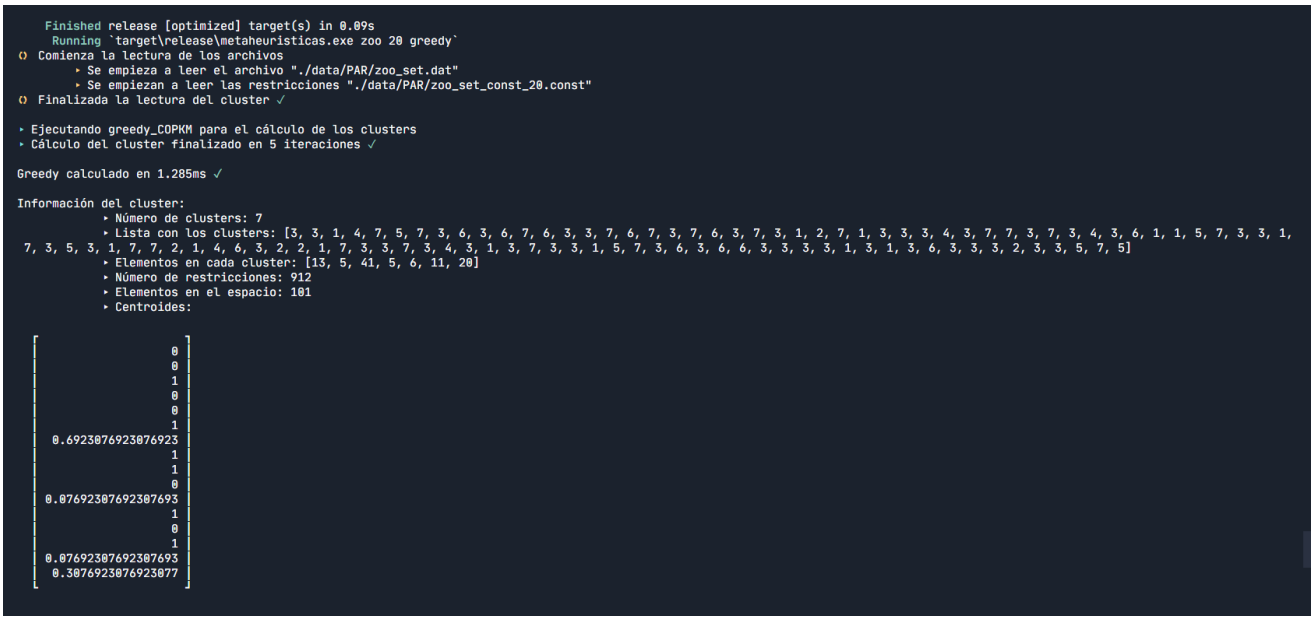
Descripción del algoritmo

Partiendo de un clúster vacío, pero con todos los elementos cargados, consideramos una serie de centroides aleatorios, tantos como número de clústers debamos generar. Recorremos los elementos del espacio de forma aleatoria, de manera que asignamos cada uno al clúster en el que menor número de restricciones se viola (esto es, de menor infeasibility). En caso de empate, se asigna al clúster con centroide más cercano a nuestro punto, entendiendo por cercano a aquel centroide que minimiza la distancia euclidiana. Se actualizan los centroides, y se repite todo hasta que la solución se estabilice.

El pseudocódigo, por tanto, quedaría así:

```
Greedy_COPKM

1. Generar centroides aleatorios con distribución uniforme en R^d.
2. Barajar los índices de forma aleatoria y sin repetición.
3. Mientras se produzcan cambios en el clúster:
  3.1. Para cada índice, mirar qué incremento supone en la infeasibility al asignarlo a un clúster.
Tomar el menor de estos.
  3.2. Actualizar los centroides
```



En la sección Análisis de resultados comprobaremos cómo de buena es la solución obtenida.

Búsqueda local

Búsqueda local es la primera metaheurística que programaremos. Aunque es un algoritmo conceptualmente sencillo, supone una mejora en ciertos aspectos con respecto a Greedy. Se basa en la exploración de vecinos a la solución actual, tomando el mejor de entre los posibles. Por su naturaleza, suele generar óptimos locales.

Descripción del algoritmo

Como hemos citado, se exploran los vecinos de una cierta solución, mirando en cada iteración cuáles son las mejores soluciones. Se define un vecino como la asignación de clúster c al $i-ésimo$ elemento del espacio partiendo de una solución actual. Debemos verificar que la posible solución generada con este operador es válida, pues en otro caso, no tiene sentido seguir.

El concepto de *mejor solución* es el que proporciona el fitness. Es decir, en cada iteración, se comprobará si el vecino tiene un fitness menor que el actual. Si es así, la siguiente solución a explorar será esta.

