UNIVERSIDAD DE GRANADA

ETSIIT
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Práctica 2: Técnicas de Búsqueda basadas en Poblaciones para el Problema del Agrupamiento con Restricciones

Metaheurísticas

Grupo A3 — Lunes de 17:30 a 19:30

Autor:
Lugli, Valentino Glauco · YB0819879 · valentinolugli@correo.ugr.es

Índice

1. Descripción del Problema

Esta práctica aborda el Problema del Agrupamiento con Restricciones, siendo este una generalización del problema del Agrupamiento Clásico. Este problema tiene como objetivo clasificar un conjunto de datos u observaciones en un grupo definido de clusters o agrupamientos de acuerdo a posibles similitudes entre los datos. Es un problema de aprendizaje no supervisado que permite descubrir grupos inicialmente desconocidos en un conjunto de datos, también permite el agrupamiento de objetos similares entre sí.

La variante que se trabajará, llamada con Restricciones o Contrained Clustering, generaliza el anterior problema añadiendo restricciones, de esta manera el problema se vuelve de aprendizaje semi-supervisado en donde se tiene un conjunto de datos X con n instancias, donde se desea encontrar una partición C del mismo que minimice la desviación general y cumpla con las restricciones del conjunto llamadas R.

Se utilizarán restricciones de instancias para R, es decir, dada una pareja de instancias se establecerán dos tipos de restricciones: Must-Link, en la que se indica que ese par de instancias deben pertenecer a un mismo cluster y Cannot-Link, que indica que ese par no debe pertenecer al mismo cluster. Se interpretarán las restricciones de manera débil, es decir que se desea minimizar el número de restricciones incumplidas pero se pueden incumplir algunas.

2. Descripción de los algoritmos empleados

2.1. Consideraciones generales

En esta práctica se realizaron 4 variantes de algoritmos genéticos y 3 variantes de algoritmos meméticos.

Los algoritmos representan la solución de manera similar, hacen uso de un vector de enteros de una longitud n en donde se indica que para la i-ésima instancia en ese vector, esta está asociada a un cluster j, con $j \in \{0, .., k-1\}$.

Los datos referentes a las instancias están almacenados para ambos problemas en una matriz X de tamaño $n \times d$ y hacen uso de, o bien una matriz de restricciones R denominada MR de tamaño $n \times n$ o bien de un vector de restricciones que internamente utiliza un tipo de dato abstracto sencillo denominado `triplet` el cual contiene un par de índices de la matriz R y que clase de restricción poseen.

De igual manera, se creó una clase denominada `cromosome` el cual mantiene el vector de enteros mencionado anteriormente, junto con un vector de contadores el cual existe para llevar un conteo de cuantos genes posee asociado un cluster, también se tiene almacenado el valor calculado del fitness y un booleano que indica si el cromosoma ha cambiado. Esta clase fue utilizada dentro de un vector para representar la población de individuos del problema. Se tiene también otro dato abstacto que se utilizó para mantener una copia al mejor cromosoma de la población, esto se utiliza en todas las variantes del algoritmo.

2.2. Función Objetivo y Operadores Comunes

2.2.1. Registro triplet

```
registro triplet
    entero: x_0
    entero: x_1
    entero: restriccion
fregistro
```

2.2.2. Registro bestCromosome

```
registro bestCromosome
    entero: posicion // Utilizada para depurar y para seleccionar de la población
       en general.
    flotante: fitness
    booleano: estaVivo // Cambia con las mutaciones y el cruce.
    cromosome: datos // Guarda una copia del cromosoma.
fregistro
```

2.2.3. Clase cromosome

```
clase cromosome
    atributos:
        privados:
            vector de entero: genes
            vector de entero: cuentaDeClusters /*Para cada cluster i, cuantas
               instancias están asignados a él*/
            flotante: fitness
            booleano: tieneCambio
    metodos:
        publicos: /* Se obvian constructores y destructores */
            func obtenerGenes(): vector de entero
            func tamañoGen(): entero
            func obtenerCluster(entero: gen): entero
            func obtenerFitness(): flotante
            func obtenerCambios(): booleano
            func obtenerClusters(): vector de entero
            func obtenterCuentaDeCuster(entero: gen): entero
            proc asignarGen(vector de entero: argGen)
            proc asignarCluster(vector de entero: argCluster)
            proc asignarFitness(flotante: argFitness)
            proc asignarCambios(booleano: argCambios)
            proc cambiarGen(entero: gen, entero: cluster)
            /* ... y más funciones ...*/
fclase
```

2.2.4. Evaluación de la Función Objetivo

La función para evaluar los algoritmos es exactamente la misma que fue utilizada en el algoritmo de Búsqueda Local, se trata de una función principal llamada solamente `fitness`, la cual se utiliza para unir las dos partes de la misma, la distancia instracluster promedio sumado a la infeasibilidad por el lambda.

```
funcion fitness(vector de entero: S, vector de vector de flotante: X, vector de
   triplet: ML, vector de triplet: CL, flotante: lambda, entero: k): flotante
inicio:
    retornar intraClusterDistance(S, X, k) + (lambda * infeasibility(S, ML, CL))
ffunc
```

Para obtener la distancia intracluster promedio, se necesita saber que instancias están en qué clusters, para ello lo primero que se realiza es calcularlas, para el cluster `k`, se llama a la función `calcularInstancia()`, internamente esta función va almacenando que instancias del vector `S` se encuentran asociadas al cluster `i`, estas se retornan y se almacenan en `instanciasDeCluster`, luego, con estas instancias se promedian sus coordenadas en todas las dimensiones para obtener la posición del cluster `i`, esto se almacena en el vector `coords` y finalmente, se realiza el cálculo de la distancia intracluster de ese cluster con sus elementos asociados, y cuando se han realizado todos los clusters, se devuelve el promedio de todos ellos.

```
funcion intraClusterDistance(vector de entero: S, vector de vector de flotante: X,
   entero: k): flotante
variables:
    vector de entero: instanciasDeCluster
    vector de flotante: coords, actDist, total <- 0
inicio:
    para i <- 0 hasta k hacer
        // Obtener las instancias asociadas a un cluster.
        instanciasDeCluster <- calcularInstancia(X, i, S)
        // Obtener las coordenadas del centroide de un cluster.
        coords <- calcularCoordenadasCentroide(instancia, X)
        //Obtener la distancia intracluster de un cluster.
        actDist <- calcIntraDist(coords, instancia, X)
        total += actDist
    fpara

    retornar totalDist / k
ffunc
```

Como se ha comentado, se realiza aquí la media de las diferencias para un cluster en particular, en la función principal se realiza es el promedio de las distacias de todos los clusters.

```
funcion calcIntraDist(vector de flotante: coords, vector de entero: instancia,
   vector de vector de flotante X): flotante
variables:
    entero: numInstancias <- instancia.tamaño(), numDimensiones <- X[0].tamaño()
    flotante: suma <- 0, distanciaActual
inicio:
    para i <- 0 hasta numInstancias hacer
        para j <- 0 hasta numDimensiones hacer
            distanciaActual += ( X[ instancia[i] ][j] - coords[j] )**2
        fpara
        distanciaActual <- sqrt(distanciaActual)
        suma += distanciaActual
    fpara

    retornar suma / numInstancias
ffunc
```

Para la infesabilidad, se recorre una vez los vectores de las restricciones, sumando si se incumplen las restricciones en el vector `S`.

```
funcion infeasibility(vector de entero: S, vector de triplet: ML, vector de triplet
    : CL): entero
variables:
    entero: contador <- 0, numML <- ML.tamaño(), numCL <- CL.tamaño()
inicio:
    para i=0 hasta numML hacer
        // Si x_0 y x_1 están en clusters diferentes, no se cumple ML
        si(S[ML[i].x_0] != S[ML[i].x_1]) entonces
            contador++
        fsi
    fpara

    para i=0 hasta numCL hacer
        // Si x_0 y x_1 están en clusters iguales, no se cumple CL
        si(S[CL[i].x_0] == S[CL[i].x_1]) entonces
            contador++
        fsi
    fpara

    retornar contador
ffunc
```

2.2.5. Evaluación de la Función Objetivo de la Población

El código realiza las evaluaciones de la función objetivo de los genes que han sido modificados, es decir, aquellos que han sufrido una mutación o han cruzado, del resto no porque se estarían gastando evaluaciones innecesarias ya que el valor del fitness pasado lo mantiene la clase `cromosome` internamente. Naturalmente, la primera vez que se ejecuta, la población está marcada para que todos los cromosomas posean cambios y así su fitness esté ya inicializado.

Además de esto, se reinicializa la estructura que almacena el mejor de los cromosomas, se realiza para siempre escoger en cada llamada a esta función el mejor de todos, debido a que no se puede saber a priori si seguirá siendo el mismo que la generación pasada, se compara con el fitness de todos y almacena una copia de su cromosoma junto con el fitness.

