

---

# **PRÁCTICA 3: TÉCNICAS DE BÚSQUEDA BASADAS EN TRAYECTORIAS**

---

**Problema: Aprendizaje de Pesos en Características**  
**Algoritmos: BMB, ES, ILS, ILS-ES, VNS**

Juan Manuel Rodríguez Gómez  
49559494Z  
juanmaarg6@correo.ugr.es

Metaheurísticas, Grupo 1 (Lunes 17:30 – 19:30)  
Curso 2022 – 2023



**UNIVERSIDAD  
DE GRANADA**

# Índice

1. Descripción del Problema.....	1
2. Descripción de la Aplicación de los Algoritmos Empleados al Problema .....	2
2.1. Esquema de Representación de Soluciones (Práctica 1).....	2
2.2. Esquema de Representación de Soluciones (Práctica 2).....	2
<b>2.3. Esquema de Representación de Soluciones (Práctica 3).....</b>	<b>3</b>
2.4. Operadores Comunes (Práctica 1) .....	4
2.5. Operadores Comunes (Práctica 2) .....	5
<b>2.6. Operadores Comunes (Práctica 3) .....</b>	<b>7</b>
2.7. Función Objetivo .....	8
3. Descripción de los Algoritmos Implementados.....	10
3.1. Algoritmos Implementados (Práctica 1) .....	10
3.1.1. Algoritmo Greedy RELIEF .....	10
3.1.2. Algoritmo de Búsqueda Local .....	11
3.2. Algoritmos Implementados (Práctica 2) .....	14
3.2.1. Algoritmos Genéticos Generacionales.....	16
3.2.2. Algoritmos Genéticos Estacionarios .....	18
3.2.3. Algoritmos Meméticos .....	20
<b>3.3. Algoritmos Implementados (Práctica 3) .....</b>	<b>23</b>
<b>3.3.1. Algoritmo Búsqueda Multiarranque Básica .....</b>	<b>23</b>
<b>3.3.2. Algoritmo Enfriamiento Simulado .....</b>	<b>24</b>
<b>3.3.3. Algoritmo Búsqueda Local Iterativa .....</b>	<b>26</b>
<b>3.3.4. Algoritmo Búsqueda de Vecindario Variable .....</b>	<b>28</b>
4. Procedimiento Considerado para Desarrollar la Práctica .....	31
5. Experimentos y Análisis de Resultados .....	32
<b>5.1. Resultados Obtenidos .....</b>	<b>32</b>
5.2. Análisis de los Resultados Obtenidos (Práctica 1).....	36
5.3. Análisis de los Resultados Obtenidos (Práctica 2).....	38
<b>5.4. Análisis de los Resultados Obtenidos (Práctica 3).....</b>	<b>44</b>
5.5. Conclusiones (Práctica 1).....	50
5.6. Conclusiones (Práctica 2).....	50
<b>5.7. Conclusiones (Práctica 3).....</b>	<b>52</b>

Con el fin de mantener en un único documento toda la información relevante de las diferentes prácticas, se han incluido en esta memoria las explicaciones y descripciones de los algoritmos que ya se comentaron en las memorias de las prácticas anteriores, así como los análisis de los resultados que ya se realizaron.

Para destacar la nueva información, se marcan en rojo las nuevas secciones añadidas referente a esta práctica.

## 1. Descripción del Problema

El **Problema del Aprendizaje de Pesos en Características (APC)** es un problema que consiste en obtener un vector de pesos que permita ponderar las características asociadas a un dato con la intención de obtener un mejor porcentaje de clasificación de datos futuros. Tenemos una muestra de datos  $X = \{x_1, x_2, \dots, x_n\}$ , donde cada dato de dicha muestra está formado por un conjunto de características  $\{c_1, c_2, \dots, c_m\}$  y su CATEGORÍA, de forma que busquemos obtener un vector de pesos  $W = \{w_1, w_2, \dots, w_m\}$  donde el peso  $w_i$  pondera la característica  $c_i$  y que ese vector de pesos nos permita optimizar el rendimiento del clasificador k-NN. Nosotros usaremos el clasificador 1-NN, por lo que vamos a considerar el vecino más cercano (con el que se tiene menor **distancia euclídea ponderada por pesos**) para predecir la clase de cada dato.

A lo largo de nuestro proceso vamos a realizar dos tareas: primero obtendremos el vector de pesos (aprendizaje) y a continuación valoraremos la calidad del mismo (validación). Usaremos la técnica de validación cruzada **5-fold cross validation**. El conjunto de datos ya está dividido en 5 particiones disjuntas al 20% desordenadas, y con la misma distribución de clases. Aprenderemos un clasificador utilizando el 80% de los datos disponibles (4 particiones de las 5) y validaremos con el 20% restante (la partición restante). Así obtendremos un total de 5 valores de porcentaje de clasificación en el conjunto de prueba, uno para cada partición empleada como conjunto de validación. La calidad del método de clasificación se medirá con un único valor, correspondiente a la media de los 5 porcentajes de clasificación del conjunto de prueba.

Para la función de evaluación se combinan dos criterios:

- **Precisión**  $\rightarrow tasa\_clas = 100 * \frac{N^\circ \text{ de instancias bien clasificadas en } T}{N^\circ \text{ de instancias en } T}$
- **Simplicidad**  $\rightarrow tasa\_red = 100 * \frac{N^\circ \text{ de valores } w_i < 0.1}{N^\circ \text{ de características}}$

Utilizaremos una agregación sencilla que combine ambos objetivos en un único valor. Así, la **función objetivo** será  $F(w) = \alpha * tasa\_clas(w) + (1 - \alpha) * tasa\_red(w)$ . El valor de  $\alpha$  pondera la importancia entre el acierto y la reducción de características de la solución encontrada (el clasificador generado). Usaremos  $\alpha = 0.8$ , dando más importancia al acierto. El objetivo es obtener el conjunto de pesos  $W$  que maximiza esta función, es decir, que maximice el acierto del clasificador 1-NN y, a la vez, que considere el menor número de características posible.

Los distintos algoritmos se van a ejecutar sobre **tres conjuntos de datos distintos: diabetes, ozone y spectf-heart**. Hay que tener en cuenta que debemos normalizar los datos.

## 2. Descripción de la Aplicación de los Algoritmos Empleados al

### Problema

#### 2.1. Esquema de Representación de Soluciones (Práctica 1)

Para representar los datos en el programa se emplea un **struct Ejemplo** que recoge toda la información necesaria: un **vector<double>** con los valores de cada una de las  $n$  características del ejemplo concreto, así como un **string** que representa su clase o categoría.

---

```
struct Ejemplo {  
    vector<double> val_caracts;  
    string categoria;  
    int num_caracts;  
}
```

---

Cada **conjunto de datos** se almacena en un **vector<Ejemplo>** y se emplean las funciones *leerFicheroARFF* y *normalizarValores* para leer los datos del conjunto, normalizarlos y almacenarlos en dicho vector.

Por otro lado, la **solución** a nuestro problema es un vector de pesos,  $w$ , representando mediante **vector<double>**, de  $n$  componentes (siendo  $n$  el número de características de los ejemplos de nuestro conjunto de datos). En cada componente del vector tendremos un valor real  $w_i \in [0, 1]$ , que representa el peso asociado a la característica  $i$ -ésima.

#### 2.2. Esquema de Representación de Soluciones (Práctica 2)

Para desarrollar los diferentes algoritmos genéticos, empleamos un **struct** que representa un **cromosoma**. En este contexto, un cromosoma será un vector de pesos (una posible solución) ya evaluado, es decir, con un valor otorgado por la función objetivo (*fitness*).

---

```
struct Cromosoma {  
    vector<double> w;  
    double fitness;  
}
```

---

El valor por defecto del *fitness* de un cromosoma es -1, lo cual resulta útil para saber que un cromosoma no está correctamente evaluado.

Para representar una **población** de soluciones (un conjunto de cromosomas) utilizamos el tipo de dato **multiset** (contenedor asociativo y ordenado que admite repetidos). Es importante permitir repetidos, pues podemos tener una población con varios cromosomas iguales. Definimos la siguiente relación de orden entre cromosomas: *Si  $C_1$  y  $C_2$  son cromosomas, diremos que  $C_1 < C_2$  si  $C_1.fitness < C_2.fitness$ .*

---

```
struct CromosomaComp {  
    bool operator()(const Cromosoma& lhs, const Cromosoma& rhs) const {  
        return lhs.fitness < rhs.fitness  
    }  
}
```

---

---

```
typedef multiset<Cromosoma, CromosomaComp> Poblacion
```

---

De esta forma, cuando queramos obtener el mejor cromosoma de la población, simplemente tendremos que mirar el último elemento del contenedor, y si queremos obtener el peor cromosoma de la población pues miraremos el primer elemento del contenedor.

Respecto a la representación de la población, hay ciertas fases de los diferentes algoritmos en los que no es necesario mantener los cromosomas ordenados. Por ello, introducimos un nuevo tipo de dato llamado **PoblacionIntermedia** que no es más que un **vector de cromosomas**, aprovechando así la posibilidad de acceso aleatorio que ofrece la clase *vector*.

---

```
typedef vector<Cromosoma> PoblacionIntermedia
```

---

### **2.3. Esquema de Representación de Soluciones (Práctica 3)**

Para el desarrollo de los diferentes algoritmos de esta práctica, empleamos un **struct** que representa una **solución**, la cual está compuesta por un vector de pesos junto con su valor de la función objetivo (*fitness*) correspondiente.