Implementación

El pseudocódigo del algoritmo es el siguiente:

```
Generar una solución válida inicial.

Hasta que no se haya alcanzado un óptimo local
├ Guardar la información de la solución actual relevante: fitness, infeasibility,
├ Barajar los índices {0, ..., num_elementos}
│
├ Para i en los índices barajados
│  ├ Barajar los clústers {1, ..., num_clusters}
│  │
│  ├ Para c en los clústers barajados
│  │  ├ Si el clúster del i-ésimo elemento no es c
│  │  │  ├ Comprobar si el vecino nuevo es válido
│  │  │  ├ En ese caso, comprobar si tiene un fitness menor.
│  │  │  └ Si es así, reexplorar todos los índices de nuevo y actualizar la información de la solución
nueva.
│  │  │
└  └  └─ Si se ha encontrado una nueva solución, ignorar el resto de índices.
```

Las operaciones más relevantes de este algoritmo las realizamos en la función `bl_fitness_posible_sol(i, c, antiguo_infeasibility)`. Por cómo está gestionado internamente el clúster, es mucho más rápido verificar que la solución es válida dentro del propio clúster, y no fuera. Esta función se encarga de asignar temporalmente el vecino, comprobar si es válido, y en caso de serlo, devolver qué fitness produciría. El motivo de que necesite el antiguo infeasibility es por eficiencia. En vez de calcular todo el sistema para cada vecino nuevo, se calcula de la siguiente manera:

Como mencionamos en uno de los apartados anteriores, el cálculo de esta delta es muchísimo más rápido que el de todo el sistema. En la siguiente sección comprobaremos cuánto tarda en total un benchmark.

```
    Finished release [optimized] target(s) in 5.62s
     Running `target\release\metaheuristicas.exe zoo 20 bl`
Comienza la lectura de los archivos
        ‣ Se empieza a leer el archivo "./data/PAR/zoo_set.dat"
        ‣ Se empiezan a leer las restricciones "./data/PAR/zoo_set_const_20.const"
Finalizada la lectura del cluster ✓

‣ Ejecutando búsqueda local para el cálculo de los clusters
‣ Cálculo del cluster finalizado en 200 ms ✓

Información del cluster:
Información del cluster:
            ‣ Número de clusters: 7
            ‣ Lista con los clusters: [5, 5, 6, 1, 4, 7, 4, 5, 1, 5, 7, 4, 7, 5, 3, 4, 7, 4, 5, 4, 1, 5, 4, 5, 6, 1, 4, 6, 5, 5, 5, 1, 5, 4, 4, 5, 4, 5, 2, 5, 7, 6, 6, 1, 4, 5, 5, 6, 4, 5, 1, 5, 6, 4,
4, 1, 6, 2, 7, 5, 1, 1, 6, 4, 5, 5, 4, 5, 2, 5, 6, 5, 4, 5, 5, 6, 1, 4, 5, 7, 5, 7, 7, 5, 5, 5, 5, 6, 5, 6, 5, 7, 5, 5, 5, 1, 5, 5, 1, 4, 1]
            ‣ Elementos en cada cluster: [14, 3, 1, 20, 40, 13, 10]
            ‣ Número de restricciones: 912
            ‣ Elementos en el espacio: 101
            ‣ Centroides:

  ┌                     ┐
  │                   0 │
  │                   0 │
  │  0.8571428571428571 │
  │                   0 │
  │                   0 │
  │                 0.5 │
  │  0.8571428571428571 │
  │                 0.5 │
  │  0.5714285714285714 │
  │  0.6428571428571428 │
  │  0.3571428571428571 │
  │                   0 │
  │               0.375 │
  │                 0.5 │
  │                   0 │
  │ 0.14285714285714285 │
  └                     ┘
```

Práctica 2: algoritmos genéticos y meméticos

Conceptos básicos de los algoritmos genéticos y meméticos

En esta segunda práctica, realizaremos una versión adaptada de los algoritmos **genéticos** y **meméticos**. Los meméticos tienen como base un algoritmo genético al que se le introduce una fase de mejora de las soluciones presentes en cada generación. Por tanto, presentaremos primero los conceptos básicos para ambos.

Los genéticos introducen un conjunto de soluciones llamadas **cromosomas**. Cada componente de un cromosoma se denomina **gen**. Al conjunto de todos los cromosomas se le conoce como **población**.

La idea de estos algoritmos es que los cromosomas evolucionan, de forma que cada **generación** produce unos descendientes atendiendo a una serie de pasos ordenados: **selección**, **cruce**, **mutación** y **reemplazamiento**.

La forma en la que se realizan estas fases se define mediante los **operadores**.

Operadores

Los operadores son funciones que reciben una o dos soluciones y producen otra. Cada paso tiene su operador específico, y todos poseen algún componente de aleatoriedad.