Generacional

```
funcion evaluarPoblacionGene(vector de cromosome: &poblacion, vector de vector de
    flotante: X, vector de triplet: ML, vector de triplet: CL, entero: numClusters,
    doble: lambda, entero: numClusters, bestCromosome: &mejorCromosoma) : entero

variables:
    entero: evaluacionesInternas <- 0
inicio

    mejorCromosoma.posicion <- -1
    mejorCromosoma.fitness <- MAX_FLOTANTE
    mejorCromosoma.estaVivo <- verdadero

    para i <- 0 hasta poblacion.tamaño() hacer
```

```
        si(poblacion[i].tieneCambios) entonces
            poblacion[i].asignarFitness( fitness(poblacion[i].obtenerGenes(), X, ML
                , CL, lambda, numClusters))
            poblacion[i].asignarCambios(falso)
            evaluacionesInternas++
        fsi

        si(poblacion[i].obtenerFitness() < mejorCromosoma.fitness) entonces
            mejorCromosoma.posicion <- i
            mejorCromosoma.fitness <- poblacion[i].obtenerFitness()
        fsi
    fpara

    mejorCromosoma.datos <- poblacion[ mejorCromosoma.posicion ]

    retornar evaluacionesInternas

ffunc
```

Estacionario

Notar que la función se mantiene relativamente igual, por lo tanto se han obviado partes del código. Lo importante es que ahora existe una lista de pares `<entero, flotante>` que almacena los dos peores cromosomas. Se inserta población entera en la lista con su índice y fitness, se ordena de mayor a menor por el fitness (la segunda compontente del par) y se dejan los dos primeros como se ha pedido en la práctica. Esta función se utiliza al principio del algoritmo y también durante el bucle, aunque en esta última realmente no evalúa los cromosomas ya que esto se realiza en el reemplazamiento dado que debe saberse si los hijos son mejores o peores que los peores cromosomas de la generación, pero se decidió mantenerla, naturalmente evitando que los cromosomas se reevalúen porque sería un gasto de evaluaciones y eficiencia, debido a que también actualiza el mejor cromosoma y la lista de los peores. Los cromosomas ya vienen "sin cambios" luego del reemplazamiento.

```
funcion evaluarPoblacionEsta(vector de cromosome: &poblacion, ..., lista de par<
   entero, flotante> &peoresCromosomas) : entero

variables:
   /* Mismo que Generacional */
inicio
        .
        . /* Ídem */
        .
    para i <- 0 hasta poblacion.tamaño() hacer
        .
        . /* Ídem */
        .
        si(poblacion[i].obtenerFitness() < mejorCromosoma.fitness) entonces
            mejorCromosoma.posicion <- i
            mejorCromosoma.fitness <- poblacion[i].obtenerFitness()
        fsi

        peoresCromosomas.push(i, poblacion[i].obtenerFitness())
```

```
    fpara
    // El primero de la lista es el peor de todos, el del fitness más alto.
    peoresCromosomas.ordenar(mayorMenor)
    peoresCromosomas.cambiarTamaño(2)

    mejorCromosoma.datos <- poblacion[ mejorCromosoma.posicion ]

    retornar evaluacionesInternas
ffunc
```

2.2.6. Generación de solución aleatoria

La generación de una solución aleatoria es idéntica en el genético y memético, es básicamente el procedimiento utilizado en Búsqueda Local dentro de un bucle. Para garantizar que sea factible la solución, primero se asigna a cada instancia un cluster secuencialmente desde 0 hasta el número de clusters, luego, el resto del vector se rellena aleatoriamente y finalmente se realiza un barajeo para desordenar las instancias. Esto se repite para cada individuo, de igual manera se va llevando un conteo de qué instancias están en qué clusters.

```
procedimiento generarPoblacionInicial(vector de cromosome: poblacion, entero: k)
variables:
    vector de entero: auxGen(poblacion[0].tamañoGen()) // Inicializado al tamaño
       del cromosoma
    vector de entero: auxConteoCluster(0, k) // Inicializado a la cantidad de
       clusters k con 0s
inicio

    para i <- 0 hasta poblacion.tamaño() hacer

        // Se asigna a la instancia j el cluster j.
        para j <- 0 hasta k hacer
            auxGen[j] <- j
            auxConteoCluster[j]++
        fpara

        // El resto se asigna aleatoriamente.
        para j <- k hasta poblacion.tamañoCromosoma() hacer
            auxGen[j] <- enteroRand(0, k)
            auxConteoCluster[ auxGen[j] ]++
        fpara

        // Una vez rellenado, se barajea para desordenar.
        random_shuffle(auxGen)

        // Se asigna al vector de población
        poblacion[i].asignarGen(auxGen)
        poblacion[i].asignarConteoCluster(auxConteoCluster)
    fpara

fproc
```

2.3. Algoritmos Genéticos

2.3.1. Función principal

Generacional

La función principal de los algoritmos es muy similar tanto para el generacional como la estacionaria pero se ha preferido diferenciar las dos para que sea más cómodo distinguir sus diferencias. En general, lo que se hace es que, inicialmente se declaran 3 vectores de tipo cromosome, dónde estará nuestra población, una vez inicializado el principal que se llama "`poblacion`", se genera una solución aleatoria y se evalúa, aquí el registro "`bestCromosome`" ahora posee valores internos. Se pasa al bucle principal, el cual es sencillo debido a que cada paso del algoritmo se ha decidido realizar como una función diferente, pero se puede ver que se hace la selección, de aquí los individuos seleccionados estarán en "`poblacionPadre`", esta cruzará de manera uniforme o de segmento fijo dependiendo del parámetro que recibe la función principal y quedarán los hijos, los cuales mutarán, reemplazarán la población original y el resultado se reevalúa, sumando las evaluaciones que se hayan realizado.

```
funcion AGG(vector de vector de flotante: X, vector de triplet: ML, vector de
   triplet: CL, entero: numClusters, doble: lambda, entero: tamañoPoblacion,
   booleano: usarOperadorUniforme): vector de entero

variables:
    entero: evaluaciones <- 0
    vector de cromosome: poblacion, poblacionPadre, poblacionHijo
    bestCromosome: mejorCromosoma

inicio:
    poblacion.inicializarEntero(tamañoPoblacion) // Se reserva la memoria para cada
        cromosoma, internamente las variables están también inicializadas.
    poblacionPadre.inicializar(tamañoPoblacion) // Se reserva la memoria para cada
       cromosoma.

    generarPoblacionInicial(poblacion, numClusters)
    evaluarPoblacion(poblacion, X, ML, CL, lambda, k, mejorCromosoma)

    mientras (evaluaciones < 100000) hacer
        seleccionPoblacion(poblacion, poblacionPadre, mejorCromosoma)

        si (usarOperadorUniforme) entonces
            poblacionHijo = cruceUniforme(poblacionPadre, mejorCromosoma)
        sino
            poblacionHijo = cruceSegFijo(poblacionPadre, mejorCromosoma)
        fsi

        mutarPoblacionGen(poblacionHijo, mejorCromosoma)
        reemplazarPoblacionGen(poblacion, poblacionHijo, mejorCromosoma)
        evaluaciones += evaluarPoblacion(poblacion, X, ML, CL, lambda, k,
           mejorCromosoma)

    fmientras
    retornar mejorCromosoma.datos.obtenerGenes()
ffunc
```

Estacionario

El principal cambio con el Generacional recae en que ahora se posee una lista de pares `[entero, flotante]` donde se mantienen los peores cromosomas de la población, además, en muchos de los operadores ahora no se pasa como argumento el mejor cromosoma puesto que para esta variante, no es necesario saber si está vivo o no el mejor cuando toca reemplazar las poblaciones.

```
funcion AGE(vector de vector de flotante: X, vector de triplet: ML, vector de
   triplet: CL, entero: numClusters, doble: lambda, entero: tamañoPoblacion, entero
   : tamañoHijos, booleano: usarOperadorUniforme): vector de entero

variables:
    entero: evaluaciones <- 0
    vector de cromosome: poblacion, poblacionPadre, poblacionHijo
    bestCromosome: mejorCromosoma
    lista de par[entero, flotante] peoresCromosomas

inicio:
    poblacion.inicializarEntero(tamañoPoblacion) // Se reserva la memoria para cada
        cromosoma, internamente las variables están también inicializadas.
    poblacionPadre.inicializar(tamañoHijos) // Se reserva la memoria para cada
       cromosoma.

    generarPoblacionInicial(poblacion, numClusters)
    evaluarPoblacionEsta(poblacion, X, ML, CL, lambda, k, mejorCromosoma,
       peoresCromosomas)

    mientras (evaluaciones < 100000) hacer
        seleccionPoblacion(poblacion, poblacionPadre)

        si (usarOperadorUniforme) entonces
            poblacionHijo = cruceUniforme(poblacionPadre)
        sino
            poblacionHijo = cruceSegFijo(poblacionPadre)
        fsi

        mutarPoblacionEsta(poblacionHijo)

        evaluaciones += reemplazarPoblacionEsta(poblacion, poblacionHijo,
           peoresCromosomas, X, ML, CL, lambda, k)

        evaluarPoblacionEsta(poblacion, X, ML, CL, lambda, k, mejorCromosoma,
           peoresCromosomas)

    fmientras
    retornar mejorCromosoma.datos.obtenerGenes()
ffunc
```

2.3.2. Operador de Selección

Si se está en la variante generacional, se hace uso de `bestCromosome` para saber si el mejor cromosoma es seleccionado en los padres o si no se selecciona, por ello primero se inicializa a falso, y si se encuentra cambia a verdadero. Todo esto es ignorado por la variante Estacionaria.

La selección es por duelo, se seleccionan dos individuos aleatoriamente y el que posea el mejor fitness de ambos se copia al vector de individuos padre.

```
procedimiento seleccionPoblacion(vector de cromosome: pop, vector de cromosome: &
   poblacionPadre, bestCromosome: mejorCrom)
variables:
    entero: luchadorA, luchadorB
inicio:

    // Ignorado por Estacionario
    mejorCrom.estaVivo <- falso

    // Notar que es el tamaño de población padre, así pueden ser 50 o 2 y funciona
       igual. El vector ya tiene el espacio reservado, por eso se puede acceder
       directamente a las componentes.
    para i <- 0 hasta poblacionPadre.tamaño() hacer

        // Se asume que randEntero devuelve los valores de los entre [min,max)
        luchadorA = randEntero(0, pop.tamaño()-1)
        luchadorB = randEntero(0, pop.tamaño()-1)

        si( pop[luchadorA].obtenerFitness() < pop[luchadorB].obtenerFitness() )
           entonces
            poblacionPadre[i] <- pop[luchadorA]

            // Ignorado por Estacionario
            si(luchadorA == mejorCrom.posicion) entonces
                mejorCrom.posicion <- i
                mejorCrom.estaVivo <- verdadero
            fsi
        sino
            poblacionPadre[i] <- pop[luchadorB]

            // Ignorado por Estacionario
            si(luchadorB == mejorCrom.posicion) entonces
                mejorCrom.posicion <- i
                mejorCrom.estaVivo <- verdadero
            fsi
        fsi
    fpara

fproc
```

2.3.3. Operador de Cruce Uniforme

En el cruce uniforme, se realiza la esperanza matemática del cruce y siempre se comienza desde el principio de la población para ahorrarse usar llamadas a aleatorios.