---

```
struct Solución {  
    vector<double> w;  
    double fitness;  
}
```

---

## 2.4. Operadores Comunes (Práctica 1)

Todos los algoritmos hacen uso del cálculo de la distancia. Como dijimos, para este cálculo se emplea **la distancia euclídea ponderada mediante un vector de pesos**. En el caso de que la distancia deseada sea la estándar (es decir, sin ponderación de pesos), se asume que los pesos valen siempre uno. Se han realizado dos implementaciones de la distancia euclídea, una con pesos y otra sin pesos.

Como simplemente usamos la distancia para comparar, la vamos a implementar como si la eleváramos al cuadrado, es decir, eliminando la raíz cuadrada (podemos hacerlo sin problema ya que se mantiene el orden.). Hacemos esto para ahorrar tiempo de cálculo (ya que esta es la función más llamada a lo largo del programa).

---

```
función distanciaEuclidea(e1, e2)  
    distancia = 0  
    for i:=0 to n-1 do                (n es el número de características)  
        distancia += (e1[i].val_caracts - e2[i].val_caracts) *  
                    * (e1[i].val_caracts - e2[i].val_caracts)
```

---

---

```
función distanciaEuclideaPesos(e1, e2, w)  
    distancia = 0  
    for i:=0 to n-1 do                (n es el número de características)  
        if w[i] > 0.1 then  
            distancia += w[i] * (e1[i].val_caracts - e2[i].val_caracts) *  
                            * (e1[i].val_caracts - e2[i].val_caracts)
```

---

También tenemos la función **clasificador1NNPesos**, que clasifica un ejemplo basándose en la técnica del vecino más cercano. Debemos pasarle también el conjunto de entrenamiento (donde hay otros ejemplos ya están clasificados), y el vector de pesos. Al igual que pasaba con la distancia euclídea, si queremos que el clasificador no tenga en cuenta los pesos, hacemos que todas las componentes del vector de pesos sean 1.

---

```

función clasificador1NNPesos(e, entrenamiento, w)
    distancia_min = +infinito
    for i:=0 to n-1 do           (n es el número de ejemplos de entrenamiento)
        distancia = distanciaEuclideaPesos(e, entrenamiento[i], w)
        if distancia < distancia_min then
            distancia_min = distancia
            clase = entrenamiento[i].categoria
    return clase

```

---

## 2.5. Operadores Comunes (Práctica 2)

En lo que sigue, para generar números aleatorios reales o enteros, utilizamos los tipos predefinidos ***uniform\_real\_distribution*** y ***uniform\_int\_distribution***, respectivamente.

Disponemos de una función ***inicializarPoblacion*** que se encarga de generar la población inicial siguiendo una distribución uniforme en el intervalo [0, 1] para el vector de pesos de cada cromosoma que se añade a la población.

---

```

función inicializarPoblacion(n, m, entrenamiento)
    poblacion = ∅
    for i:=0 to n-1 do           (n es el número de cromosomas a crear)
        c = ∅
        for j:=0 to m-1 do       (m es el número de genes por cromosoma)
            c.w[j] = uniformInt(0,1)
        c.fitness = fitness(
            tasaClasificacionLeaveOneOut(entrenamiento,c.w),
            tasaReduccion(c.w)
        )
        poblacion.insert(c)
    return poblacion

```

---

Describamos ahora los diferentes operadores usados para los algoritmos genéticos (selección, cruces y mutación). Comenzamos con el operador de **selección**, el cual realiza un torneo binario entre dos padres (cromosomas) escogidos al azar y devuelve aquel que tenga mayor valor de *fitness*. Como tenemos los cromosomas de una población almacenados en orden de menor a mayor a *fitness*, tenemos que iterar aleatoriamente sobre dicha población a la hora de escoger los dos padres.



---

**función seleccion(poblacion)**

```
(n es el tamaño de la población)

padre1 = advance(poblacion.begin(), uniformInt(0, n-1) )
padre2 = advance(poblacion.begin(), uniformInt(0, n-1) )
return padre1->fitness < padre2->fitness ? *padre2 : *padre1
```

---

Por otro lado, tenemos el operador de **cruce BLX**, que recibe dos cromosomas padres y devuelve dos cromosomas descendientes cuyos pesos se obtienen de forma aleatoria en un intervalo concreto dependiente de los vectores de pesos de los padres. La exploración que realiza está determinada por el valor del parámetro *alpha\_blx*, el cual es de 0.3. Nótese que se podrían generar pesos fuera del intervalo [0, 1], por lo que, en caso de que sea necesario, habría que truncarlos a 0 o 1.

---

**función cruceBLX(c1, c2)**

```
(n es el número de genes de los cromosomas)

for i:=0 to n-1 do
    c_min = min(c1.w[i], c2.w[i])
    c_max = max(c1.w[i], c2.w[i])
    dif = c_max - c_min
    h1.w[i] = uniformReal(c_min - dif*alpha_blx, c_max + dif*alpha_blx)
    h2.w[i] = uniformReal(c_min - dif*alpha_blx, c_max + dif*alpha_blx)
    if h1.w[i] < 0 then
        h1.w[i] = 0
    if h1.w[i] > 1 then
        h1.w[i] = 1
    if h2.w[i] < 0 then
        h2.w[i] = 0
    if h2.w[i] > 1 then
        h2.w[i] = 1
    h1.fitness = -1
    h2.fitness = -1
return (h1, h2)
```

---

También contamos con el operador de **cruce aritmético**, el cual recibe dos cromosomas padres y devuelve un único cromosoma descendiente cuyos pesos se obtienen realizando la media aritmética componente a componente de los pesos de los padres.

---

```
función cruceAritmetico(c1, c2)
    for i:=0 to n-1 do          (n es el número de genes de los cromosomas)
        h.w[i] = (c1.w[i] + c2.w[i]) / 2
    h.fitness = -1
    return h
```

---

Finalmente, tenemos el operador de **mutación**, que simplemente altera un gen (una componente del vector de pesos) del cromosoma según un valor aleatorio de una distribución normal de *media* = 0 y desviación típica *sigma* =  $\sqrt{0.3}$ . En caso de ser necesario, truncamos el peso asociado a dicha componente a 0 o 1.

---

```
función mutacion(c, comp)
    c.w[comp] += normal(0, sqrt(0.3))
    c.fitness = -1
    if c.w[comp] < 0 then
        c.w[comp] = 0
    if c.w[comp] > 1 then
        c.w[comp] = 1
    return c
```

---

## 2.6. Operadores Comunes (Práctica 3)

En primer lugar, disponemos de una función **inicializarSolucion** que se de generar una solución inicial aleatoria siguiendo una distribución uniforme [0, 1] para los pesos. Esta solución generada se devuelve ya evaluada (es decir, con su *fitness* correspondiente).

---

```

función inicializarSolucion(n, entrenamiento)
    sol = {}
    for i:=0 to n-1 do                                (n es el tamaño del vector de pesos)
        sol.w[i] = uniformReal(0, 1)
    sol.fitness = fitness(
        tasaClasificacionLeaveOneOut(entrenamiento, sol.w),
        tasaReduccion(sol.w)
    )
    return sol

```

---

También contamos con una función para realizar una **mutación fuerte** en los algoritmos ILS, ILS-ES y VNS. En dicha mutación fuerte, cambiamos el 10% de las características de un vector de pesos por valores aleatorios de una distribución uniforme [0, 1]. Como mínimo, se cambiarán al menos 2 característica.

---

```

función mutacionFuerte(w)
    num_mutaciones = min(0.1 * w.size(), 2)
    indices = {}
    for i:=0 to n-1 do                                (n es el tamaño del vector de pesos)
        indices.push_back(i)
    shuffle(indices)
    for i:=0 to num_mutaciones do
        w[indices[i]] = uniformReal(0, 1)
    return w

```

---

## 2.7. Función Objetivo

La función objetivo que queremos maximizar se implementa tal y como se dijo en la descripción del problema, donde el valor de *alpha* es de 0.8, dando más importancia al acierto que a la reducción. De igual forma, para la tasa de clasificación y la tasa de reducción, usamos las expresiones dadas en la descripción del problema.

---

```

función fitness(tasa_clas, tasa_red)
    return (alpha * tasa_clas) + ( (1 - alpha) * tasa_red )

```

---

---

```
función tasaClasificacion(test, entrenamiento)
    num_instancias_bien_clas = 0
    for i:=0 to n-1 do                                (n es el número de ejemplos de test)
        if clasificador1NNPesos(test[i], entrenamiento, w) ==
            == test[i].categoria then
                num_instancias_bien_clas++
    return 100.0 * num_instancias_bien_clas / n
```

---

---

```
función tasaReduccion(w)
    num_caracts_descartadas = 0
    for i:=0 to n-1 do                                (n es el tamaño del vector de pesos)
        if w[i] < 0.1 then
            num_caracts_descartadas++
    return 100.0 * num_caracts_descartadas / n
```

---

También se ha creado otra función para calcular la tasa de clasificación usando la técnica *Leave-One-Out* empleada en el algoritmo de búsqueda local.

---

```
función tasaClasificacionLeaveOneOut(entrenamiento, w)
    aux = entrenamiento
    num_instancias_bien_clas = 0
    for i:=0 to n-1 do                                (n es el número de ejemplos de aux)
        ejemplo = aux[i]
        aux.erase(ejemplo)
        if clasificador1NNPesos(ejemplo, aux, w) == ejemplo.categoria then
            num_instancias_bien_clas++
        aux.insert(ejemplo)                            (en la misma posición donde estaba)
    return 100.0 * num_instancias_bien_clas / n
```

---

Nótese que en las tasas de clasificación tenemos incorporada la función *clasificador1NNPesos* para así evaluar un vector de pesos sobre el conjunto correspondiente (test o entrenamiento) y pasar directamente el valor de dicha tasa a la función fitness para calcular directamente el valor de la función objetivo.