En la implementación, únicamente los operadores de cruce se han separado. El resto se encuentran incrustados en el código del algoritmo genético principal.

Operadores de selección

El operador de selección será un **torneo binario**. Enfrentaremos dos cromosomas para ver quién tiene mejor fitness, y nos quedaremos con ese.

La selección tendrá un componente de aleatoriedad en todos los algoritmos genéticos. Elegiremos cierto número de cromosomas dependiendo del modelo en el que nos encontremos, y los emparejaremos al azar.

El pseudocódigo es el siguiente:

```
operador de selección(p1, p2):
    Si fitness(p1) < fitness(p2)
        p1
    En otro caso
        p2
```

Operadores de cruce

Los operadores de cruce reciben dos cromosomas, y producen un nuevo hijo a partir de sus genes. En estas prácticas usamos dos tipos:

Cruce uniforme

El operador de cruce uniforme toma la mitad de los genes de un padre, la otra mitad del otro padre, y los combina en un hijo.

La implementación es la siguiente:

```
cruce_uniforme(p1, p2):
    descendencia: Vector del mismo tamaño que p1 y p2
    genes_a_copiar: Vector vacío

    Para _ en 0 .. p1.len()/2
        loop
            pos_gen = generar aleatorio en [0, p1.len())
            Si genes_a_copiar no contiene a pos_gen
                genes_a_copiar.push(pos_gen)
                break

    Para i en genes_a_copiar
        descendencia[i] = p1[i]

    Para i en 0 .. descendencia.len()
        Si descendencia[i] == 0
            descendencia[i] = p2[i]

    descendencia
```

Cruce de segmento fijo

El operador de cruce de segmento fijo determina un fragmento de tamaño aleatorio y un inicio, de forma que copia los genes de `p1` desde el inicio hacia delante, *dando la vuelta* por el principio del cromosoma si fuera necesario.

Para el resto de genes todavía no determinados, se procede de manera análoga al cruce uniforme: se coge la mitad de los genes de un padre, la otra mitad del otro, y se traspasan.

Debemos destacar que este operador está claramente sesgado, pues siempre se copian más genes del primer cromosoma que del segundo.

```
cruce_segmento_fijo (p1, p2):
    descendencia: Vector del mismo tamaño que p1 y p2
    inicio_segmento = aleatorio en [0, p1.len())
    tamaño_segmento = aleatorio en [0, p1.len())

    i = inicio_segmento
    copias = 0

    Mientras que copias < tamaño_segmento
        descendencia[i] = p1[i]

        i = (i+1) % p1.len()
        copias = copias + 1

    inicio = min (
        (inicio_segmento + 1)%p1.len(),
        (inicio_segmento + tamano_segmento + 1)%p1.len()
    )
    fin = max (
        (inicio_segmento + 1)%p1.len(),
        (inicio_segmento + tamano_segmento + 1)%p1.len()
    )

    genes_a_copiar: Vector vacío

    Para _ en 0 .. fin - inicio + 1
```

```
        loop
            pos_gen = aleatorio en [inicio, fin]

            Si genes_a_copiar no contiene a pos_gen
                genes_a_copiar.push(pos_gen)
                break

    Para i en genes_a_copiar
        descendencia[i] = p1[i]

    Para i en 0 .. descendencia.len()
        Si descendencia[i] == 0
            descendencia[i] = p2[i]

    descendencia
```

Operadores de mutación

Este operador elige un cromosoma al azar de la población, y muta un gen aleatorio manteniendo la validez de la solución. No obstante, esto es únicamente el operador. La elección de cuántas mutaciones se deben hacer en la población total dependen de una serie de parámetros.

La implementación se verá en el pseudocódigo del algoritmo completo.

Otras consideraciones

Reparación

A veces, los operadores de cruce no generan soluciones válidas, debido a que se pueden dejar clústers vacíos. Para ello, se le aplica una reparación, descrita por el siguiente pseudocódigo:

```
reparar (hijo, k):
    recuento: Vector de tamaño k inicializado a 0

    Para c en hijo
        recuento[c - 1] = recuento[c-1] + 1

    Para indice en 0 .. recuento.len()
        Si recuento[indice] == 0
            loop
                i = aleatorio en [0, hijo.len())

                Si recuento[hijo[i]-1] > 1
                    recuento[hijo[i]-1]--
                    hijo[i] = indice + 1
                    recuento[indice]++
                    break
```

Recordemos que los clústers toman valores en $[1, k]$.