La función crea primero una copia de la población padre en la población hijo, luego, en el bucle principal, se tienen dos variables auxiliares las cuales son dos cromosomas, se realiza entonces el cruce de la manera explicada; se generan $\frac{n}{2}$ posiciones aleatorias y se asignan de un padre al hijo y el resto del otro padre. Esto se realiza inversamente en el código por facilidad, es decir, se asigna todo de un padre al hijo y luego se cambian posiciones aleatorias del otro padre. Una vez finalizado,

se reemplaza el individuo por la copia modificada en la población de los hijos.

Se realiza esto en dos hijos, cada hijo toma un padre base diferente y las copias del opuesto, de manera simétrica pero variando las posiciones y los clusters que se cambian.

Para evitar que queden clusters vacíos, la generación de los números aleatorios está dentro de un `hacer-mientras` que si detecta que la posición a reemplazarse es aquella donde hay un cluster que solo tiene un miembro, vuelve a generar otra posición.

```
funcion cruceUniforme(vector de cromosome: poblacionPadre, bestCromosome mejorCrom)
    : vector de cromosome
variables:
    vector de entero: hijoA, hijoB
    vector de cromosome: poblacionHijo <- poblacionPadre

    entero: numeroCruces <- techo(poblacionPadre.tamaño() * probabilidadCruce), //
        Variable global
            genesQueCambian <- poblacionPadre[0].TamañoGen() / 2,
            tamañoGen <- poblacionPadre[0].TamañoGen()
            cruceA,
            cruceB
inicio:
    // Esto es ignorado por el Estacionario
    si (mejorCrom.estaVivo ∧ mejorCrom.posicion < numeroCruces) entonces
        // Si su posición está en los que van a cruzar, indudablemente sufrirá
           alguna modificación, luego, no será el mismo cromosoma. Se pone que ha
           muerto.
        mejorCrom.estaVivo = falso
    fsi

    // Se hace de dos en dos, primero con segundo, tercero con cuarto, ...
    para i <- 0 hasta numeroCruces de 2 en 2 hacer
        hijoA <- poblacionPadre[i].obtenerGenes()
        hijoB <- poblacionPadre[i+1].obtenerGenes()

        para j <- 0 hasta genesQueCambian hacer
            hacer
                cruceA <- randEntero(0, tamañoGen)
            mientras deje el cluster en hijoA[cruceA] vacío

            hacer
                cruceB <- randEntero(0, tamañoGen)
            mientras deje el cluster en hijoB[cruceB] vacío

            // Se cruza de un padre al hijo, se realiza dos veces.
            hijoA[cruceA] <- poblacionPadre[i+1].obtenerCluster(cruceA)
            hijoB[cruceB] <- poblacionPadre[i].obtenerCluster(cruceB)

        fpara

        // La cuenta de los clusters se actualiza también.
        poblacionHijo[i].asignarGen(hijoA)
        poblacionHijo[i].asignarCambios(verdad)

        poblacionHijo[i+1].asignarGen(hijoA)
```

```
        poblacionHijo[i+1].asignarCambios(verdad)

    fpara

    retornar poblacionHijo

ffunc
```

2.3.4. Operador de Cruce de Segmento Fijo

El funcionamiento es similar al del cruce uniforme, se crea una copia de los padres en los hijos y se modifican los primeros cromosomas para ahorrar las llamadas a los números aleatorios.

Se utilizan nuevamente dos vectores auxiliares junto a las variables de segmento, de la misma manera que se implementó el cruce uniforme, aquí también se realizan las operaciones pero en otro orden, se copia un padre entero al hijo y luego dado el segmento donde inicia y su longitud, se decide aleatoriamente si se copia un gen en una posición dada de ese padre base o del otro padre al hijo, si se llega al final del vector se comienza desde el principio. Esto equivaldría a decidir un inicio de segmento y copiar todo por una longitud y lo restante obtenerlo aleatoriamente.

Nuevamente como se tienen que generar dos hijos, se realiza la operación dos veces en el bucle, cada hijo tiene como base un padre diferente y se cruza con el padre que no es base. Se ha pensado que así se puede mantener un equilibrio entre exploración y explotación ya que en la mayoría de los casos un padre será mejor que el otro, si un hijo tiene una mayoría del mejor padre y otro del peor padre, se estará explorando y explotando de la misma manera.

```
funcion cruceFijo(vector de cromosoma: poblacionPadre, bestCromosome mejorCrom) :
   vector de cromosome
variables:
    vector de entero: hijoA, hijoB
    vector de cromosome: poblacionHijo <- poblacionPadre
                                    // Variable global ↓
    entero: numeroCruces <- poblacionPadre.tamaño() * probabilidadCruce,
            tamañoGen <- poblacionPadre[0].TamañoGen()
            inicioSegmento,
            longitudSegmento,
            cruceA,
            cruceB
inicio:

    // Esto es ignorado por el Estacionario
    si (mejorCrom.estaVivo ∧ mejorCrom.posicion < numeroCruces) entonces
        // Si su posición está en los que van a cruzar, indudablemente sufrirá
           alguna modificación, luego, no será el mismo cromosoma. Se pone que ha
           muerto.
        mejorCrom.estaVivo = falso
    fsi

    para i <- 0 hasta numeroCruces de 2 en 2 hacer
        hijoA <- poblacionPadre[i].obtenerGenes()
        hijoB <- poblacionPadre[i+1].obtenerGenes()
```

```
        longitudSegmento <- enteroRand(0, tamañoGen)
        inicioSegmento <- enteroRand(0, tamañoGen)

        para j <- 0 hasta longitudSegmento hacer

            cruceA = enteroRand(0,1)
            cruceB = enteroRand(0,1)

            si(cruceA ∧ el cluster en  hijoA[(j+inicioSegmento)%tamañoGen] no queda
                vacío) entonces
                hijoA[(j+inicioSegmento)%tamañoGen] <- poblacionPadre[i+1].
                   obtenerCluster((j+inicioSegmento)%tamañoGen)
            fsi

            si(cruceB ∧ el cluster en hijoB[(j+inicioSegmento)%tamañoGen] no queda
               vacío) entonces
                hijoB[(j+inicioSegmento)%tamañoGen] <- poblacionPadre[i].
                   obtenerCluster((j+inicioSegmento)%tamañoGen)
            fsi

        fpara

        // La cuenta de los clusters se actualiza también.
        poblacionHijo[i].asignarGen(hijoA)
        poblacionHijo[i].asignarCambios(verdad)

        poblacionHijo[i+1].asignarGen(hijoA)
        poblacionHijo[i+1].asignarCambios(verdad)

    fpara

    retornar poblacionHijo

ffunc
```

2.3.5. Operador de Mutación

Generacional

Para el caso del generacional, se hace uso de la esperanza matemática nuevamente para obtener el número esperado de mutaciones y se decidió añadir un elemento más: que no necesariamente sean los primeros cromosomas quienes muten, sino que aleatoriamente se decide en donde comenzar, teniendo en consideración que este valor se acota con el número esperado de mutaciones menos el tamaño del vector para que nunca se sobrepase la longitud del vector.

Una vez se entra en el bucle, se genera una posición aleatoriamente, comprobando que en esa posición no hay un cluster con una sola instancia, luego se genera un valor del nuevo cluster procurando que sea distinto al cluster actual, una vez se tienen esos dos valores, se modifica el gen y se pasa al siguiente.

```
procedimiento mutarPoblacionGen(vector de cromosome: &poblacionHijo, bestCromosome
   mejorCrom)
variables:
```

```
                              // Variable global ↓
    entero: mutacionTamaño <- probabilidadMutacion * poblacionHijo[0].tamaño(),
            mutacionInicio <- randEntero(0, poblacionHijo[0].tamaño() -
               mutacionTamaño)
            posicionMutacion,
            genMutado,
            tamañoGen <- poblacionHijo[0].tamañoGen(),
            tamañoCluster <- poblacionHijo[0].obtenerCantidadClusters()

inicio:
    // Ignorado por Estacionario
    si (mejorCrom.estaVivo ∧ bestCrom.pos está entre mutacionInicio y
       mutacionInicio + mutacionTamaño) entonces
        mejorCrom.estaVivo <- falso
    fsi

    para i <- 0 hasta mutacionTamaño hacer
        hacer
            posicionMutacion <- randEntero(0, tamañoGen)
        mientras deje el cluster en esa posición vacío

        hacer
            genMutado <- randEntero(0, tamañoCluster)
        mientras sea el mismo cluster que está en posicioMutacion

        // Internamente se actualizan los contadores y se cambia el booleano de que
            el cromosoma ha sido modificado
        poblacionHijo[i+mutacionInicio].cambiarGen(posicionMutacion, genMutado)

    fpara
fproc
```

Estacionario

Debido a que en la práctica se utilizan solo dos individuos para la población de hijos debió realizarse una ligera modificación al código, ahora la probabilidad es calculada con un aleatorio en vez de con la esperanza matemática, de esta manera puede suceder que ninguno mute, uno mute o ambos muten. El resto del código es idéntico a la variante generacional.

```
procedimiento mutarPoblacionEsta(vector de cromosome: &poblacionHijo, bestCromosome
    mejorCrom)
variables:

    entero: posicionMutacion,
            genMutado,
            tamañoGen <- poblacionHijo[0].tamañoGen(),
            tamañoCluster <- poblacionHijo[0].obtenerCantidadClusters()

inicio:

    para i <- 0 hasta poblacionHijo.tamaño() hacer
        si (randFlotante(0, 1) > (1 - probabilidadMutacion)) entonces
            hacer
                posicionMutacion <- randEntero(0, tamañoGen)
```

```
            mientras deje el cluster en esa posición vacío

            hacer
                genMutado <- randEntero(0, tamañoCluster)
            mientras sea el mismo cluster que está en posicioMutacion

            // Internamente se actualizan los contadores y se cambia el booleano de
                que el cromosoma ha sido modificado
            poblacionHijo[i+mutacionInicio].cambiarGen(posicionMutacion, genMutado)
        fsi
    fpara
fproc
```

2.3.6. Operador de Reemplazo

Generacional

```
procedimiento reemplazarPoblacionGen(vector de cromosome: &poblacion, vector de
   cromosoma: poblacionHijo, bestCromosome: mejorCrom)
inicio:
    // Se reemplaza por completo la población actual por la población de hijos
    poblacion <- poblacionHijo

    // Si el mejor cromosoma "murió", es decir, fue modificado, no fue seleccionado
        o mutó, entonces se reemplaza en la primera posición del vector de població
       n.
    si(!mejorCrom.estaVivo) entonces
        poblacion[0] <- mejorCrom.datos
    fsi
fproc
```

Estacionario

Para esta variante se mantiene una lista (en la práctica de solo 2 elementos) de los peores dos cromosomas de la población actual, aquí ahora se evalúan los dos hijos, contando estas evaluaciones para el total y se juntan tanto los hijos como los peores cromosomas en una lista que se ordena de menor a mayor fitness; si sale un hijo (que para diferenciarlo de la población actual, se le asigna que tenga un índice negativo, esto es luego devuelto al índice real del individuo), entonces este reemplazará al cromosoma que peor tiene su fitness, es decir, el que está en el frente de la lista original de los peores cromosomas. Así seguirá hasta que queden vacías las listas.