### 3. Descripción de los Algoritmos Implementados

En esta sección se describen los dos algoritmos implementados en la primera práctica para el problema del Aprendizaje de Pesos en Características: Greedy RELIEF y Búsqueda Local. En ambos lo que se pretende es rellenar un vector de pesos para maximizar la función objetivo.

#### 3.1. Algoritmos Implementados (Práctica 1)

**Compararemos los resultados obtenidos por el algoritmo de búsqueda local con los resultados obtenidos por los nuevos algoritmos implementados en esta práctica 3.**

##### 3.1.1. Algoritmo Greedy RELIEF

Este es un algoritmo voraz muy sencillo, que **servirá como caso base para comparar las diferentes metaheurísticas desarrolladas**. Se trata de buscar, para cada ejemplo, su amigo (otro ejemplo que tenga la misma clase) y su enemigo (otro ejemplo que tenga distinta clase) más cercano. Después, componente a componente, se suma al vector de pesos la distancia a su enemigo y se resta la distancia a su amigo.

En este proceso es posible que los pesos se salgan del intervalo  $[0, 1]$ , por lo que al finalizar la suma componente a componente es necesario normalizar. Además, el vector de pesos inicialmente tiene todas sus componentes a 0 (para realizar esta inicialización disponemos de una función *inicializarVectorPesos*).

A continuación, se encuentran las funciones que se encargan de encontrar el amigo y el enemigo más cercano a un ejemplo dado (hay que tener en cuenta que el amigo más cercano no puede ser el propio ejemplo).

---

```
función amigoMasCercano(e, entrenamiento)
    distancia_min = +infinito
    for i:=0 to n-1 do          (n es el número de ejemplos de entrenamiento)
        if (e.val_caracts != entrenamiento[i].val_caracts and
            entrenamiento[i].categoria == e.categoria) then
            distancia = distanciaEuclidea(e, entrenamiento[i])
            if distancia < distancia_min then
                distancia_min = distancia
                amigo_mas_cercano = entrenamiento[i]
    return amigo_mas_cercano
```

---

---

```

función enemigoMasCercano(e, entrenamiento)
    distancia_min = +infinito
    for i:=0 to n-1 do           (n es el número de ejemplos de entrenamiento)
        if entrenamiento[i].categoria != e.categoria then
            distancia = distanciaEuclidea(e, entrenamiento[i])
            if distancia < distancia_min then
                distancia_min = distancia
                enemigo_mas_cercano = entrenamiento[i]
    return enemigo_mas_cercano

```

---

Utilizando dichas funciones, el **algoritmo Greedy RELIEF** sería el siguiente:

---

```

función greedy(entrenamiento)
    w = inicializarVectorPesos()
    for i:=0 to n-1 do           (n es el número de ejemplos de entrenamiento)
        enemigo_mas_cercano = enemigoMasCercano(entrenamiento[i], entrenamiento)
        amigo_mas_cercano = amigoMasCercano(entrenamiento[i], entrenamiento)
        for j:=0 to m-1 do       (m es el tamaño del vector de pesos)
            w[j] = w[j] +
+ |entrenamiento[i].val_caracts[j] - entrenamiento[pos_enemigo].val_caracts[j]|
- |entrenamiento[i].val_caracts[j] - entrenamiento[pos_amigo].val_caracts[j]|
        w_max = max(w)
        for j:=0 to m-1 do
            if w[j] < 0 then
                w[j] = 0
            else
                w[j] = w[j] / w_max
    return w

```

---

### 3.1.2. Algoritmo de Búsqueda Local

Empleamos la técnica de búsqueda local del **primer mejor** para rellenar el vector de pesos. La idea es mutar en cada iteración una componente aleatoria y **distinta** del vector de pesos, sumándole un valor extraído de una normal de *media* = 0 y desviación típica *sigma* =  $\sqrt{0.3}$ . Si tras esta mutación se mejora la función objetivo, nos quedamos con este nuevo vector, y si no lo desechamos. Si algún peso se sale del intervalo [0, 1] tras la mutación, directamente lo truncamos a 0 (si es menor que 0) o a 1 (si es mayor que 1).

Para la **generación de la solución inicial** sobre la que iterar, consideramos valores extraídos de una distribución uniforme  $U[0,1]$ , que obtenemos gracias al tipo `uniform_real_distribution<double>` de la librería *random*. Generamos así una solución inicial aleatoria.

---

**(\*)**

```
// Inicializamos el vector de índices y el vector de pesos (este último
// se hace aleatoriamente)
for i:=0 to m-1 do                (m es el tamaño del vector de pesos)
    indices.push_back(i)
    w[i] = uniformInt(0,1)
```

---

Para escoger el valor con el que se muta cada componente utilizamos esta vez el tipo predefinido `normal_distribution<double>` de la librería *random*. De esta forma, el **operador de generación de vecino** sería de la siguiente forma:

---

```
// Suponemos que comp_mutada es la componente del vector de pesos que
// vamos a modificar
w_mutado = w
w_mutado[comp_mutada] += normal(0, sqrt(0.3))
// Truncamos si es necesario
if w_mutado[comp_mutada] > 1 then
    w_mutado[comp_mutada] = 1
else if w_mutado[comp_mutada] < 0 then
    w_mutado[comp_mutada] = 0
```

---

Por otro lado, el **método de exploración del entorno** es el siguiente: Para escoger qué componente del vector de pesos vamos a mutar, tenemos un vector de índices del mismo tamaño que el vector de pesos, que barajamos de forma aleatoria y recorremos secuencialmente. Si llegamos al final, volvemos a barajarlo para seguir generando nuevas soluciones. Para **comprobar si se mejora el fitness con una nueva solución**, se realiza lo siguiente:

---

**(\*\*)**

```
fitness_actual = fitness(
    tasaClasificacionLeaveOneOut(entrenamiento,w),
    tasaReduccion(w)
)
```

---

Con todo lo anterior, el algoritmo de Búsqueda Local sería el siguiente:

---

```
función busquedaLocal(entrenamiento)
    num_iteraciones = 0
    num_vecinos = 0
    hay_mejora = false
    w, indices = (*)          (Código de inicialización descrito anteriormente)
    shuffle(indices)
    mejor_fitness = (**) con w (Código de evaluación descrito anteriormente)
    while(num_iteraciones < MAX_ITER and num_vecinos < m*COEF_MAX_VECINOS) do
        (m es el tamaño del vector de pesos,
         MAX_ITER = 15000,
         COEF_MAX_VECINOS = 20)

        comp_mutada = index[num_iteraciones % m]
        w_mutado = w
        w_mutado[comp_mutada] += normal(0, sqrt(0.3))
        // Truncamos si es necesario
        if w_mutado[comp_mutada] > 1 then
            w_mutado[comp_mutada] = 1
        else if w_mutado[comp_mutada] < 0 then
            w_mutado[comp_mutada] = 0
        fitness_actual = (**) con w_mutado
        num_iteraciones++
        if fitness_actual > mejor_fitness then
            num_vecinos = 0
            w = w_mutado
            mejor_fitness = fitness_actual
            hay_mejora = true
        else
            num_vecinos++
        if (num_iteraciones % m == 0 or hay_mejora) then
            shuffle(indices)
            hay_mejora = false
    return w
```

---



### 3.2. Algoritmos Implementados (Práctica 2)

Compararemos los resultados obtenidos por el algoritmo AM-(10,0.1) / AM\_Rand (ya que este es el que mejor resultados nos proporcionó en la práctica anterior) con los resultados obtenidos por los nuevos algoritmos implementados en esta práctica 3.

Los diferentes algoritmos desarrollados en esta práctica comparten una serie de **comportamientos comunes** que detallamos a continuación con el fin de evitar repeticiones.

Por un lado, en todos ellos tenemos un contador **num\_iteraciones** para llevar la cuenta del número de evaluaciones que llevamos de la función objetivo. Saldremos del bucle principal cuando **num\_iteraciones** sea mayor que 15000.

Por otro lado, la evolución de la población inicial se lleva a cabo generalmente mediante una población intermedia **p\_intermedia**, de tipo PoblacionIntermedia, a la cual podemos acceder por índices debido a que no es más que un vector de cromosomas. El proceso de **selección de un número concreto (n) de padres** se realiza siempre de la misma forma:

---

**SELECCIONAR (n) :**

```
for i:=0 to n-1 do
    p_intermedia[i] = seleccion(p)
```

---

El **proceso de cruce de 2n padres** sigue también un esquema similar, pero hay pequeñas diferencias (de implementación) entre el cruce BLX y el cruce aritmético. Llamando  $t$  al tamaño de la población intermedia, en el cruce aritmético nos aseguraremos de que  $4n \leq p$ , para que no disminuya el tamaño de la población (recordemos que este cruce nos devuelve un solo hijo por cada dos padres).