Generación aleatoria de la población inicial

La mayor parte de los algoritmos requieren generar una población aleatoria inicial. Para ello, hacemos lo siguiente:

```
Para _ en 0 .. tamano_poblacion
    solucion_inicial: Vector de tamaño numero_genes inicializado a 0

    Mientras que solucion_inicial no sea válida
        Para c en solucion_inicial
            c = aleatorio en [0, k]

    poblacion.push(solucion_inicial)
```

Algoritmos genéticos considerados

Implementaremos 4 tipos de algoritmos genéticos en total, que surgirán de combinar operadores y modelos de reemplazamiento. Estos son:

- Genético generacional con operador de cruce uniforme (**AGG_UN**).
- Genético generacional con operador de cruce de segmento fijo (**AGG_SF**).
- Genético estacionario con operador de cruce uniforme (**AGE_UN**).
- Genético estacionario con operador de cruce de segmento fijo (**AGE_SF**).

Todos estos algoritmos dependen de unos determinados parámetros, los cuales son:

- **Tamaño de la población**: cuántos individuos existen al final de un ciclo. Por defecto, consideramos 50.
- **Número de genes** que tiene un cromosoma. Depende del dataset.
- **Evaluaciones del fitness máximas**. Por defecto 100000.
- **Número de cromosomas que se desarrollan**. Lo llamaremos m. Depende del esquema de reemplazamiento.
- **Probabilidad de cruce**. Depende del esquema de reemplazamiento.
- **Número de cruces esperado** = `probabilidad del cruce * m / 2`. Consideramos un único cruce al del cromosoma i con $i+1$, así como el del $i+1$ con i. Los motivos son de eficiencia, pues en la selección, ya se considera que es aleatoria.
- **Operador de cruce**.
- **Probabilidad de mutación** = `0.1/número de genes`.
- **Número de mutaciones** = `probabilidad de mutación * m * número de genes`. Mutaremos considerando la población como una matriz, y eligiendo una entrada al azar.

Esquemas de reemplazamiento

El esquema de reemplazamiento refleja cómo se procesa la generación actual, y cuántos descendientes se generan. Como hemos citado antes, usamos el modelo **generacional** y **estacionario**

El **modelo generacional** considera para el desarrollo de una generación el mismo número de cromosomas que el de la población. En nuestro caso, esto significa que `m = tamaño de la población` y que la probabilidad de cruce es de 0.7.

El **modelo estacionario** toma dos individuos aleatorios y los procesa, reintroduciéndolos finalmente en la población. Se eliminarán los dos peores cromosomas al final de este proceso.
Por tanto, $m = 2$, y la probabilidad de cruce es de 1.

Implementación

Los 4 tipos de genéticos resultarán de cambiar los parámetros de la llamada de la función principal. Por tanto, solo consideraremos el pseudocódigo de ésta.