Dado que aquí se realiza la evaluación de la función objetivo, se mantiene que todos estos cromosomas no han cambiando para que la función de evaluación no cuente evaluaciones demás, se ha dejado la función de evaluación debido a que ella también actualiza el mejor cromosoma y la lista de los peores.

```
funcion reemplazarPoblacionEsta(vector de cromosome: &siguientePoblacion, vector de
    cromosoma: poblacionHijo, lista de par[entero, flotante] peoresCromosomas,
   vector de vector de flotante: X, vector de triplet: ML, vector de triplet: CL,
   flotante: lambda, entero: k) : entero
variable:
    flotante: auxFitness
    entero: evaluacionInterna <- 0
```

```
    lista de par[entero, flotante]: listaAux
inicio:

    listAux <- peoresCromosomas

    para i <- 0 hasta poblacionHijo.tamaño() hacer
        auxFitness <- fitness(poblacionHijo[i].obtenerGenes(), X, ML, CL, lambda, k
            )
        evaluacionInterna++
        // Se crea un par con un índice negativo para saber que es un hijo y no un
            cromosoma original.
        listAux.push_back( {-i-1, auxFitness} )
        poblacionHijo[i].asignarFitness(auxFitness)
        poblacionHijo[i].asignarCambios(falso)
    fpara

    // El que está de primero es el que tiene el mejor fitness, el más bajo.
    listAux.ordenar(menorMayor)
    listAux.cambiarTamaño(2)

    mientras (!listAux.vacio() ∧ !peoresCromosomas.vacio()) hacer:
        // Si la primera componente es negativa, es un hijo.
        si (listAux.frente().primero < 0)
            // Reemplazar el cromosoma de peor fitness con el hijo, para acceder al
                hijo se revierte el índice negativo al índice original.
            siguientePoblacion[ peoresCromosomas.frente().primero ] <-
               poblacionHijo[ abs(listAux.frente().primero)-1 ]
            peoresCromosomas.pop_frente()

        fsi
        listAux.pop_frente()

    fmientras

    retornar evaluacionInterna

ffunc
```

2.4. Algoritmos Meméticos

2.4.1. Función principal

De los algoritmos genéticos, el que en promedio dio los valores más bajos de fitness –con un margen pequeño, se debe añadir– entre todos los grupos de datos fue el Algoritmo Genético Estacionario con cruce por Segmento Fijo, por lo tanto esta ha sido la variante que se ha modificado para añadirle la componente de Búsqueda Local.

El algoritmo soporta un porcentaje arbitrario entre 0 y 1 para la población y también que sea utilizando los mejores individuos o simplemente los que sean, en este caso, se decidió en utilizar los primeros en el vector de población

```
funcion AM(vector de vector de flotante: X, vector de triplet: ML, vector de
   triplet: CL, entero: numClusters, doble: lambda, entero: tamañoPoblacion, entero
```

```
   : tamañoHijos, entero: generacionBL, flotante: porcentajePopblacion, booleano:
   seleccionarMejores): vector de entero

variables:
    entero: evaluaciones <- 0, generaciones <- 1, numeroFallos <- 0.1 * X.tamaño()
    vector de cromosome: poblacion, poblacionPadre, poblacionHijo
    bestCromosome: mejorCromosoma
    lista de par[entero, flotante]: peoresCromosomas

inicio:
    poblacion.inicializarEntero(tamañoPoblacion) // Se reserva la memoria para cada
        cromosoma, internamente las variables están también inicializadas.
    poblacionPadre.inicializar(tamañoHijos) // Se reserva la memoria para cada
       cromosoma.

    generarPoblacionInicial(poblacion, numClusters)
    evaluarPoblacion(poblacion, X, ML, CL, lambda, k, mejorCromosoma,
       peoresCromosomas)

    mientras (evaluaciones < 100000) hacer
        seleccionPoblacion(poblacion, poblacionPadre)

        poblacionHijo = cruceSegFijo(poblacionPadre)

        mutarPoblacionEsta(poblacionHijo)

        evaluaciones += reemplazarPoblacionEsta(poblacion, poblacionHijo,
           peoresCromosomas, X, ML, CL, lambda, k)

        si(generacion = generacionBL) entonces
            generacion <- 1
            si (!seleccionarMejor) entonces
                para i <- 0 hasta tamañoPoblacion * porcentajePopblacion hacer
                    evaluaciones += BLS(poblacion[i], k, numeroFallos, X, ML, CL,
                       lambda)
                fpara

            sino
                para i <- 0 hasta tamañoPoblacion hacer
                    cromAux.primero <- i
                    cromAux.segundo <- poblacion[i]
                    mejoresCroms.push_back(cromAux)
                fpara

                mejoresCroms.ordenar(mejorFitness)

                para i <- 0 hasta tamañoPoblacion * porcentajePoblacion hacer
                    cromAux <- mejoresCroms.frente()
                    mejoresCroms.pop_frente()

                    evaluaciones += BLS(cromAux.segundo, k, numeroFallos, X, ML, CL
                       , lambda)

                    si (cromAux.segundo.obtenerCambios()) entonces
```

```
                        poblacion[cromAux.primero] <- cromAux.segundo
                        poblacion[cromAux.primero].asignarCambios <- falso
                    fsi
                fpara
            fsi
        fsi

        evaluarPoblacionEsta(poblacion, X, ML, CL, lambda, k, mejorCromosoma,
           peoresCromosomas)
        generaciones++

    fmientras
    retornar mejorCromosoma.datos.obtenerGenes()
ffunc
```

2.4.2. Búsqueda Local Suave

La BLS fue adaptada directamente de la descripción de las transparencias del Seminario, recibe un cromosoma de entrada junto a los datos importantes para evaluar el fitness, el vector de índices `RSI` se genera y se realiza un barajeo. luego, en el bucle después de verificar que hay cambio y que el número de errores no supera el límite, se copia el cromosoma, se verifica que el cambio no dejaría un cluster sin miembros o es el mismo cluster que ya está asignado, se cambia y se evalúa; si el cambio resulta en una mejora, entonces se guardan los datos y se cambia el booleano. El algoritmo prosigue ya que se pretente conseguir el mejor cambio posible, por lo tanto no se puede saber cual será y deben probarse todos. Una vez que se realiza esto, se verifica si ha habido mejora; sino, se aumenta el contador de fallos.

```
funcion BLS(cromosome: &S, entero: clusters, entero: maxFallos, vector de vector de
    entero: X, vector de triplet: ML, vector de triplet: CL, flotante: lambda) :
   entero
variables:
    entero: numeroFallos <- 0, nuevoGen, nuevoCluster
    flotante:
    booleano: hayMejora <- verdadero
    // RSI se rellena con los valores desde 0 hasta S.tamañoGen()
    vector de entero: RSI(0, S.tamañoGen())
    cromosome: actS
inicio:
    random_shuffle(RSI)
    para i <- 0 hasta S.tamañoGen() hacer
        si (numeroFallos > maxFallos && !hayMejora) entonces
            break;
        fsi

        hayMejora <- falso

        para j <- 0 hasta clusters hacer
            actS <- S
            si(j != actS.obtenerCluster(RSI[i] ∧ el cluster en la posición RSI[i]
               posee más de una instancia asociada) entonces
                actS.cambiarGen(RSI[i], k)
```

```
                nuevaFitness <- fitness(actS.obtenerGenes(), X, ML, CL, lambda,
                    clusters)

                evaluacionesInternas++

                si(nuevaFitness < actualFitness) entonces
                    actualFitness <- nuevaFitness
                    nuevoGen <- RSI[i]
                    nuevoCluster <- j
                    hayMejora <- verdadero
                fsi
        fpara

            si (!hayMejora) entonces
                numeroFallos++
            sino
                S.cambiarGen(nuevoGen, nuevoCluster)
                S.cambiarFitness(actualFitness)
            fsi

    fpara

    retornar evaluacionesInternas

ffunc
```

3. Procedimiento considerado al desarrollar la práctica

Para esta práctica se consideró el uso del lenguaje de programación C++ debido a su versatilidad, la familiaridad del autor con el mismo y también por ser un lenguaje rápido en su ejecución, a diferencia de lenguajes interpretados, esta consideración se realiza debido a la advertencia de que los tiempos de ejecución pueden ser elevados en los algoritmos.

Los códigos fueron implementados por el autor basándose en las diapositivas del Seminario 3, junto con el guión de prácticas.

Debido a que en general se tratan de dos algoritmos, genético y memético al cual se aplican variantes, se decidió realizar dos archivos de código fuente que internamente poseen las implementaciones pedidas explícitamente diferenciadas en lo posible.

Se ha hecho uso de la librería proporcionada para la generación de números aleatorios en los algoritmos además de librerías de C++ Estándar ya sea para barajar un vector o tomar el tiempo de ejecución.