---

**BLX (n) :**

```
for i:=0 to 2*n - 1 step 2 do
    p_intermedia[i], p_intermedia[i+1] = cruceBLX(p_intermedia[i],
                                                    p_intermedia[i+1])
```

---

---

```
ARITMETICO(n, t):
```

```
  for i:=0 to 2*n - 1 do
    p_intermedia[i] = cruceAritmetico(p_intermedia[i],
                                     p_intermedia[2*t - i - 1])
```

---

Para el esquema de mutación tenemos dos alternativas:

- 1) **Calcular el número esperado de mutaciones de genes**, es decir,  $\text{mutaciones\_esperadas} = (P_m / N) * M$ , donde  $P_m$  es la probabilidad de que un cromosoma mute,  $N$  es el número de genes del cromosoma de la población intermedia y  $M$  es el número total de genes de toda la población intermedia. En nuestro caso,  $P_m = 0.1$ .

En vez de truncar el valor obtenido, consideramos su parte entera ( $PE$ ) y su parte decimal ( $PD$ ) y, en cada iteración del algoritmo se realizan, al menos, tantas mutaciones como indique  $PE$ . Establecemos la posibilidad de que se realice una mutación extra, generando en cada iteración un número aleatorio  $u$  entre 0 y 1 y sumando 1 a  $\text{mutaciones\_esperadas}$  si  $u \leq PD$ .

Además, como queremos introducir diversidad en la población, imponemos que al menos se realice una mutación en cada generación.

---

```
(ngc es el número de genes de cada cromosoma)
(ngt es el número total de genes de toda la población)

función mutacionesEsperadas(ngc, ngt)
  mutaciones_esperadas = (Pm / ngc) * ngt
  if mutaciones_esperadas <= 1 then
    return 1
  resto = modf(mutaciones_esperadas, &mutaciones_esperadas)
  if uniformReal(0,1) <= resto then
    mutaciones_esperadas++
  return mutaciones_esperadas
```

---

También es importante tener en cuenta que, al sustituir la comprobación en cada gen por realizar un número fijo de mutaciones, debemos **evitar mutar dos veces el mismo gen**. Para ello, usamos un **set** llamado **mutados**, que no admite repetidos.

Usaremos este esquema de mutación en los **algoritmos genéticos generacionales** y en los **algoritmos meméticos**.

---

```

                                (ngc es el número de genes de cada cromosoma)
                                (ngt es el número total de genes de toda la población)
MUTAR_1(ngc, ngt)
    mutados = ∅
    num_mut_esperadas = mutacionesEsperadas(ngc, ngt)
    for i:=0 to num_mut_esperadas - 1 do
        while mutados.size() == i do
            comp = uniformInt(0, ngt - 1)
            mutados.insert(comp)
            comp_seleccionada = comp / w.size()      (w es un vector de pesos)
            gen_seleccionado = comp % w.size()
            mutacion(p_intermedia[comp_seleccionada], gen_seleccionado)

```

---

- 2) **Realizar las mutaciones a nivel de cromosoma.** Esta alternativa es mejor cuando el tamaño de la población intermedia es pequeño, pues en esos casos el número esperado de mutaciones según la primera alternativa será casi siempre 0. De esta forma, si  $P_m$  es la probabilidad de que un cromosoma mute:

---

```

                                (n es el tamaño de la población intermedia)
                                (ngc es el número de genes de cada cromosoma)
MUTAR_2(n, ngc)
    for i:=0 to n- 1 do
        if uniformReal(0,1) <=  $P_m$  then
            gen_seleccionado = uniformInt(0, ngc - 1)
            mutacion(p_intermedia[i], gen_seleccionado)

```

---

Usaremos este esquema de mutación en los **algoritmos genéticos estacionarios**.

### 3.2.1. Algoritmos Genéticos Generacionales

Hemos implementado dos algoritmos genéticos generacionales, uno con cruce BLX (**AGG\_BLX**) y otro con cruce aritmético (**AGG\_Arit**). En ambos, la población será de **TAMANIO\_AGG = 50**. Hay que tener en cuenta que el cruce aritmético produce un hijo por cada dos padres (a diferencia del cruce BLX, que produce dos hijos por cada dos padres), por lo que a la hora de seleccionar los padres para el torneo binario y cruzar, tendremos que escoger  $2 * \text{TAMANIO\_AGG}$  padres (es decir, escoger el doble para seguir manteniendo una población de **TAMANIO\_AGG** individuos tras el cruce aritmético).

También cabe destacar que la probabilidad de cruce es  $P_c = 0.7$ , luego, en cada iteración realizamos el número de cruces esperado, el cual es igual a la parte entera de  $P_c * (TAMANIO\_AGG / 2)$ .

Por tanto, el pseudocódigo del **algoritmo genético generacional con cruce BLX** sería el siguiente:

---

**función AGG\_BLX(entrenamiento)**

```

    w(entrenamiento[0].num_caracts)    (w es el vector de pesos a calcular)
    p = inicializarPoblacion(TAMANIO_AGG, w.size(), entrenamiento)
    num_iteraciones = TAMANIO_AGG
    num_generaciones = 1
    num_genes_individuo = w.size()
    num_total_genes = w.size() * TAMANIO_AGG
    num_cruces_esperados =  $P_c * (TAMANIO\_AGG / 2)$ 
    while num_iteraciones < MAX_ITER do                                     (MAX_ITER = 15000)
        p_intermedia = Ø                                                    (No ordenada)
        nueva_p = Ø                                                         (Ordenada)
        mejor_padre = p.end()
        SELECCIONAR(TAMANIO_AGG)
        BLX(num_cruces_esperados)
        MUTAR_1(num_genes_individuo, num_total_genes)
        for i:=0 to TAMANIO_AGG do                                         (Reemplazo)
            if p_intermedia[i].fitness == -1 then
                p_intermedia[i].fitness = fitness(
                    tasaClasificacionLeaveOneOut(entrenamiento,p_intermedia[i].w),
                    tasaReduccion(p_intermedia[i].w)
                )
                num_iteraciones++
            nueva_p.insert(p_intermedia[i])
        mejor_padre_actual = nueva_p.end()
        if mejor_padre_actual->fitness < mejor_padre->fitness then
            nueva_p.erase(nueva_p.begin())                                (Elitismo)
            nueva_p.insert(*mejor_padre)
        p = nueva_p                                                         (Nueva generación)
        num_generaciones++
    w = p.end()->w
    return w

```

---

Para el pseudocódigo del **algoritmo genético generacional con cruce aritmético**, lo único que cambiaría con respecto al pseudocódigo anterior serían las líneas destacadas en azul por lo que hemos comentado al principio de este apartado. De esta forma, el pseudocódigo sería idéntico al anterior, pero con el siguiente cambio:

---

```
función AGG_Arit(entrenamiento)
    ...                                     (Idéntico a AGG_BLX)
    SELECCIONAR(2*TAMANIO_AGG)
    ARITMETICO(num_cruces_esperados, 2*TAMANIO_AGG)
    ...                                     (Idéntico a AGG_BLX)
```

---

En el algoritmo AGG\_Arit seleccionamos  $2 \cdot \text{TAMANIO\_AGG}$  padres para la población intermedia, pero tras el cruce trabajamos únicamente con los  $\text{TAMANIO\_AGG}$  primeros, respetando así el esquema generacional, en el cual la población intermedia tiene el mismo tamaño que la población original.

Tanto en AGG\_BLX como en AGG\_Arit, al reemplazar la nueva población nos aseguramos de que todos los cromosomas estén correctamente evaluados (y ordenados según el criterio aplicado). Finalmente, antes de pasar a la siguiente generación, aplicamos un comportamiento elitista: si hemos perdido la mejor solución que teníamos en la generación anterior, la introducimos en la nueva generación eliminando la peor solución de esta.

### 3.2.2. Algoritmos Genéticos Estacionarios

Al igual que en los algoritmos genéticos generacionales, para los algoritmos genéticos estacionarios también hemos implementado dos tipos, uno con cruce BLX (**AGE\_BLX**) y otro con cruce aritmético (**AGE\_Arit**). En este caso, cambia el esquema de reemplazamiento con respecto a los generacionales. De igual forma, como ahora la población intermedia está formada por únicamente dos cromosomas (que siempre cruzan),  $\text{TAMANIO\_AGE} = 2$ , también cambia el esquema de mutación, utilizando la segunda alternativa que comentábamos en el apartado 3.2.

Los dos cromosomas que componen la población intermedia compiten con los dos peores de la población anterior para entrar en la nueva población. Al final nos quedamos con los dos mejores de entre los cuatro.