En la llamada no se ha tenido en cuenta la semilla.

```
genetico(cluster, modelo, operador_cruce):
   tamano_poblacion = 50
   numero_genes = cluster.num_elementos
   max_evaluaciones_fitness = 100_000
   m =
      2                si modelo == estacionario,
      tamano_poblacion si m == generacional

   probabilidad_cruce =
      0.7 si modelo == estacionario,
```

```
   1.0 si modelo == generacional

numero_cruces = (probabilidad_cruce * m/2).floor()

probabilidad_mutacion = 0.1/numero_genes
numero_mutaciones = (probabilidad_mutacion * m * numero_genes).ceil()

rango_clusters = distribución uniforme en [1, clusters.num_clusters]
rango_poblacion = distribución uniforme en [0, tamano_poblacion)
rango_m = distribución uniforme en [0, m)
rango_genes = distribución uniforme en [0, numero_genes)

Poblacion: Vector de tamaño tamano_poblacion
fitness_poblacion: Vector de tamaño tamano_poblacion

Generar la población inicial y evaluar su fitness

t = 0
evaluaciones_fitness = 0

Mientras que evaluaciones_fitness < max_evaluaciones_fitness

// ———————————————————————————————————— SELECCION ————

    p_padres: Vector nuevo vacío
    combate: par de números entero

    Para _ en 0 .. m
        combate = (aleatorio en rango_poblacion, aleatorio en rango_poblacion)

        Si fitness(poblacion[combate.0]) < fitness(poblacion[comabte.1])
            p_padres.push(poblacion[combate.0])
        En otro caso
            p_padres.push(poblacion[combate.1])

// ———————————————————————————————————— CRUCE ————

     p_intermedio: Vector nuevo vacío

     cruces_restantes = numero_cruces

     Para i en 0 .. m
        Si cruces_restantes > 0
            hijo: Vector nuevo vacío

            Si i%2 == 0 y i < m
                hijo = operador_cruce(p_padres[i], p_padres[i+1])
            En otro caso
                hijo = operador_cruce(p_padres[i], p_padres[i-1])
                cruces_restantes--

            Si hijo no es un cromosoma válido
                reparar(hijo)

            p_intermedia.push(hijo)

        En otro caso
            p_intermedia.push(p_padres[i])

// ———————————————————————————————————— MUTACION ————

    p_hijos = p_intermedia
    i = 0

    Para _ en 0 .. numero_mutaciones
        i = aleatorio en rango_m

        loop
            gen_a_mutar = aleatorio en rango_genes
```

```
            antiguo_cluster = p_hijos[i][gen_a_mutar]
            p_hijos[i][gen_a_mutar] = aleatorio en rango_clusters

            Si p_hijos[i] no es una solución válida
                p_hijos[i][gen_a_mutar] = antiguo_cluster
            En otro caso
                break

// ———————————————————————————————————————— REEMPLAZAMIENTO ————

    Si el modelo es el estacionario
        Para i en 0 .. m
            fitness_poblacion.push(fitness(p_hijos[i]))
            poblacion.push(p_hijos[i])
            evaluaciones_fitness++

        // Ordenar de menor a mayor
        Para i en 0 .. fitness_poblacion.len()
            Para j en 0 .. fitness_poblacion.len() - i - 1
                Si fitness_poblacion[j+1] < fitness_poblacion[j]
                    fitness_poblacion.swap(j, j+1)
                    poblacion.swap(j, j+1)

        Para _ en 0 .. m
            poblacion.pop()
            fitness_poblacion.pop()

    Si el modelo es el generacional
        posicion_mejor = 0
        mejor_fitness = máximo f64 posible

        Para (i, valor) en fitness_poblacion.enumerate()
            Si valor < mejor_fitness
                mejor_fitness = valor
                posicion_mejor = i

        mejor_cromosoma_antiguo = poblacion[posicion_mejor]
        poblacion = p_hijos

        Para (i, cromosoma) en poblacion.enumerate()
            fitness_poblacion[i] = fitness(cromosoma)

        evaluaciones_fitness = evaluaciones_fitness + m

        posicion_peor = 0
        peor_fitness = 0.0

        Para (i, valor) en fitness_poblacion.enumerate()
            Si valor > peor_fitness
                peor_fitness = valor
                posicion_peor = i

        poblacion[posicion_peor] = mejor_cromosoma_antiguo
        fitness[posicion_peor] = mejor_fitness

    t = t+1

posicion_mejor = 0
mejor_fitness = máximo f64 posible

Para (i, valor) en fitness_poblacion.enumerate()
    Si valor < mejor_fitness
        mejor_fitness = valor
        posicion_mejor = i

cluster.asignar_clusters(poblacion[posicion_mejor])

cluster
```

Algoritmos meméticos

Los **algoritmos meméticos** son algoritmos genéticos a los que se les introduce una fase de exploración del entorno cada ciertas generaciones. De esta forma, se optimiza localmente de forma periódica.

De base utilizaremos el algoritmo genético generacional con operador de cruce uniforme. Por tanto, los parámetros serán los mismos.

Aparte, debemos considerar algunos nuevos:

- **Periodo generacional**: cada cuántas generaciones se aplica el optimizador local. Valdrá 10.
- **Probabilidad**: para cada cromosoma, cuál es la probabilidad de que se le aplique el optimizador.
- **Solo a mejores**: si este parámetro está activo, únicamente se aplicará a los `probabilidad * número de cromosomas` mejores soluciones de la población.

Consideraremos las siguientes versiones:

- **AM_10_1**: `probabilidad = 1`, `solo_a_mejores = false`
- **AM_10_01**: `probabilidad = 0.1`, `solo_a_mejores = false`
- **AM_10_01_mejores**: `probabilidad = 0.1`, `solo_a_mejores = true`

Búsqueda local suave

Para el PAR utilizaremos como optimizador local una búsqueda local suave, descrita de la siguiente manera:

```
busqueda_local_suave(solucion, cluster, fallos_permitidos):
   indices_barajados: Vector con sus componentes inicializadas a 0 .. solucion.len()

   indices_barajados.shuffle()

   fallos = 0
   evaluaciones_fitness = 0
   mejora = true
   i = 0

   Mientras (mejora || fallos < fallos_permitidos) && i < solucion.len()
       mejora = false

       mejor_fitness = fitness(solucion)
       mejor_cluster = solucion[indices_barajados[i]]

       Para c en 1 ..= cluster.num_clusters
           Si c != mejor_cluster
               solucion[indices_barajados[i]] = c

               Si solución es válida
                   fitness_actual = fitness(solucion)
                   evaluaciones_fitness++

                   Si fitness_actual < mejor_fitness
                       mejor_cluster = c
                       mejor_fitness = fitness_actual
                       mejora = true
                   En otro caso
                       solucion[indices_barajados[i]] = mejor_cluster
               En otro caso
                   solucion[indices_barajados[i]] = mejor_cluster

       Si !mejora
           fallos = fallos + 1

       i++
```

```
    evaluaciones_fitness
```

Como propuesta de optimización, se podría modificar la devolución para incluir el fitness generado. Por ejemplo, en Rust, se devolvería el par `(evaluaciones_fitness, mejor_fitness)`.