3.1. Manual de Usuario

3.1.1. Compilación

Para la compilación, se ha provisto de un fichero `Makefile` el cual posee las siguientes recetas:

- `Make all`: Compila ambos algoritmos, así como la librería `random.h` proporcionada.

- `Make GENE_exe`: Compila el ejecutable de los algoritmos genéticos.
- `Make MEME_exe`: Compila el ejecutable de los algoritmos meméticos.
- `Make clean_obj`: Elimina los ficheros objeto.
- `Make clean`: Elimina los ejecutables y los ficheros objeto.

Existen otras recetas pero estas son utilizadas de manera interna por lo que se recomienda utilizar solo las recetas presentadas.

3.1.2. Ejecución

Para realizar una única ejecución, los algoritmos utilizan los siguientes parámetros de entrada:

```
$./algoritmo_exe n d k setPath constPath randomSeed
```

Donde

- `n`: Número de instancias del conjunto de datos.
- `d`: Número de dimensiones del conjunto de datos.
- `k`: Número de clusters.
- `setPath`: Ruta hacia el fichero `*.dat`.
- `constPath`: Ruta hacia el fichero `*.const`
- `randomSeed`: Número que se utiliza para inicializar el generador de números aleatorios.

Se incluye también un script de Bash el cual se utilizó para obtener los datos requeridos.

```
$bash gatherData_P2.sh
```

El script espera que en la ubicación dónde sea ejecutado exista, además de los programas ejecutables, las carpetas "input", donde estarían los datos de entrada y "output", donde se almacenarán las salidas de los programas siguiendo el formato `dataSet_algoritmo_restricciones.txt`.

4. Resultados y Análisis

4.1. Casos de Problema y Parámetros Empleados

Para la resolución de esta práctica, se utilizaron los tres grupos de datos provistos en la práctica, estos son:

- Zoo: 101 instancias, 16 atributos y 7 clusters. Se trata de clasificar animales para agruparlos en grupos similares.
- Glass: 214 instancias, 9 atributos y 7 clusters. Se trata de agrupar diferentes tipos de vidrios.
- Bupa: 345 instancias, 5 atributos y 16 clusters. Se trata de agrupar personas dependiendo de sus hábitos de consumo de alcohol.

Además, se incluye también otro pequeño grupo de datos que fue utilizado para poder depurar los algoritmos, denominado simplemente "Test", fue inspirado en el ejemplo que se puede observar en las diapositivas del Seminario 2 de la asignatura. Posee 8 instancias, 2 atributos y 2 clusters.

Referente a los parámetros del algoritmo, se utilizaron los grupos de datos las instancias, atributos y clusters originales. Se hizo uso de la página web `https://www.random.org/integers/` para obtener semillas de números enteros.

Las semillas que se utilizaron son las siguientes:

```
860681 980472 206894 954919 426969
```

4.2. Resultados Obtenidos

Haciendo uso del script de Bash anteriormente mencionado, se obtuvieron los siguientes resultados de 5 ejecuciones de cada variante de los algoritmos, tanto para 10 % como 20 % de restricciones.

4.2.1. Algoritmo Genético Generacional Elitista con Operador Uniforme

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	19	0,631091	0,787469	57,9461	24	0,256269	0,289495	120,673	342	0,156946	0,278395	303,549
Ejecución 2	11	0,620266	0,710801	59,8792	43	0,199159	0,258689	114,054	418	0,152449	0,300886	280,294
Ejecución 3	13	0,622592	0,729588	61,9773	23	0,249045	0,280886	118,18	252	0,133959	0,223448	279,519
Ejecución 4	8	0,619921	0,685765	61,2819	54	0,18905	0,263808	113,035	283	0,164	0,264497	279,851
Ejecución 5	25	0,744967	0,950729	58,6207	25	0,228469	0,263079	107,699	351	0,152847	0,277492	278,118
Media	**15,20**	**0,6478**	**0,7729**	**59,9410**	**33,80**	**0,2244**	**0,2712**	**114,73**	**329,20**	**0,1520**	**0,2689**	**284,27**

10 % Restricciones

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	22	0,713206	0,809697	62,5902	44	0,248673	0,280899	119,266	736	0,148749	0,284743	288,45
Ejecución 2	24	0,752158	0,857422	58,8991	53	0,249827	0,288646	122,401	500	0,150661	0,243048	284,813
Ejecución 3	53	0,613161	0,845617	58,436	46	0,249475	0,283166	120,155	547	0,138091	0,239163	286,674
Ejecución 4	42	0,67527	0,859481	58,3482	89	0,215425	0,280611	95,3851	593	0,150046	0,259617	294,305
Ejecución 5	31	0,66422	0,800185	53,5774	104	0,210828	0,287	106,478	624	0,154483	0,269782	296,793
Media	**34,40**	**0,6836**	**0,8345**	**58,3702**	**67,20**	**0,2348**	**0,2841**	**112,74**	**600,00**	**0,1484**	**0,2593**	**290,21**

20 % Restricciones

4.2.2. Algoritmo Genético Generacional Elitista con Operador de Segmento Fijo

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	12	0,620798	0,719564	58,0487	59	0,188182	0,269862	120,608	322	0,157934	0,27228	302,037
Ejecución 2	3	0,758404	0,783095	59,2383	53	0,197724	0,271097	113,658	403	0,156048	0,299159	292,765
Ejecución 3	13	0,620492	0,727488	60,9856	63	0,189554	0,276771	117,577	238	0,168186	0,252703	279,473
Ejecución 4	6	0,759393	0,808776	59,9832	33	0,225816	0,271501	118,496	253	0,148138	0,237982	279,953
Ejecución 5	13	0,605985	0,712981	61,3217	30	0,209472	0,251004	111,165	419	0,140463	0,289255	279,02
Media	**9,40**	**0,6730**	**0,7504**	**59,9155**	**47,60**	**0,2021**	**0,2680**	**116,30**	**327,00**	**0,1542**	**0,2703**	**286,65**

10 % Restricciones

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	16	0,760026	0,830202	61,3782	36	0,247233	0,2736	126,008	794	0,161323	0,308034	288,03
Ejecución 2	28	0,71758	0,840387	56,9311	133	0,209988	0,3074	121,537	728	0,149035	0,283551	284,811
Ejecución 3	10	0,759762	0,803621	58,8045	38	0,249593	0,277425	116,403	462	0,148812	0,234178	289,312
Ejecución 4	16	0,717476	0,787651	60,3718	109	0,213571	0,293405	99,2106	400	0,152549	0,226459	297,726
Ejecución 5	11	0,74383	0,792075	56,6595	89	0,237606	0,302791	100,827	482	0,140665	0,2297	303,712
Media	**16,20**	**0,7397**	**0,8108**	**58,83**	**81,00**	**0,2316**	**0,2909**	**112,80**	**573,20**	**0,1505**	**0,26**	**292,72**

20 % Restricciones

4.2.3. Algoritmo Genético Estacionario con Operador Uniforme

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	21	0,768276	0,941115	59,2915	25	0,248568	0,283179	118,804	381	0,158248	0,293546	293,952
Ejecución 2	14	0,715005	0,830232	60,0557	17	0,248329	0,271864	117,639	457	0,178919	0,341206	297,995
Ejecución 3	18	0,607411	0,755559	63,9795	62	0,186167	0,272	118,635	610	0,180302	0,396921	289,661
Ejecución 4	13	0,632099	0,739095	58,8999	26	0,253848	0,289843	122,819	470	0,174197	0,3411	288,878
Ejecución 5	11	0,780908	0,871443	63,0012	48	0,19287	0,259321	110,828	522	0,161019	0,346388	296,206
Media	**15,40**	**0,7007**	**0,8275**	**61,0456**	**35,60**	**0,2260**	**0,2752**	**117,75**	**488,00**	**0,1705**	**0,3438**	**293,34**

10 % Restricciones

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	42	0,642665	0,826875	60,6014	113	0,210141	0,292904	129,324	605	0,15389	0,265679	316,806
Ejecución 2	30	0,714599	0,846178	57,5312	108	0,21105	0,290152	120,52	774	0,161835	0,304851	300,049
Ejecución 3	37	0,598492	0,760773	59,1971	42	0,249987	0,280749	111,651	1004	0,153107	0,338621	301,071
Ejecución 4	29	0,617326	0,744519	57,3873	110	0,215927	0,296494	97,8436	653	0,161674	0,282332	327,422
Ejecución 5	18	0,735568	0,814515	60,1354	112	0,213396	0,295427	100,074	985	0,166139	0,348142	310,132
Media	**31,20**	**0,6617**	**0,7986**	**58,9705**	**97,00**	**0,2201**	**0,2911**	**111,88**	**804,20**	**0,1593**	**0,3079**	**311,10**

20 % Restricciones

4.2.4. Algoritmo Genético Estacionario con Operador de Segmento Fijo

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	9	0,617589	0,691663	59,8903	23	0,232596	0,264437	116,577	474	0,16695	0,335274	296,523
Ejecución 2	11	0,611006	0,701541	61,7876	56	0,193627	0,271154	120,732	562	0,162165	0,361738	299,692
Ejecución 3	24	0,774765	0,972296	64,7418	44	0,195563	0,256477	121,98	447	0,162804	0,321539	285,967
Ejecución 4	15	0,619472	0,742929	56,505	28	0,225641	0,264404	124,844	471	0,174564	0,341823	292,759
Ejecución 5	10	0,602761	0,685065	61,0849	31	0,211971	0,254888	110,286	485	0,175077	0,347307	296,993
Media	**13,80**	**0,6451**	**0,7587**	**60,8019**	**36,40**	**0,2119**	**0,2623**	**118,88**	**487,80**	**0,1683**	**0,3415**	**294,39**