Así, el pseudocódigo del **algoritmo genético estacionario con cruce BLX** sería el siguiente:

---

**función AGE\_BLX(entrenamiento)**

```
w(entrenamiento[0].num_caracts)    (w es el vector de pesos a calcular)
p = inicializarPoblacion(TAMANIO_AGG, w.size(), entrenamiento)
num_iteraciones = TAMANIO_AGG
num_generaciones = 1
num_genes_individuo = w.size()
num_cruces_esperados = 1.0 * (TAMANIO_AGE / 2)
while num_iteraciones < MAX_ITER do                                     (MAX_ITER = 15000)
    p_intermedia = Ø                                                    (No ordenada)
    nueva_p = Ø                                                         (Ordenada)
    SELECCIONAR(TAMANIO_AGE)
    BLX(num_cruces_esperados)
    MUTAR_2(TAMANIO_AGE, num_genes_individuo)
    for i:=0 to TAMANIO_AGE do                                         (Reemplazo)
        p_intermedia[i].fitness = fitness(
            tasaClasificacionLeaveOneOut(entrenamiento,p_intermedia[i].w),
            tasaReduccion(p_intermedia[i].w)
        )
        num_iteraciones++
        nueva_p.insert(p_intermedia[i])
    peor_cromosoma = p.begin()
    segundo_peor_cromosoma = p.begin() + 1
    mejor_descendiente_actual = nueva_p.end()
    segundo_mejor_descendiente_actual = nueva_p.end() - 1
    if segundo_mejor_descendiente_actual->fitness >
        segundo_peor_cromosoma ->fitness then
        p.erase(segundo_peor_cromosoma)    (Sobreviven los dos hijos)
        p.erase(peor_cromosoma)
        p.insert(*segundo_mejor_descendiente_actual)
        p.insert(*mejor_descendiente_actual)
    else if mejor_descendiente_actual->fitness >
        peor_cromosoma ->fitness then
        p.erase(peor_cromosoma)            (Sobrevive solo el mejor hijo)
        p.insert(*mejor_descendiente_actual)
    num_generaciones++                                                  (Nueva generación)
    w = p.end()->w
return w
```

---

Al igual que ocurría para los algoritmos genéticos generacionales, para el pseudocódigo del **algoritmo genético estacionario con cruce aritmético**, lo único que cambiaría con respecto al pseudocódigo anterior serían las líneas destacadas en azul. De esta forma, el pseudocódigo sería idéntico al anterior, pero con el siguiente cambio:

---

```
función AGE_Arit(entrenamiento)
    ...                                     (Idéntico a AGE_BLX)
    SELECCIONAR(2*TAMANIO_AGE)
    ARITMETICO(num_cruces_esperados, 2*TAMANIO_AGE)
    ...                                     (Idéntico a AGE_BLX)
```

---

### 3.2.3. Algoritmos Meméticos

Para implementar los algoritmos meméticos nos basamos en el algoritmo AGG\_BLX (el cual es el que mejores resultados proporciona comparado con AGG\_Arit, tal y como se puede observar en el apartado 5.3). En este caso, el tamaño de la población es de **TAMANIO\_AM = 50** cromosomas.

Tenemos que ir contando el número de generaciones mediante una variable *num\_generaciones* y, cada diez generaciones, es decir, si  $num\_generaciones \% 10 == 0$ , tenemos que aplicar un **algoritmo de búsqueda local de baja intensidad** (estructura idéntica al algoritmo de búsqueda local desarrollado en la primera práctica, solo que cambia el esquema de representación de las soluciones y, lo más destacable, cambia el criterio de parada, el cual consiste en parar cuando se hayan evaluado  $2*n$  vecinos distintos en la ejecución, siendo  $n$  el número de genes del cromosoma).

---

```
INDICES(n) :
    for i:=0 to n - 1 do
        índices.push_back(i)
    shuffle(índices)
```

---

---

```

función busquedaLocalBajaIntensidad(entrenamiento, c)
    num_iteraciones = 0
    indices = INDICES(n)                                (n = c.w.size())
    mejor_fitness = c.fitness
    while num_vecinos < n * COEF_MAX_VECINOS_BAJA_INTENSIDAD do
                                                (COEF_MAX_VECINOS_BAJA_INTENSIDAD = 2)
        comp_mutada = index[num_iteraciones % n]
        c_mutado = c
        c_mutado[comp_mutada] += normal(0, sqrt(0.3))
        if c_mutado[comp_mutada] > 1 then
            c_mutado[comp_mutada] = 1
        else if c_mutado[comp_mutada] < 0 then
            c_mutado[comp_mutada] = 0
        c_mutado.fitness = fitness(
            tasaClasificacionLeaveOneOut(entrenamiento, c_mutado.w),
            tasaReduccion(c_mutado.w)
        )
        num_iteraciones++
        if c_mutado.fitness > mejor_fitness then
            c = c_mutado
            mejor_fitness = c_mutado.fitness
        if num_iteraciones % n == 0 then
            shuffle(indices)
    return num_iteraciones

```

---

Hemos implementado tres versiones diferentes de **algoritmos meméticos** y, como están basados en el algoritmo AGG\_BLX, los pseudocódigos serán casi idénticos, cambiando simplemente *TAMANIO\_AGG* por *TAMANIO\_AM* (aunque en nuestro caso, coinciden) y añadiendo la búsqueda local de baja intensidad al final, que es lo que vamos a especificar en los pseudocódigos:

- 1) **AM-(10,1.0) / AM\_All**: Cada 10 generaciones, aplicamos la búsqueda local de baja intensidad sobre todos los cromosomas de la población.



---

**función AM\_All(entrenamiento)**

```
... (Idéntico a AGG_BLX)
while ...
    ...
    p = nueva_p
    if num_generaciones % 10 == 0 then
        nueva_p.clear()
        for c in p
            num_iteraciones += busquedaLocalBajaIntensidad(entrenamiento, c)
            nueva_p.insert(c)
        p = nueva_p
    num_generaciones++ (Nueva generación)
w = p.end()->w
return w
```

---

- 2) **AM-(10,0.1) / AM\_Rand**: Cada 10 generaciones, aplicamos la búsqueda local de baja intensidad sobre un subconjunto de la población seleccionado aleatoriamente con probabilidad  $P_{LS} = 0.1$  para cada cromosoma.

---

**función AM\_Rand(entrenamiento)**

```
... (Idéntico a AGG_BLX)
while ...
    ...
    p = nueva_p
    if num_generaciones % 10 == 0 then
        nueva_p.clear()
        for i:=0 to TAMANIO_AM *  $P_{LS}$  do
            num_random = uniformInt(0, TAMANIO_AM - 1)
            c = advance(p.begin(), random)
            num_iteraciones += busquedaLocalBajaIntensidad(entrenamiento, c)
            p.erase(p.begin() + random)
            p.insert(c)
        num_generaciones++ (Nueva generación)
w = p.end()->w
return w
```

---

- 3) **AM-(10,0.1mej) / AM\_Best**: Cada 10 generaciones, aplicamos la búsqueda local de baja intensidad sobre los  $0.1 \cdot \text{TAMANIO\_AM}$  mejores cromosomas de la población actual.

---

```
función AM_Best(entrenamiento)
    ...
while ...
    ...
    p = nueva_p
    if num_generaciones % 10 == 0 then
        nueva_p.clear()
        for i:=0 to TAMANIO_AM * PLS do
            c = prev(p.end(), i+1)
            num_iteraciones += busquedaLocalBajaIntensidad(entrenamiento, c)
            p.erase(p.end() - i - 1)
            p.insert(c)
        num_generaciones++
    w = p.end()->w
return w
```

---

### 3.3. Algoritmos Implementados (Práctica 3)

#### 3.3.1. Algoritmo Búsqueda Multiarranque Básica

El **algoritmo de Búsqueda Multiarranque Básica (BMB)** consiste en generar un determinado número de soluciones aleatorias (en nuestro caso, 15 soluciones aleatorias) y aplicarle a cada una de ellas el algoritmo de Búsqueda Local para optimizarlas. Se devolverá la mejor solución encontrada en todo el proceso.

Para la Búsqueda Local usaremos la función ***busquedaLocalKcomp* con  $k = 1$**  (para  $k = 1$ , dicho algoritmo es exactamente el algoritmo *busquedaLocal* desarrollado en la práctica 1). Se puede observar el pseudocódigo de esta función en el apartado 3.3.4 (Algoritmo Búsqueda de Vecindario Variable).

---

```

función BMB(entrenamiento)
    mejor_solucion = Ø
    for i:=0 to ITER_BMB do                                     (ITER_BMB = 15)
                                                                    (n es el número de características)
        sol = inicializarSolucion(n, entrenamiento)
        busquedaLocalKcomp(entrenamiento, sol, 1)
        if sol.fitness > mejor_solucion.fitness then
            mejor_solucion = sol
    return mejor_solucion.w

```

---

### 3.3.2. Algoritmo Enfriamiento Simulado

El **algoritmo de Enfriamiento Simulado (ES)** es un algoritmo de búsqueda por trayectorias que persigue subsanar uno de los problemas más destacados de la búsqueda local: la caída en óptimos locales. Para ello, este algoritmo **permite aceptar soluciones peores que la mejor solución actual**.

Para llevar a cabo este procedimiento nos basamos en un modelo termodinámico en el que hay una “temperatura” figurada que va disminuyendo (se va enfriando) de forma que, a mayor temperatura, mayor probabilidad de aceptación de peores soluciones. Con el aumento de iteraciones disminuye la probabilidad de aceptar soluciones peores que el actual. Así, el algoritmo acepta con más frecuencia soluciones mucho peores al principio de la ejecución pero apenas acepta al final.

Para controlar la probabilidad de aceptación de soluciones utilizamos el modelo de Metrópolis. En este modelo se genera una perturbación (mutación) aleatoria en el sistema y se calculan los cambios de energía resultantes: si hay una caída energética (mejora la solución), entonces el cambio se acepta automáticamente y si se produce un incremento energético (empeora la solución), entonces el cambio será aceptado con una probabilidad  $p = e^{-\Delta f / T}$ , donde  $\Delta f$  es la diferencia de la función a maximizar sobre las soluciones y  $T$  es la temperatura actual del sistema.

La mutación de una solución se realiza siguiendo el esquema de **mutación normal** y la solución inicial se construye de forma aleatoria. Un último detalle a tener en cuenta es que es posible que una solución y su mutación tengan el mismo valor de la función objetivo. En este caso, el modelo de Metrópolis aceptaría siempre la mutación, por ello nosotros introducimos una pequeña perturbación en la diferencia para evitar que sea exactamente 0 y que la mutación pueda ser rechazada.