Implementación

```
memetico(cluster, periodo_generacional, probabilidad, solo_a_mejores):
    // Mismos pasos que AGG_UN
    ...

    fallos_maximos = 0.1 * numero_genes

    ...

    Mientras que evaluaciones < max_evaluaciones_fitness
        Si t % periodo_generacional == 0 && t > 0
            Si solo_a_mejores
                busquedas_totales = probabilidad * poblacion.len()

                Ordenar poblacion y fitness_poblacion de menor a mayor

                Para i en 0 .. busquedas_totales
                    evaluaciones = BLS(poblacion[i], cluster, fallos_maximos)
                    fitness_poblacion[i] = fitness(poblacion[i])
                    evaluaciones_fitness = evaluaciones_fitness + evaluaciones
            En otro caso
                Para i en 0 .. tamano_poblacion
                    Si aleatorio en [0, 1] <= probabilidad
                        evaluaciones = BLS(poblacion[i], cluster, fallos_maximos)
                        fitness_poblacion[i] = fitness(poblacion[i])
                        evaluaciones_fitness = evaluaciones_fitness + evaluaciones


// ———————————————————————————————————————— SELECCION ————
        ...
```

Debemos destacar un aspecto de implementación muy importante: ¿**por qué no aplicamos la BLS después de mutar**? Esta decisión se ha tomado debido al algoritmo que hay de base: el genético generacional.

En él, tras aplicar la mutación, se produce el reemplazamiento. Esta fase sustituye todo lo que se encuentre en la generación t para pasar a la generación $t + 1$, calculando justo después el fitness de todos los miembros. Antes del reemplazamiento, se elige al mejor miembro de la generación t y se elimina al peor que han producido los hijos.

La diferencia entre aplicar la BLS antes o después del reemplazamiento es que se optimice o no el mejor de la generación t. En la práctica, este método va a producir diferencias insignificativas. Además, personalmente me parece más intuitivo y sencillo de programar al hacerlo al inicio de la generación.

Si el algoritmo de base fuera el genético estacionario, esta decisión no tendría sentido, pues deberíamos tratar únicamente los hijos que se están desarrollando. Sin embargo, no es el caso.

Siguiendo la propuesta de la sección anterior, se podría modificar la implementación para tener en cuenta el fitness que produce la búsqueda local suave. No obstante, la convergencia es suficientemente rápida como para que esto no afecte en gran medida.

Análisis de resultados

En esta sección discutiremos los resultados obtenidos por todos algoritmos. Presentaremos los parámetros de los datasets utilizados, las distancias óptimas generadas por el Greedy suponiendo que no hay restricciones, y, lo más importante, **cómo rinden nuestros algoritmos**.

Recordamos que los resultados se pueden obtener en cualquier momento ejecutando `cargo run --release benchmark`. Para más información, consultar la sección Benchmark de uno o más algoritmos

Descripción de los casos del problema empleados

Los datasets usados reciben el nombre de `Zoo`, `Glass`, y `Bupa`. Los dos primeros presentan una dificultad similar, mientras que el último requiere de un mayor tiempo de cómputo.

	Zoo	**Glass**	**Bupa**
Atributos	`16`	`9`	`5`
Clústers	`7`	`7`	`16`
Instancias	`101`	`214`	`345`
Distancia óptima generada por Greedy	`0.9048`	`0.36429`	`0.229248`

Sobre estos conjuntos se ha impuesto un número de restricciones: al 10% y al 20%. Naturalmente, cuantas más restricciones, más costoso resulta computar una solución aceptable, pues el sistema contempla una mayor cantidad de enlaces entre sus elementos.
Los ficheros ubicados en `./data/PAR` guardan la información sobre los datasets.

Benchmarking y resultados obtenidos

Cada algoritmo se ha ejecutado 5 veces por dataset. Como tenemos 3 datasets y 2 restricciones para cada uno, nos encontramos un total de 30 ejecuciones por algoritmo. Estudiaremos las siguientes medidas:

- **Tasa de infeasibility**: número de restricciones que se incumplen en la solución.
- **Desviación media intraclúster**: mide cómo de cohesionados están nuestros clústers. Cuanto más bajo es este valor, mejor.
- **Agregado**: el fitness de la solución. Cuanto más bajo, mejor.
- **Tiempo de ejecución** (ms).