10 % Restricciones

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	51	0,614822	0,838506	59,0556	30	0,249694	0,271667	126,159	779	0,154007	0,297947	310,267
Ejecución 2	8	0,803886	0,838974	58,8177	43	0,250349	0,281843	123,317	978	0,169859	0,350568	300,212
Ejecución 3	26	0,71114	0,825176	58,9194	52	0,249937	0,288023	108,486	1091	0,15505	0,356639	299,341
Ejecución 4	37	0,614826	0,777107	57,6993	47	0,249302	0,283726	100,383	960	0,166621	0,344005	317,474
Ejecución 5	36	0,737841	0,895736	60,6664	30	0,249727	0,2717	102,676	905	0,162988	0,330209	315,852
Media	**31,60**	**0,6965**	**0,8351**	**59,0317**	**40,40**	**0,2498**	**0,2794**	**112,20**	**942,60**	**0,1617**	**0,3359**	**308,63**

20 % Restricciones

4.2.5. Algoritmo Memético cada 10 generaciones al 100 % de la población

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	12	0,821402	0,920168	8,69753	251	0,341029	0,688514	19,2408	689	0,219913	0,464586	100,705
Ejecución 2	7	0,658353	0,715966	8,53756	268	0,312519	0,683539	19,5729	634	0,227658	0,4528	104,594
Ejecución 3	5	0,841249	0,882402	8,74962	172	0,344067	0,582184	19,2988	723	0,21302	0,469767	102,915
Ejecución 4	6	0,721068	0,770451	8,51329	260	0,322601	0,682545	17,6455	707	0,214964	0,466029	102,985
Ejecución 5	4	0,832111	0,865033	8,78345	258	0,36909	0,726266	17,4574	668	0,221025	0,458241	102,41
Media	**6,80**	**0,7748**	**0,8308**	**8,6563**	**241,80**	**0,3379**	**0,6726**	**18,64**	**684,20**	**0,2193**	**0,4623**	**102,72**

10 % Restricciones

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	5	0,790282	0,812211	10,0068	371	0,347619	0,619348	19,6941	1088	0,209437	0,410471	112,477
Ejecución 2	9	0,761767	0,80124	9,10058	334	0,379333	0,623962	18,9549	1260	0,212078	0,444894	109,918
Ejecución 3	10	0,756451	0,800311	8,62979	311	0,33942	0,567203	18,3783	1286	0,217364	0,454984	108,189
Ejecución 4	20	0,720977	0,808696	8,82207	364	0,372705	0,639307	18,8978	1256	0,207521	0,439598	111,669
Ejecución 5	19	0,737	0,820333	8,87718	402	0,32572	0,620154	18,8394	1219	0,215144	0,440384	109,291
Media	**12,60**	**0,7533**	**0,8086**	**9,0873**	**356,40**	**0,3530**	**0,6140**	**18,95**	**1.221,80**	**0,2123**	**0,4381**	**110,31**

20 % Restricciones

4.2.6. Algoritmo Memético cada 10 generaciones al 10 % de la población aleatorio

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	13	0,617818	0,724813	8,32803	36	0,214538	0,264376	18,33	219	0,152769	0,230539	64,0543
Ejecución 2	6	0,69552	0,744903	8,38323	37	0,202313	0,253536	17,9896	247	0,160871	0,248584	64,593
Ejecución 3	5	0,701905	0,743057	8,38355	18	0,245411	0,27033	17,2071	194	0,15759	0,226482	64,3224
Ejecución 4	10	0,670927	0,753231	8,49226	44	0,198795	0,259709	17,1562	212	0,158769	0,234053	63,7257
Ejecución 5	9	0,75731	0,831384	8,49017	11	0,248801	0,264029	17,2948	215	0,156138	0,232487	62,8133
Media	**8,60**	**0,6887**	**0,7595**	**8,4154**	**29,20**	**0,2220**	**0,2624**	**17,60**	**217,40**	**0,1572**	**0,2344**	**63,90**

10 % Restricciones

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	11	0,736299	0,784545	9,23029	35	0,249997	0,275632	17,7794	325	0,160069	0,220121	62,9483
Ejecución 2	17	0,732119	0,806681	8,98874	23	0,251193	0,268039	17,4973	318	0,163631	0,222389	65,2024
Ejecución 3	8	0,802528	0,837616	8,55415	28	0,248106	0,268614	18,6046	288	0,152823	0,206038	64,0123
Ejecución 4	6	0,804256	0,830572	9,27784	27	0,251027	0,270803	18,0983	338	0,149325	0,211779	64,4542
Ejecución 5	18	0,715887	0,794834	9,0873	28	0,250537	0,271045	18,064	201	0,152201	0,189341	64,1663
Media	**12,00**	**0,7582**	**0,8108**	**9,0277**	**28,20**	**0,2502**	**0,2708**	**18,01**	**294,00**	**0,1556**	**0,2099**	**64,16**

20 % Restricciones

4.2.7. Algoritmo Memético cada 10 generaciones a los 10 % mejores

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	10	0,608666	0,69097	8,74457	13	0,254511	0,272508	18,8005	230	0,161752	0,243428	62,7208
Ejecución 2	2	0,756408	0,772869	8,77663	14	0,247466	0,266847	17,0711	192	0,164273	0,232455	62,6273
Ejecución 3	13	0,671179	0,778175	8,99076	30	0,212041	0,253573	17,7957	149	0,150495	0,203407	62,9637
Ejecución 4	13	0,60885	0,715846	8,45316	12	0,248656	0,265269	17,1881	215	0,153079	0,229428	63,9215
Ejecución 5	7	0,670931	0,728544	8,62958	35	0,211523	0,259977	17,9921	214	0,144781	0,220775	62,7002
Media	**9,00**	**0,6632**	**0,7373**	**8,7189**	**20,80**	**0,2348**	**0,2636**	**17,77**	**200,00**	**0,1549**	**0,2259**	**62,99**

10 % Restricciones

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
Ejecución 1	23	0,664109	0,764987	9,0817	30	0,25095	0,272923	18,5191	303	0,15246	0,208447	63,048
Ejecución 2	15	0,719162	0,784951	8,8019	42	0,250534	0,281296	17,6504	217	0,148524	0,18862	63,5924
Ejecución 3	13	0,719245	0,776262	8,49237	28	0,251626	0,272134	18,3769	375	0,148127	0,217418	64,8802
Ejecución 4	5	0,806787	0,828717	8,68536	29	0,25088	0,27212	18,565	334	0,152152	0,213867	64,3881
Ejecución 5	16	0,709738	0,779914	8,62591	38	0,250309	0,278141	18,3717	330	0,150678	0,211654	63,1027
Media	**14,40**	**0,7238**	**0,7870**	**8,7374**	**33,40**	**0,2509**	**0,2753**	**18,30**	**311,80**	**0,1504**	**0,2080**	**63,80**

20 % Restricciones

4.2.8. Resultados Globales

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
COPKM	31,60	1,0096	1,2697	0,3691	435,60	0,4077	1,0108	8,3806	798,60	0,2026	0,4862	94,9271
BL	16,00	0,6916	0,8233	0,5457	26,80	0,2331	0,2702	1,9462	109,20	0,1236	0,1624	57,5137
AGG-UN	15,20	0,6478	0,7729	59,9410	33,80	0,2244	0,2712	114,7282	329,20	0,1520	0,2689	284,2662
AGG-SF	9,40	0,6730	0,7504	59,9155	47,60	0,2021	0,2680	116,3008	327,00	0,1542	0,2703	286,6496
AGE-UN	15,40	0,7007	0,8275	61,0456	35,60	0,2260	0,2752	117,7450	488,00	0,1705	0,3438	293,3384
AGE-SF	13,80	0,6451	0,7587	60,8019	36,40	0,2119	0,2623	118,8838	487,80	0,1683	0,3415	294,3868
AM-(10,1.0)	6,80	0,7748	0,8308	8,6563	241,80	0,3379	0,6726	18,6431	684,20	0,2193	0,4623	102,7218
AM-(10,0.1)	8,60	0,6887	0,7595	8,4154	29,20	0,2220	0,2624	17,5955	217,40	0,1572	0,2344	63,9017
AM-(10,0.1mej)	9,00	0,6632	0,7373	8,7189	20,80	0,2348	0,2636	17,7695	200,00	0,1549	0,2259	62,9867

10 % Restricciones

	Zoo				Glass				Bupa			
	Infeasable	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**	**Infeasable**	**Distancia**	**Fitness**	**T**
COPKM	30,60	1,1434	1,2776	0,2994	323,40	0,4031	0,6400	10,2299	1.471,80	0,2424	0,5143	77,2708
BL	25,40	0,7356	0,8470	0,3061	42,80	0,2489	0,2803	1,8000	171,60	0,1248	0,1565	39,4667
AGG-UN	34,40	0,6836	0,8345	58,3702	67,20	0,2348	0,2841	112,7370	600,00	0,1484	0,2593	290,2070
AGG-SF	16,20	0,7397	0,8108	58,8290	81,00	0,2316	0,2909	112,7971	573,20	0,1505	0,2564	292,7182
AGE-UN	31,20	0,6617	0,7986	58,9705	97,00	0,2201	0,2911	111,8825	804,20	0,1593	0,3079	311,0960
AGE-SF	31,60	0,6965	0,8351	59,0317	40,40	0,2498	0,2794	112,2042	942,60	0,1617	0,3359	308,6292
AM-(10,1.0)	12,60	0,7533	0,8086	9,0873	356,40	0,3530	0,6140	18,9529	1.221,80	0,2123	0,4381	110,3088
AM-(10,0.1)	12,00	0,7582	0,8108	9,0277	28,20	0,2502	0,2708	18,0087	294,00	0,1556	0,2099	64,1567
AM-(10,0.1mej)	14,40	0,7238	0,7870	8,7374	33,40	0,2509	0,2753	18,2966	311,80	0,1504	0,2080	63,8023