---

**función ES(entrenamiento)**

```
num_iteraciones = 0

                                (n es el número de características)

sol = inicializarSolucion(n, entrenamiento)
num_iteraciones++
mejor_solucion = sol

                                (MU = 0.3, PHI = 0.2)

 $T_0 = (MU * mejor\_solucion.fitness) / -\log(PHI)$ 
 $T = T_0$ 
 $T_f = 0.0001$ 
while  $T_f \geq T_0$  do                                (Debemos asegurarnos de que  $T_0 > T_f$ )
     $T_f = T_f / 100$ 
    MAX_VECINOS = 10 * mejor_solucion.w.size()
    MAX_EXITOS = 0.1 * MAX_VECINOS

                                (MAX_ITER_ES = 15000)
     $M = MAX\_ITER\_ES / MAX\_VECINOS$     (Número de enfriamientos a realizar)
     $\beta = (T_0 - T_f) / (M * T_0 * T_f)$ 
    num_exitos = 1
    num_vecinos = 0
    while (num_iteraciones < MAX_ITER_ES and num_exitos != 0) do
        num_vecinos = 0
        num_exitos = 0
        while (num_iteraciones < MAX_ITER_ES and num_vecinos < MAX_VECINOS
            and num_exitos < MAX_EXITOS) do
            comp = uniformInt(0, n-1)
            sol_mutada = sol
            sol_mutada.w[comp] += normal(0, sqrt(0.3))
            if sol_mutada.w[comp] > 1 then
                sol_mutada.w[comp] = 1
            else if sol_mutada.w[comp] < 0 then
                sol_mutada.w[comp] = 0
            sol_mutada.fitness = fitness(
                tasaClasificacionLeaveOneOut(entrenamiento, sol_mutada.w),
                tasaReduccion(sol_mutada.w)
            )
            num_iteraciones++
            num_vecinos++
             $\Delta f = sol.fitness - sol\_mutada.fitness$ 
```

```

    if  $\Delta f == 0$  then
         $\Delta f = 0.001$ 
    if ( $\Delta f < 0$  or uniformReal(0, 1)  $\leq e^{-\Delta f / T}$ ) then
        num_exitos++
        sol = sol_mutada
        if sol.fitness > mejor_solucion.fitness then
            mejor_solucion = sol
         $T = T / (1 + \beta * T)$  (Esquema de enfriamiento de Cauchy modificado)
    return mejor_solucion.w

```

---

También hemos desarrollado una función en la que le aplicamos el **algoritmo de Enfriamiento Simulado a una solución (ES\_Solucion)**. Dicha función se utilizará en el algoritmo ILS-ES.

---

```

función ES_Solucion(entrenamiento, sol)
    num_iteraciones = 0
    mejor_solucion = sol
    num_iteraciones++
    ... (Idéntico a ES, la única diferencia es que MAX_ITER_ES = 15000
        se sustituye por MAX_ITER_ES_SOL = 1000)
    sol = mejor_solucion
    return sol

```

---

### 3.3.3. Algoritmo Búsqueda Local Iterativa

El **algoritmo de Búsqueda Local Iterativa (ILS)** consiste en generar una solución inicial aleatoria y aplicar el algoritmo de Búsqueda Local sobre ella. Una vez obtenida la solución optimizada, se estudia si es mejor que la mejor solución encontrada hasta el momento y se realizará una **mutación fuerte**, tal y como se describe en el apartado 2.6 (Operadores Comunes de la Práctica 3) sobre la mejor de estas dos, volviendo a aplicar el algoritmo de Búsqueda Local sobre esta solución mutada. Este proceso se repite un determinado número de veces (en nuestro caso, 15 veces), devolviéndose la mejor solución encontrada en todo el proceso.

Al igual que hacíamos en el algoritmo BMB, para la Búsqueda Local usaremos la función **busquedaLocalKcomp con  $k = 1$**  (para  $k = 1$ , dicho algoritmo es exactamente el algoritmo *busquedaLocal* desarrollado en la práctica 1). El pseudocódigo de esta función se puede encontrar en el apartado 3.3.4 (Algoritmo Búsqueda de Vecindario Variable).

---

```
función ILS(entrenamiento)
    (n es el número de características)
    sol_inicial = inicializarSolucion(n, entrenamiento)
    busquedaLocalKComp(entrenamiento, sol_inicial, 1)
    mejor_solucion = sol_inicial
    for i:=1 to ITER_ILS do (ITER_ILS = 15)
        sol_mutada = mejor_solucion
        mutacionFuerte(sol_mutada.w)
        sol_mutada.fitness = fitness(
            tasaClasificacionLeaveOneOut(entrenamiento, sol_mutada.w),
            tasaReduccion(sol_mutada.w)
        )
        busquedaLocalKComp(entrenamiento, sol_mutada, 1)
        if sol_mutada.fitness > mejor_solucion.fitness then
            mejor_solucion = sol_mutada
    return mejor_solucion.w
```

---

Se ha desarrollado también el **algoritmo de Búsqueda Local Iterativa usando el algoritmo de Enfriamiento Simulado (ILS-ES)** en vez de la Búsqueda Local. Para ello, tal y como se comentó en el apartado 3.3.2 (Algoritmo Enfriamiento Simulado), se utiliza la función **ES\_Solucion** cuyo pseudocódigo puede observarse en dicho apartado.

---

```

función ILS_ES(entrenamiento)
                                (n es el número de características)
    sol_inicial = inicializarSolucion(n, entrenamiento)
    ES_Solucion(entrenamiento, sol_inicial)
    mejor_solucion = sol_inicial
    for i:=1 to ITER_ILS do                                     (ITER_ILS = 15)
        sol_mutada = mejor_solucion
        mutacionFuerte(sol_mutada.w)
        sol_mutada.fitness = fitness(
            tasaClasificacionLeaveOneOut(entrenamiento, sol_mutada.w),
            tasaReduccion(sol_mutada.w)
        )
        ES_Solucion(entrenamiento, sol_inicial)
        if sol_mutada.fitness > mejor_solucion.fitness then
            mejor_solucion = sol_mutada
    return mejor_solucion.w

```

---

### 3.3.4. Algoritmo Búsqueda de Vecindario Variable

El **algoritmo de Búsqueda de Vecindario Variable (VNS)** consiste en generar una solución inicial aleatoria y aplicar el algoritmo de Búsqueda Local sobre ella. A diferencia del algoritmo ILS, cada vez que se aplique la Búsqueda Local, esta se aplicará con un vecindario distinto.

El vecindario determina el número de atributos que se cambia simultáneamente en la Búsqueda Local aplicada al vector de pesos de la solución. Inicialmente  $k = 1$  (es decir, solo se cambia un atributo, como se hace en el algoritmo *busquedaLocal* de la práctica 1), luego pasa a  $k = 2$  (en este caso no se aplica en cada paso la Búsqueda Local a un atributo, sino a dos, de forma aleatoria sin repetir), luego  $k = 3$ , y así sucesivamente. Esta funcionalidad la implementamos en la función ***busquedaLocalKComp***, cuyo pseudocódigo es el siguiente:

---

```

función busquedaLocalKComp(entrenamiento, sol, k)

    num_iteraciones = 0
    num_vecinos = 0
    hay_mejora = false
    for i:=0 to n-1 do                                     (n es el número de características)
        indices.push_back(i)
    shuffle(indices)
    mejor_fitness = fitness(
        tasaClasificacionLeaveOneOut(entrenamiento, sol.w),
        tasaReduccion(sol.w)
    )
    (MAX_ITER_BL_K_COMP = 1000, COEF_MAX_VECINOS = 20)
while (num_iteraciones < MAX_ITER_BL_K_COMP
        and num_vecinos < n*COEF_MAX_VECINOS) do
    sol_mutada = sol
    for i:=0 to k do
        comp_mutada = index[num_iteraciones % n]
        sol_mutada.w[comp_mutada] += normal(0, sqrt(0.3))
        if sol_mutada.w[comp_mutada] > 1 then
            sol_muada.w [comp_mutada] = 1
        else if sol_mutada.w[comp_mutada] < 0 then
            sol_mutada.w[comp_mutada] = 0
        num_iteraciones++
    sol_mutada.fitness = fitness(
        tasaClasificacionLeaveOneOut(entrenamiento, sol_mutada.w),
        tasaReduccion(sol_mutada.w)
    )

    if sol_mutada.fitness > mejor_fitness then
        num_vecinos = 0
        sol = sol_mutada
        mejor_fitness = sol_mutada.fitness
        hay_mejora = true
    else
        num_vecinos++
    if (num_iteraciones % n == 0 or hay_mejora) then
        shuffle(indices)
        hay_mejora = false
return sol

```

---



De esta forma, en el algoritmo VNS si tras aplicar la Búsqueda Local se produce una mejora,  $k = 1$ , y si no se produce mejora,  $k$  aumenta su valor en 1. Cuando se haya aplicado el último vecindario, volverá al primero, es decir, si  $k$  supera a  $K\_MAX$  (en nuestro caso,  $K\_MAX = 3$ ), entonces  $k = 1$ .

Una vez obtenida la solución optimizada, se estudia si es mejor que la mejor solución encontrada hasta el momento y se realizará una **mutación fuerte**, tal y como se describe en el apartado 2.6 (Operadores Comunes de la Práctica 3) sobre la mejor de estas dos, volviendo a aplicar el algoritmo de búsqueda local sobre esta solución mutada. Este proceso se repite un determinado número de veces (en nuestro caso serán 15 veces), devolviéndose la mejor solución encontrada en todo el proceso.