Se ha utilizado un ordenador con un i7 4790 @ 3.6GHz con turbo a 4Ghz, así como un i5 8250U @ 1.60 GHz con turbo 3.40 GHz. Dado que el rendimiento single core es muy similar entre ambas arquitecturas así como la velocidad base de ambas CPUs, no se aprecian diferencias muy significativas entre los tiempos de ejecución (diferencia de 2 segundos como mucho medido a ojo).

Todos los resultados han sido ordenados de menor a mayor fitness, por lo que resultará más fácil distinguir cuáles son los algoritmos que mejor rinden.

Resultados de cada dataset

Únicamente consideraremos la media de cada algoritmo en estas tablas. Si se desea consultar toda la información, se encuentran en la carpeta `./data/csv/[dataset]/[algoritmo]`.

Zoo 10

Algoritmo	Infeasibility	Desviación media intraclúster	Fitness	Tiempo (ms)
am_10_01_mejores	8	0.59921	0.65983	3512
am_10_1	7	0.65485	0.70335	3243
bl	12	0.63444	0.72082	131
am_10_01	9	0.67180	0.73697	2283
agg_un	13	0.66746	0.76597	2293
age_sf	10	0.69406	0.76983	2634
agg_sf	12	0.69145	0.78239	2351
age_un	13	0.69109	0.78657	2342
greedy	2	0.94421	0.95330	1

Zoo 20

Algoritmo	Infeasibility	Desviación media intraclúster	Fitness	Tiempo (ms)
am_10_1	17	0.68887	0.75590	3232
am_10_01	14	0.71040	0.76612	2450
am_10_01_mejores	14	0.73101	0.78674	2910
agg_sf	24	0.70817	0.80266	2516
bl	23	0.71965	0.81010	104
agg_un	22	0.73430	0.82233	2354
age_sf	24	0.73092	0.82460	2769
age_un	24	0.72944	0.82474	2537
greedy	2	0.98711	0.99519	1

Glass 10

Algoritmo	Infeasibility	Desviación media intraclúster	Fitness	Tiempo (ms)
am_10_01	50	0.19115	0.23956	6190
am_10_01_mejores	39	0.21122	0.24881	10067
age_un	48	0.20197	0.24901	6438
age_sf	52	0.20031	0.25069	7208
bl	36	0.21812	0.25276	424
am_10_1	35	0.21962	0.25406	7811
agg_un	47	0.22859	0.27405	6380
agg_sf	69	0.24371	0.31102	7261
greedy	4	0.37869	0.38262	2

Glass 20

Algoritmo	Infeasibility	Desviación media intraclúster	Fitness	Tiempo (ms)
am_10_01	81	0.21930	0.26126	8869
am_10_1	79	0.22282	0.26374	9483
age_un	50	0.24254	0.26857	7762
age_sf	58	0.23903	0.26891	8608
am_10_01_mejores	55	0.24246	0.27088	10811
agg_un	45	0.24809	0.27141	8029
bl	57	0.24204	0.27151	374
agg_sf	125	0.23334	0.29821	8249
greedy	1	0.34667	0.34677	2

Bupa 10

Algoritmo	Infeasibility	Desviación media intraclúster	Fitness	Tiempo (ms)
bl	133	0.11054	0.14618	6039
am_10_01_mejores	140	0.11622	0.15352	18042
age_un	136	0.11994	0.15639	14165
am_10_01	140	0.12113	0.15859	16888
age_sf	196	0.12354	0.17590	16229
am_10_1	261	0.15718	0.22707	16457
greedy	29	0.22779	0.23546	16
agg_un	617	0.16441	0.32955	14505
agg_sf	695	0.16744	0.35353	14975

Bupa 20

Algoritmo	Infeasibility	Desviación media intraclúster	Fitness	Tiempo (ms)
bl	211	0.11492	0.14434	4414
am_10_01_mejores	214	0.11815	0.14793	22189
am_10_01	248	0.11787	0.15237	22598
age_un	268	0.11819	0.15556	18146
age_sf	282	0.12199	0.16120	20199
am_10_1	465	0.14550	0.21023	20298
greedy	10	0.23299	0.23427	6
agg_un	1179	0.16472	0.32901	18233
agg_sf	1264	0.16978	0.34600	18866

Síntesis

Puesto que ya tenemos todas las tablas sintetizadas, es hora de sintetizar los valores que nos han salido.

Claramente podemos observar que los **algoritmos meméticos producen los mejores resultados**. En 4 de los 6 casos, un memético ha producido el menor fitness. Debemos notar que los distintos parámetros afectan de forma diferente a cada dataset. Por ejemplo, **AM_10_01 genera mejores resultados en Glass**, mientras que en **Zoo funcionan bien tanto AM_10_1 como AM_10_01_mejores**.