20 % Restricciones

4.3. Análisis de Resultados

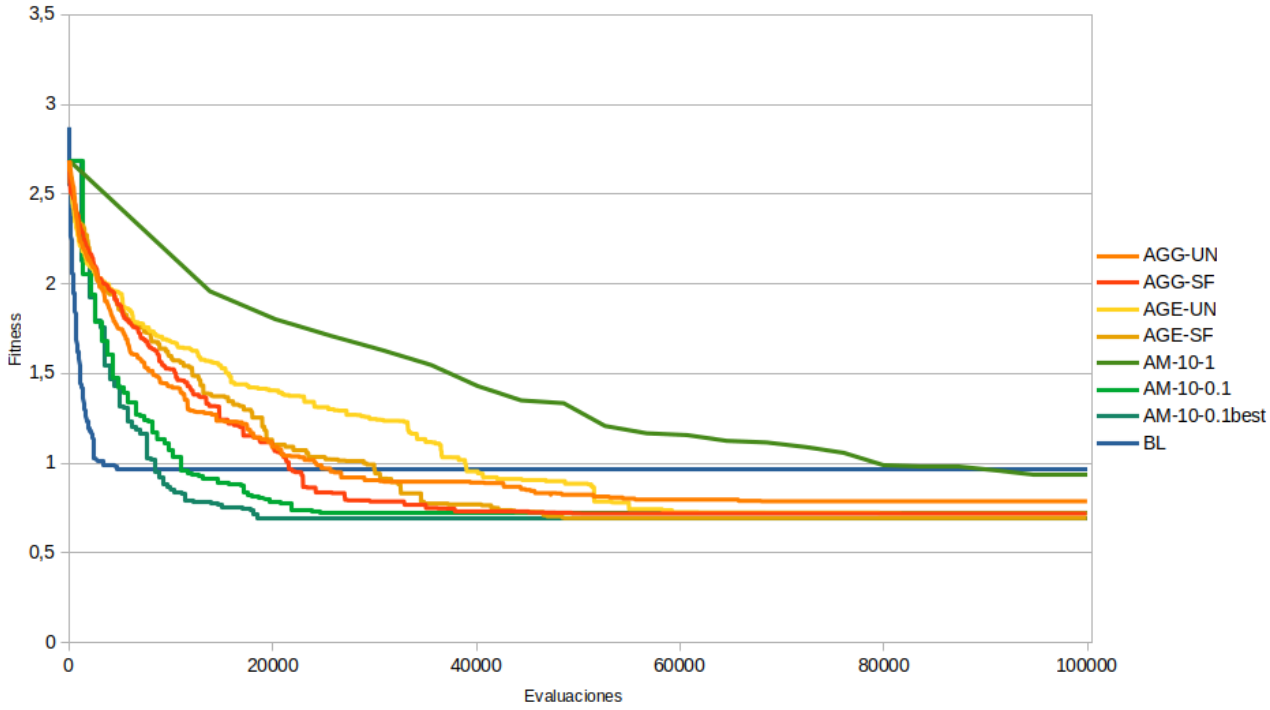
En los resultados puede observarse que los algoritmos genéticos tienen un rendimiento similar a la Búsqueda Local, cabe destacar que el fitness generado por los genéticos tiende a ser en la mayoría de los casos igual o ligeramente peor que la contraparte de BL, concuerda con la teoría de que los algoritmos genéticos son buenos exploradores pero malos explotadores, mientras que los algoritmos de Búsqueda Local son su opuesto, buenos explotadores y malos exploradores, por lo tanto, la Búsqueda Local encontrará rápidamente un óptimo, probablemente local y no moverse mucho más de allí; de hecho un ejemplo de esto es los datos de `ZOO 10` (también en la variante de 20 % hay un fitness similar) dónde una gran parte de los algoritmos genéticos y meméticos han obtenido fitness menores que la Búsqueda Local, se puede pensar que en este grupo de datos se ha conseguido obtener un óptimo local de mejor calidad que el del BL.

De la misma manera, en los datos `GLASS` sucede parecido, el fitness a veces o supera BL o se mantiene más o menos por el mismo rango, esto cambia con los datos `BUPA`, es posible inferir de los resultados que mientras más crece el problema, a los algoritmos genéticos les cuesta más evaluaciones llegar al óptimo, y más en este caso que el criterio de parada es un cierto número de evaluaciones, ya que una mutación en un cromosomas de 300 genes no afecta de la misma manera que si fuesen de 100 genes, por lo que decrece con mayor lentitud, aunque como se verá acontinuación, esto no es todo lo que afecta el resultado.

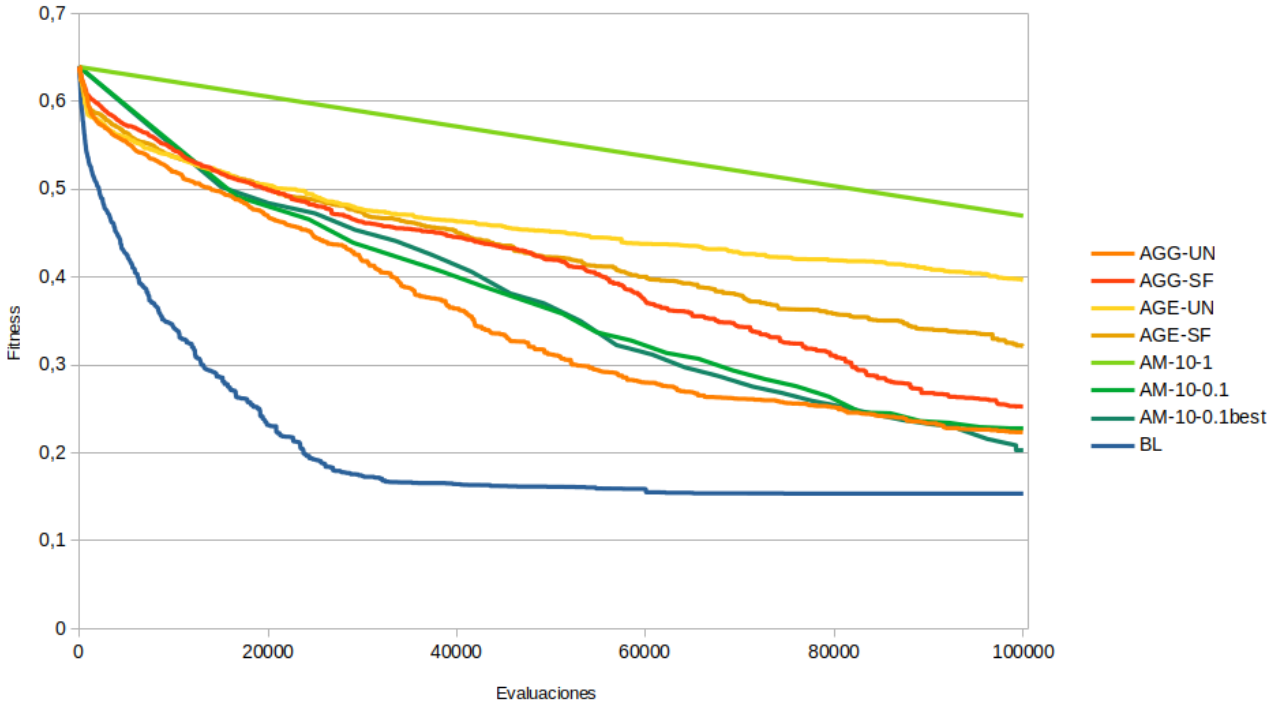
Para visualizar esto que se ha graficado el fitness por las evaluaciones de los ejemplos extremos: `ZOO` y `BUPA`. En primer lugar, en ambos se puede ver como el BL rápidamente explota la solución y no ha realizado ni la mitad de las 100000 iteraciones cuando ya ha encontrado un óptimo local. Por un lado, en `ZOO` esto hace que sea una solución de peor calidad, el algoritmo genético que si bien si tarda más en converger, lo hace en una solución de mayor calidad, exceptuando el AM-10-1,

esto se discute más adelante pero se puede ver que el tomar tantos individuos para realizar la BLS al final resulta contraproducente.

Por parte del BUPA, es fácil notar que 100000 evaluaciones se quedan cortas, todos los algoritmos genéticos y meméticos tienen todavía una tendencia decreciente, ninguno ha caído en un óptimo –lo cual se vería como una meseta, como en ZOO– seguramente si se le otorgaran más evaluaciones, daría un resultado comparable al que se obtiene con ZOO y GLASS.



Fitness en función de las evaluaciones, ZOO, 10 % Restricciones



Fitness en función de las evaluaciones, BUPA, 10 % Restricciones

Esto corrobora el hecho de que los algoritmos genéticos a diferencia del BL, exploran más y mientras más crece el tamaño del problema, la brecha entre estas dos clases de algoritmos se vuelve más notoria.

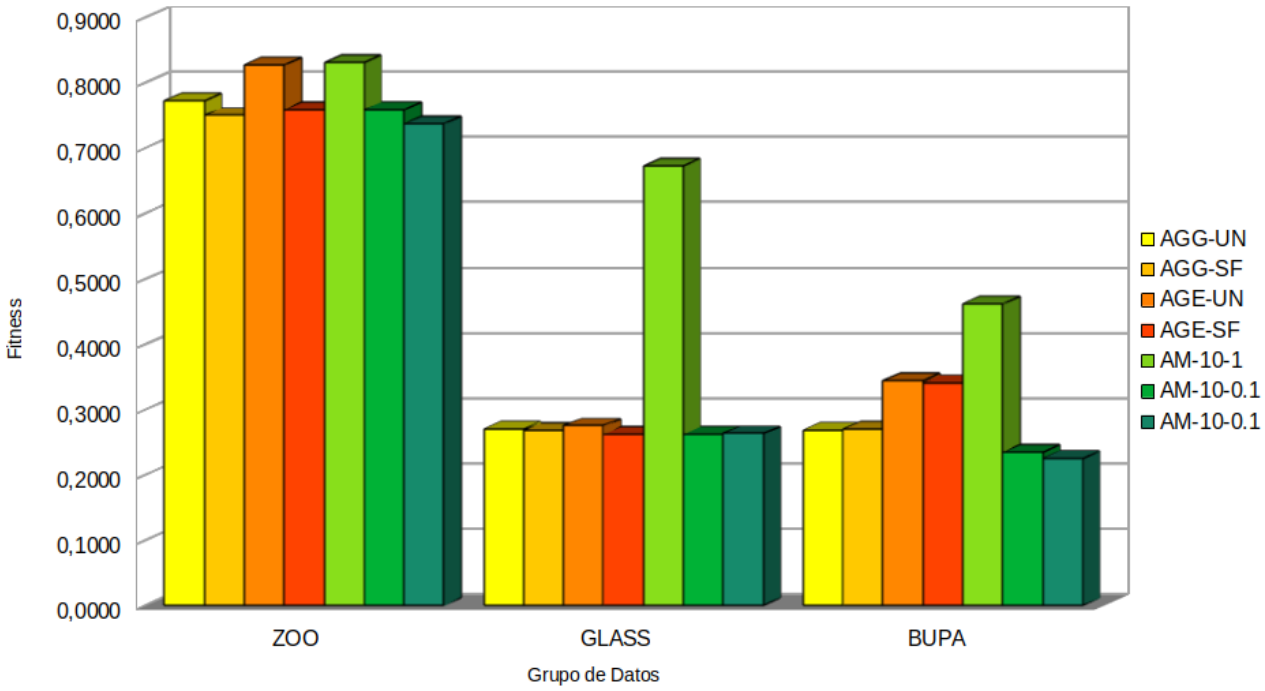
Comparando entre los propios algoritmos genéticos, si se toman los 6 grupos de datos, tanto de 10 % como de 20 % de Restricciones se observa que al menos en las ejecuciones realizadas, los algoritmos generacionales con elitismo y los estacionarios poseen un rendimiento similar, pero, si existe una diferencia notable en el tipo de cruce, el cruce por segmento fijo consistentemente es el que reduce más el fitness en todos los grupos de datos, esto se le puede atribuir a que realiza un mejor trabajo de exploración y explotación, como se ha explicado en el apartado correspondiente, cruza tanto al peor como al mejor padre. Aunque cabe notar la diferencia es muy ligera.