---

**función VNS(entrenamiento)**

```
k = 1

                                (n es el número de características)
sol_inicial = inicializarSolucion(n, entrenamiento)
busquedaLocalKComp(entrenamiento, sol_inicial, 1)
mejor_solucion = sol_inicial
for i:=1 to ITER_VNS do                                (ITER_VNS = 15)
    sol_mutada = mejor_solucion
    mutacionFuerte(sol_mutada.w)
    sol_mutada.fitness = fitness(
        tasaClasificacionLeaveOneOut(entrenamiento, sol_mutada.w),
        tasaReduccion(sol_mutada.w)
    )
    busquedaLocalKComp(entrenamiento, sol_mutada, 1)
    if sol_mutada.fitness > mejor_solucion.fitness then
        mejor_solucion = sol_mutada
        k = 1
    else
        k = (k % K_MAX) + 1                                (K_MAX = 3)
return mejor_solucion.w
```

---

## 4. Procedimiento Considerado para Desarrollar la Práctica

Todo el código de la práctica se ha desarrollado en **C++** siguiendo el estándar 2011. Utilizamos diferentes librerías como *iostream*, *vector*, *cmath*, *random*... Además, hacemos uso también del archivo ***random.hpp*** proporcionado por el profesor para trabajar de forma más cómoda con números aleatorios. **No se ha hecho uso de ningún framework de metaheurísticas.**

Respecto a la **estructura de directorios y archivos**, Los ficheros de datos se encuentran en la carpeta *BIN/DATA* y los ejecutables en la carpeta *BIN*. Por otro lado, el código fuente se encuentra en la carpeta *FUENTES*, en la cual encontramos el archivo *CMakeLists.txt* (este contiene una serie de instrucciones que al lanzar el comando *cmake* para ejecutarlo, nos generará un makefile que nos permitirá compilar el proyecto mediante la orden *make*), la carpeta *INCLUDE* con los archivos de cabecera *.h* y la carpeta *SRC* con los archivos de código *.cpp*. Respecto a los diferentes archivos de código:

- ***util.h/util.cpp*** → Contienen funciones auxiliares que se utilizan en las diferentes prácticas, tales como aquellas destinadas a la lectura y normalización de datos de los archivos, las funciones de la distancia euclídea, la función de evaluación, etc.
- ***p1.h/p1.cpp*** → Contienen los algoritmos implementados en la primera práctica (clasificador1NN, Greedy RELIEF y Búsqueda Local), así como una función para cada algoritmo con el fin de ejecutar dicho algoritmo con los diferentes conjuntos de datos y mostrar los resultados obtenidos.
- ***p2.h/p2.cpp*** → Contienen los algoritmos implementados en la segunda práctica (AGG\_BLX, AGG\_Arit, AGE\_BLX, AGE\_Arit, AM\_All, AM\_Rand y AM\_Best), así como una función para cada algoritmo con el fin de ejecutar dicho algoritmo con los diferentes conjuntos de datos y mostrar los resultados obtenidos.
- ***p3.h (INCLUDE)/p3.cpp (SRC)*** → Contienen los algoritmos implementados en la tercera práctica (BMB, ES, ILS, ILS\_ES y VNS) así como una función para cada algoritmo con el fin de ejecutar dicho algoritmo con los diferentes conjuntos de datos y mostrar los resultados obtenidos.
- ***random.hpp*** → Archivo proporcionado por el profesor para trabajar de forma más cómoda con números aleatorios.
- ***main.cpp*** → Archivo que contiene la función principal para ejecutar el programa.

Para **compilar y ejecutar el proyecto** hay que hacer lo siguiente:

- 1) Situarnos en la carpeta **software/FUENTES**.
- 2) Ejecutar el comando **"cmake ."**, generándose así el makefile correspondiente.
- 3) Ejecutar el comando **"make"** para compilar el proyecto y obtener el ejecutable **practica3\_MH** en la carpeta **software/BIN**.
- 4) Situarnos en la carpeta **software/BIN**.
- 5) Como ya hemos dicho, trabajamos con números aleatorios, luego, necesitamos especificar una semilla mediante la línea de comandos a la hora de ejecutar el programa. Por ello, el comando para ejecutar el programa es **./practica3\_MH {semilla}**.

## **5. Experimentos y Análisis de Resultados**

Ejecutamos los algoritmos sobre cada uno de los tres conjuntos de datos siguientes:

- **Diabetes:** Contiene atributos calculados a partir de mediciones de pacientes. La tarea consiste en determinar si desarrollarán diabetes en los próximos 5 años. Consta de 768 ejemplos, 8 atributos numéricos y la clase (puede ser de dos tipos, negativa o positiva).
- **Ozone:** Base de datos para la detección del nivel de ozono según las mediciones realizadas a lo largo del tiempo. Consta de 320 ejemplos, 72 atributos numéricos y la clase (puede ser de dos tipos, día normal o día con una alta concentración en ozono).
- **Spectf-heart:** Contiene atributos calculados a partir de imágenes médicas de tomografía computarizada (SPECT) del corazón de pacientes humanos. La tarea consiste en determinar si la fisiología del corazón analizado es correcta o no. Costa de 267 ejemplos, 44 atributos entero y la clase (puede ser de dos tipos, paciente sano o paciente con patología cardíaca).

### **5.1. Resultados Obtenidos**

A continuación, se muestran las tablas de los resultados obtenidos para cada uno de los algoritmos aplicados sobre los diferentes conjuntos de datos con la **semilla 49559494**. En la tabla se observan los resultados de 5 ejecuciones diferentes para cada conjunto de datos de acuerdo a la técnica de validación cruzada *5-fold cross validation*. Se muestran los valores de la tasa de clasificación (**%\_clas**), la tasa de reducción (**%red**), la función objetivo (**Fit.**) y el tiempo de ejecución en milisegundos (**T**). Finalmente, también se muestre una tabla global con los resultados medios obtenidos de cada algoritmo para los diferentes conjuntos de datos.

Tabla 5.1: Resultados obtenidos por el algoritmo 1-NN en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	%red	Fit.	T	% clas	%red	Fit.	T	% clas	%red	Fit.	T
Partición 1	66.88	0.00	53.51	0.94	78.12	0.00	62.50	1.27	77.14	0.00	61.71	0.99
Partición 2	66.88	0.00	53.51	1.05	85.94	0.00	68.75	1.27	88.57	0.00	70.86	1.36
Partición 3	68.83	0.00	55.06	1.13	79.69	0.00	63.75	1.28	91.43	0.00	73.14	1.04
Partición 4	69.48	0.00	55.58	1.35	76.56	0.00	61.25	1.23	85.71	0.00	68.57	0.94
Partición 5	69.74	0.00	55.79	0.96	81.25	0.00	65.00	1.23	84.06	0.00	67.25	0.97
Media	68,36	0,00	54,69	1,09	80,31	0,00	64,25	1,26	85,38	0,00	68,31	1,06

Tabla 5.2: Resultados obtenidos por el algoritmo Greedy RELIEF en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	%red	Fit.	T	% clas	%red	Fit.	T	% clas	%red	Fit.	T
Partición 1	63.64	0.00	50.91	9.61	73.44	6.94	60.14	5.28	84.29	22.73	71.97	4.32
Partición 2	63.64	12.50	53.41	9.72	76.56	8.33	62.92	5.15	87.14	25.00	74.71	3.94
Partición 3	70.78	0.00	56.62	10.05	78.12	8.33	64.17	5.17	85.71	25.00	73.57	3.96
Partición 4	72.73	12.50	60.68	9.61	68.75	2.78	55.56	5.38	85.71	22.73	73.12	3.85
Partición 5	67.76	0.00	54.21	9.91	81.25	8.33	66.67	5.45	79.71	25.00	68.77	3.91
Media	67,71	5,00	55,17	9,78	75,62	6,94	61,89	5,29	84,51	24,09	72,43	4,00

Tabla 5.3: Resultados obtenidos por el algoritmo BL en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	%red	Fit.	T	% clas	%red	Fit.	T	% clas	%red	Fit.	T
Partición 1	62.99	50.00	60.39	855.54	76.56	6.94	62.64	7344.61	72.86	29.55	64.19	2972.07
Partición 2	35.06	100.00	48.05	1861.66	79.69	9.72	65.69	7184.96	85.71	13.64	71.30	3162.03
Partición 3	61.04	25.00	53.83	840.30	81.25	8.33	66.67	7469.99	92.86	25.00	79.29	3057.82
Partición 4	68.83	37.50	62.56	780.20	78.12	20.83	66.67	6566.30	75.71	13.64	63.30	3205.58
Partición 5	65.13	12.50	54.61	804.04	87.50	16.67	73.33	6999.03	78.26	22.73	67.15	3043.43
Media	58,61	45,00	55,89	1028,35	80,62	12,50	67,00	7112,98	81,08	20,91	69,05	3088,19

Tabla 5.4: Resultados obtenidos por el algoritmo AGG-BLX en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	%red	Fit.	T	% clas	%red	Fit.	T	% clas	%red	Fit.	T
Partición 1	64.94	100.00	71.95	67624.10	75.00	47.22	69.44	58344.18	81.43	31.82	71.51	47225.60
Partición 2	72.08	62.50	70.16	68234.58	84.38	40.28	75.56	60198.34	88.57	47.73	80.40	44495.45
Partición 3	66.88	62.50	66.01	66636.67	79.69	51.39	74.03	60046.18	92.86	40.91	82.47	46570.37
Partición 4	68.83	75.00	70.06	65082.06	71.88	31.94	63.89	62988.75	80.00	45.45	73.09	45035.53
Partición 5	69.08	75.00	70.26	65562.97	75.00	27.78	65.56	64643.33	78.26	56.82	73.97	45381.13
Media	68,36	75,00	69,69	66628,08	77,19	39,72	69,69	61244,16	84,22	44,55	76,29	45741,61