Este gran rendimiento viene acompañado de un incremento en los tiempos de ejecución. Además, **AM_10_01_mejores tarda considerablemente más que el resto de meméticos**. Esto podría ser debido a la reordenación de la población. Como en la implementación se ha usado un bubble sort, podríamos usar uno más eficientes para acelerar esta función.

Hablemos ahora de los algoritmos genéticos. Lo primero que resulta evidente es que **el modelo generacional rinde particularmente mal**. Tanto AGG_UN como AGG_SF suelen encontrarse por la parte baja de la tabla, a excepción del dataset Zoo. En particular, en Bupa rinden peor que Greedy; hasta 0.12 puntos mayor. Por contrapartida, el **modelo estacionario bastante bien**. Excepto en Zoo, se posiciona mejor que la búsqueda local.

Búsqueda local sigue siendo muy consistente entre todos los datasets. Esto podría deberse a cómo se han organizado los datos: dado que son conjuntos preparados, lo normal es que los mínimos locales se encuentren agrupados en zonas habituales del espacio. En estos casos, un optimizador local haya soluciones de manera eficiente.

Por último, **Greedy K-Medias es el algoritmo que peor rinde de todos en relación al fitness**. Aunque es rapidísimo, sus resultados suelen encontrarse por la parte baja. No obstante, por su construcción, se encarga de minimizar primero el número de restricciones violadas. Por lo que en los casos particulares en los que interese tanto rapidez de ejecución como minimización de infeasibility, podría considerarse una elección sólida.

Hasta ahora nos hemos centrado únicamente en el fitness. Mirando el infeasibility, podemos ver que sale un alto número de violaciones en general comparado con Greedy. Por ejemplo, en Bupa 10:

Algoritmo	Infeasibility	Desviación media intraclúster	Fitness	Tiempo (ms)
bl	133	0.11054	0.14618	6039
...				
greedy	29	0.22779	0.23546	16

Esto nos induce a pensar que **resulta más beneficioso tener clústers cohesionados que intentar minimizar las violaciones**. Esto explicaría el mal rendimiento en general de Greedy, atendiendo a nuestra definición de la función fitness.

Para mejorar el rendimiento de los algoritmos genéticos generacionales, se puede cambiar el parámetro que controla la probabilidad de mutación. **Usando** $1/(\text{número de genes})$ **en vez de** $0.1/(\text{número de genes})$ arroja los siguientes resultados en Bupa 10:

Algoritmo	Fitness con probabilidad `1/número de genes`	Fitness con probabilidad `0.1/número de genes`
age_un	`0.1678`	`0.15639`
age_sf	`0.1816`	`0.16120`
agg_un	`0.2744`	`0.32955`
agg_sf	`0.2770`	`0.35353`

Claramente vemos cómo todos los genéticos consiguen un fitness considerablemente menor. Esto se mantiene para todos los datasets.

Por último, debemos destacar la rápida convergencia del límite inferior de la sucesión de fitness. Aunque no se estudiará de manera explícita, merece la pena mencionarlo. Estudiando la sucesión de máximo y mínimo fitness de todos los algoritmos genéticos, vemos que tras un cierto número de generaciones, el resultado se estabiliza. Esto nos indica que el número de evaluaciones del fitness es más que suficiente para garantizar la convergencia. Usando AGG_Un para Bupa 10, y tomando puntos aleatorios:

Generación	Peor fitness	Mejor fitness
`51`	`0.5014`	`0.4993`
`596`	`0.4206`	`0.4192`
`1058`	`0.3853`	`0.3842`
`1577`	`0.3619`	`0.3599`
`1999` (última generación)	`0.3455`	`0.3368`

Podemos ver que, conforme avanzan las generaciones, la mejora se reduce. Aunque depende del algoritmo y del dataset, la mejora que obtendríamos al aumentar el número de evaluaciones podría ser incluso nula. Por ello, podemos asegurar que no desperdiciamos tiempo de procesamiento.

Esto concluye el desarrollo de la práctica 2. Llegados a este punto, y tras hacer un estudio de todos nuestros algoritmos, **resulta difícil quedarnos con únicamente uno**. Todos tienen sus ventajas e inconvenientes, y se comportan de forma distinta en nuestros datasets. Sin embargo, es digno mencionar la consistencia de los meméticos. Ajustando parámetros, hemos conseguido que rindan mejor que ninguno, aunque a veces la diferencia sea ínfima.

En la siguiente práctica, introduciremos enfriamiento simulado y multiarranque. ¡Quedarnos con únicamente uno podría ser incluso más difícil!

Referencias

- El libro oficial de Rust
- Imagen de la primera portada
- Imagen de la segunda portada
- Documentación de Nalgebra
- Documentación de Multimap
- StackOverflow
- Material de teoría y de prácticas