	ZOO	GLASS	BUPA
Mejor Fitness	0,7504	0,2623	0,2689
Algoritmo	**AGG-SF**	**AGE-SF**	**AGG-UN**

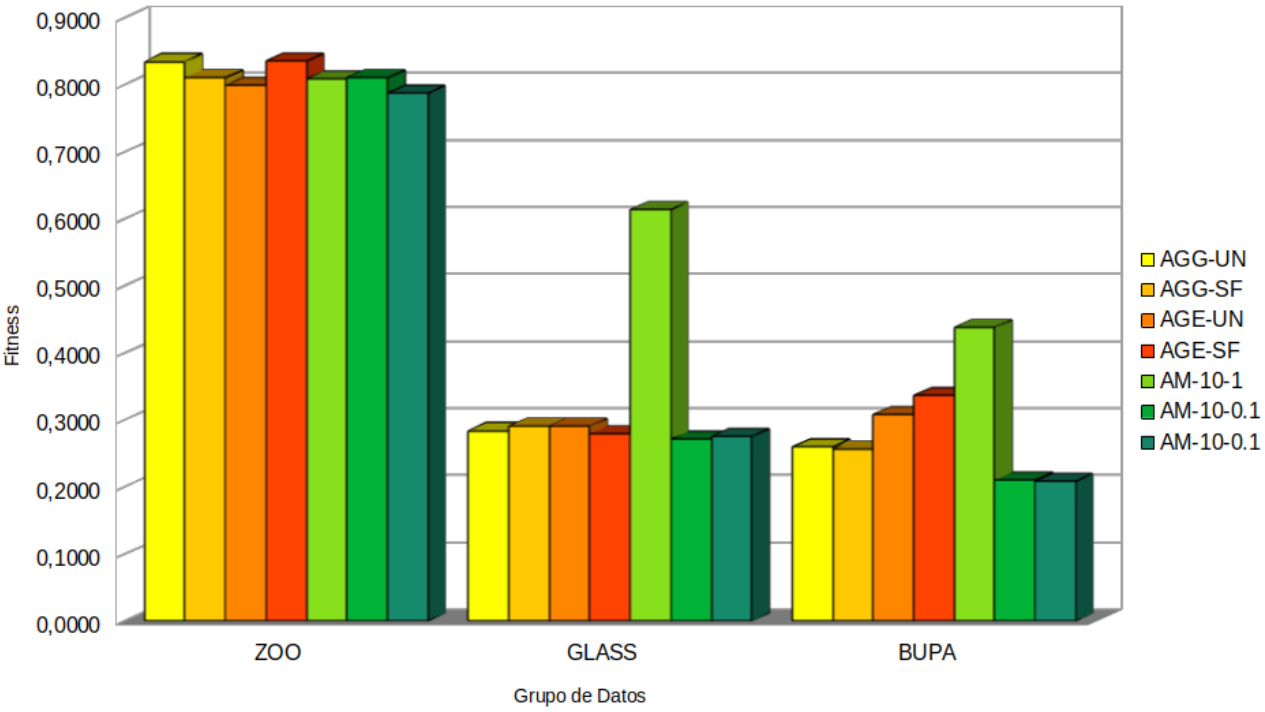
10 % Restricciones

	ZOO	GLASS	BUPA
Mejor Fitness	0,7986	0,2794	0,2564
Algoritmo	**AGE-UN**	**AGE-SF**	**AGG-SF**

20 % Restricciones



Comparación de Fitness para 10 % Restricciones



Comparación de Fitness para 20 % Restricciones

Esto da a conocer que escoger los parámetros adecuados, los tipos de cruce en este caso, puede tener una repercusión por más pequeña que sea en la calidad de la solución.

Comparado con el BL, un algoritmo genético tiene muchos más parámetros, y si se deseara optimizar el algoritmo para un cierto grupo de datos, naturalmente requeriría un estudio de qué parámetros –ya sea probabilidad de cruce o mutación– encajan mejor para los datos. Esto se pudo visualizar debido a un cambio de parámetros de la práctica, principalmente en el BUPA más que en cualquier otro grupo de datos. Inicialmente se tenía la esperanza que iban a mutar 18 genes de 18 cromosomas diferentes dada la probabilidad de mutación $p = 0{,}001$, luego esto se cambió a $p = {}^{0{,}1}/_{n}$, con n siendo el tamaño del gen, donde ahora solamente habría una esperanza de 5 mutaciones; los resultados obtenidos con la probabilidad anterior daban una reducción mayor del fitness que con la nueva.

Probabilidad Mutación $p = {}^{0{,}1}/_{n}$								
Zoo			**Glass**			**Bupa**		
Infeasable	**Distancia**	**Fitness**	**Infeasable**	**Distancia**	**Fitness**	**Infeasable**	**Distancia**	**Fitness**
19	0,631091	0,787469	24	0,256269	0,289495	342	0,156946	0,278395
11	0,620266	0,710801	43	0,199159	0,258689	418	0,152449	0,300886
13	0,622592	0,729588	23	0,249045	0,280886	252	0,133959	0,223448
8	0,619921	0,685765	54	0,18905	0,263808	283	0,164	0,264497
25	0,744967	0,950729	25	0,228469	0,263079	351	0,152847	0,277492
15,20	**0,6478**	**0,7729**	**33,80**	**0,2244**	**0,2712**	**329,20**	**0,1520**	**0,2689**
Probabilidad Mutación $p = 0{,}001$								
Zoo			**Glass**			**Bupa**		
Infeasable	**Distancia**	**Fitness**	**Infeasable**	**Distancia**	**Fitness**	**Infeasable**	**Distancia**	**Fitness**
12	0,621508	0,720273	31	0,214701	0,257617	166	0,136007	0,194956
13	0,606921	0,713917	25	0,243714	0,278324	184	0,146747	0,212088
10	0,671833	0,754138	62	0,189554	0,275387	207	0,144976	0,218485
13	0,659224	0,76622	21	0,237743	0,266816	195	0,130648	0,199895
14	0,620267	0,735493	30	0,241584	0,283116	169	0,138763	0,198777
12,40	**0,6360**	**0,7380**	**33,80**	**0,2255**	**0,2723**	**184,20**	**0,1394**	**0,2048**

Ejecuciones del AGG-UN con 10 % de Restricciones, utilizando la misma semilla y variando la probabilidad de mutación únicamente

Esto sirve como otro ejemplo para argumentar que los algoritmos genéticos tienen una mayor dispersión en sus resultados y son susceptibles a todos los parámetros que poseen, como puede observarse, un cambio relativamente pequeño puede dar resultados muy distintos. De igual manera, se pudo comprobar que, los algoritmos genéticos son más lentos pero también pueden dar muy buenos resultados, comparables a los del BL, esto sin incluir los meméticos.

Cabe notar que esta susceptibilidad incluye a la semilla inicial: esto se puede observar en que dentro de las 5 ejecuciones de un mismo algoritmo con diferentes semillas se pueden obtener resultados muy buenos y también muy malos.

Los meméticos son la fusión de estas dos metaheurísticas que toman lo mejor de cada una, la exploración de los genéticos y la explotación del BL, y se nota, los resultados obtenidos si bien, no logran a veces igualar al BL, ciertamente mejoran mucho más los resultados que los genéticos proveen, todo esto también depende mucho de los parámetros: El algoritmo memético que hace uso de toda la población en la BLS ha obtenido resultados abismales en todos los datos, y esto se nota más y más con grupos de datos más grandes ya que se están gastando muchísimas evaluaciones dentro de explotación y muy poco en la exploración, por lo tanto, dan un resultado peor, pero

utilizando el 10 % se nota una mejoría y esta aumenta mucho más si se hace uso de los mejores de la población, estos son los que han dado los mejores resultados de los meméticos.

Se puede concluir entonces que, los algoritmos genéticos poseen un gran potencial para dar resultados de calidad, posiblemente mejores que los que podría obtener la Búsqueda Local pero sufren de ser algoritmos sumamente lentos y que son muy sensibles a los parámetros que poseen, esto se puede mitigar haciendo uso de los meméticos. Si se desease resolver un problema real con esta clase de algoritmos, se debe dedicar un buen tiempo a la experimentación con diferentes parámetros, cruces y semillas (y tiempo) hasta dar con la combinación que más favorece a los datos a tratar, no obstante, para los parámetros de esta práctica, los genéticos y meméticos quedan muy cerca de la Búsqueda Local pero siempre uno o dos "pasos" detrás en la calidad de la solución.