Tabla 5.5: Resultados obtenidos por el algoritmo AGG-CA en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	%red	Fit.	T	% clas	%red	Fit.	T	% clas	%red	Fit.	T
Partición 1	67.53	62.50	66.53	66861.70	79.69	25.00	68.75	70162.63	85.71	34.09	75.39	52413.85
Partición 2	72.73	50.00	68.18	66893.50	84.38	29.17	73.33	67928.01	85.71	29.55	74.48	50162.91
Partición 3	66.23	62.50	65.49	69808.61	78.12	12.50	65.00	77051.15	90.00	36.36	79.27	48425.92
Partición 4	66.88	62.50	66.01	66769.95	68.75	23.61	59.72	68373.41	81.43	31.82	71.51	49341.40
Partición 5	69.08	75.00	70.26	67815.73	81.25	30.56	71.11	64904.95	84.06	45.45	76.34	48092.60
Media	68,49	62,50	67,29	67629,90	78,44	24,17	67,58	69684,03	85,38	35,45	75,40	49687,33

Tabla 5.6: Resultados obtenidos por el algoritmo AGE-BLX en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	%red	Fit.	T	% clas	%red	Fit.	T	% clas	%red	Fit.	T
Partición 1	69.48	62.50	68.08	66436.06	78.12	34.72	69.44	61050.48	80.00	36.36	71.27	46170.54
Partición 2	66.23	62.50	65.49	64635.30	82.81	27.78	71.81	64331.82	84.29	63.64	80.16	42729.07
Partición 3	66.88	62.50	66.01	66446.11	78.12	22.22	66.94	65244.54	88.57	43.18	79.49	44471.10
Partición 4	62.34	75.00	64.87	66009.49	71.88	38.89	65.28	58826.31	87.14	52.27	80.17	43782.17
Partición 5	69.08	62.50	67.76	65933.24	78.12	26.39	67.78	63077.72	82.61	47.73	75.63	44253.21
Media	66,80	65,00	66,44	65892,04	77,81	30,00	68,25	62506,18	84,52	48,64	77,34	44281,22

Tabla 5.7: Resultados obtenidos por el algoritmo AGE-CA en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	%red	Fit.	T	% clas	%red	Fit.	T	% clas	%red	Fit.	T
Partición 1	70.13	62.50	68.60	66690.44	76.56	34.72	68.19	64303.52	81.43	27.27	70.60	50030.63
Partición 2	67.53	62.50	66.53	66310.45	82.81	36.11	73.47	63667.43	84.29	36.36	74.70	45608.22
Partición 3	67.53	62.50	66.53	64733.53	75.00	41.67	68.33	61091.69	88.57	29.55	76.77	47161.53
Partición 4	71.43	50.00	67.14	66335.88	75.00	34.72	66.94	61681.80	87.14	38.64	77.44	46185.23
Partición 5	63.82	62.50	63.55	65304.78	78.12	27.78	68.06	69011.29	85.51	34.09	75.22	48545.27
Media	68,09	60,00	66,47	65875,02	77,50	35,00	69,00	63951,15	85,39	33,18	74,95	47506,18

Tabla 5.8: Resultados obtenidos por el algoritmo AM-(10,1.0) en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	%red	Fit.	T	% clas	%red	Fit.	T	% clas	%red	Fit.	T
Partición 1	69.48	62.50	68.08	67287.05	78.12	38.89	70.28	64861.95	80.00	59.09	75.82	57566.96
Partición 2	66.88	62.50	66.01	67443.97	78.12	37.50	70.00	66971.14	77.14	70.45	75.81	56475.04
Partición 3	66.88	62.50	66.01	66287.79	81.25	43.06	73.61	63619.21	88.57	63.64	83.58	56626.21
Partición 4	62.34	75.00	64.87	67154.74	76.56	31.94	67.64	67504.84	84.29	70.45	81.52	58432.11
Partición 5	69.08	75.00	70.26	67459.99	76.56	44.44	70.14	64597.63	85.51	56.82	79.77	58570.08
Media	66,93	67,50	67,05	67126,71	78,12	39,17	70,33	65510,95	83,10	64,09	79,30	57534,08

Tabla 5.9: Resultados obtenidos por el algoritmo AM-(10,0.1) en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	%red	Fit.	T	% clas	%red	Fit.	T	% clas	%red	Fit.	T
Partición 1	64.94	100.00	71.95	62059.87	84.38	51.39	77.78	60706.12	85.71	43.18	77.21	45605.43
Partición 2	65.58	62.50	64.97	65204.25	82.81	43.06	74.86	60536.43	85.71	59.09	80.39	43395.79
Partición 3	67.53	62.50	66.53	65394.06	82.81	54.17	77.08	57566.18	90.00	63.64	84.73	42908.73
Partición 4	67.53	62.50	66.53	66909.06	78.12	54.17	73.33	58128.68	80.00	63.64	76.73	42860.50
Partición 5	69.08	75.00	70.26	71135.69	84.38	56.94	78.89	57600.70	88.41	68.18	84.36	42946.13
Media	66,93	72,50	68,05	66140,59	82,50	51,94	76,39	58907,62	85,97	59,55	80,68	43543,32

Tabla 5.10: Resultados obtenidos por el algoritmo AM-(10,0.1mej) en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	%red	Fit.	T	% clas	%red	Fit.	T	% clas	%red	Fit.	T
Partición 1	69.48	62.50	68.08	65851.35	79.69	38.89	71.53	61379.81	88.57	43.18	79.49	45780.58
Partición 2	66.23	62.50	65.49	65127.29	81.25	44.44	73.89	58269.31	92.86	50.00	84.29	44339.34
Partición 3	66.88	62.50	66.01	66503.40	82.81	52.78	76.81	57545.23	88.57	47.73	80.40	44966.22
Partición 4	69.48	62.50	68.08	66185.94	75.00	48.61	69.72	58730.42	81.43	70.45	79.23	43201.84
Partición 5	71.05	62.50	69.34	66631.62	81.25	56.94	76.39	57850.38	81.16	61.36	77.20	47762.71
Media	68,63	62,50	67,40	66059,92	80,00	48,33	73,67	58755,03	86,52	54,55	80,12	45210,14

Tabla 5.11: Resultados obtenidos por el algoritmo BMB en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	%red	Fit.	T	% clas	%red	Fit.	T	% clas	%red	Fit.	T
Partición 1	64,94	100,00	71,95	30577,31	79,69	43,06	72,36	93241,08	88,57	52,27	81,31	66824,31
Partición 2	65,58	62,50	64,97	27805,67	76,56	50,00	71,25	90784,80	87,14	61,36	81,99	66855,43
Partición 3	66,88	62,50	66,01	28877,57	78,12	43,06	71,11	90700,25	97,14	52,27	88,17	68127,59
Partición 4	62,34	75,00	64,87	31757,09	75,00	47,22	69,44	92464,67	81,43	54,55	76,05	67061,59
Partición 5	69,08	75,00	70,26	30344,55	78,12	37,50	70,00	91785,54	76,81	59,09	73,27	67495,25
Media	65,76	75,00	67,61	29872,44	77,50	44,17	70,83	91795,27	86,22	55,91	80,16	67272,84

Tabla 5.12: Resultados obtenidos por el algoritmo ES en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	%red	Fit.	T	% clas	%red	Fit.	T	% clas	%red	Fit.	T
Partición 1	66,23	75,00	67,99	16781,38	78,12	43,06	71,11	52687,38	80,00	54,55	74,91	31511,28
Partición 2	61,69	62,50	61,85	15111,93	76,56	48,61	70,97	51557,32	84,29	70,45	81,52	22332,88
Partición 3	65,58	62,50	64,97	15740,51	78,12	52,78	73,06	50526,54	92,86	59,09	86,10	25277,67
Partición 4	67,53	62,50	66,53	12767,81	73,44	47,22	68,19	40521,60	90,00	54,55	82,91	20782,98
Partición 5	65,79	62,50	65,13	17073,45	75,00	47,22	69,44	35455,58	82,61	61,36	78,36	30427,98
Media	65,37	65,00	65,29	15495,02	76,25	47,78	70,56	46149,68	85,95	60,00	80,76	26066,56

Tabla 5.13: Resultados obtenidos por el algoritmo ILS en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	%red	Fit.	T	% clas	%red	Fit.	T	% clas	%red	Fit.	T
Partición 1	66,23	75,00	67,99	14991,92	70,31	52,78	66,81	83919,35	82,86	70,45	80,38	59848,51
Partición 2	73,38	50,00	68,70	15447,03	79,69	51,39	74,03	83997,63	84,29	68,18	81,06	60096,26
Partición 3	66,88	62,50	66,01	15097,22	79,69	54,17	74,58	85525,74	92,86	52,27	84,74	63723,22
Partición 4	68,83	75,00	70,06	13879,96	76,56	62,50	73,75	82626,63	81,43	59,09	76,96	63866,82
Partición 5	73,03	62,50	70,92	14766,96	85,94	55,56	79,86	87198,05	86,96	68,18	83,20	62162,71
Media	69,67	65,00	68,74	14836,62	78,44	55,28	73,81	84653,48	85,68	63,64	81,27	61939,50

Tabla 5.14: Resultados obtenidos por el algoritmo ILS-ES en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	%red	Fit.	T	% clas	%red	Fit.	T	% clas	%red	Fit.	T
Partición 1	70,13	62,50	68,60	31149,90	76,56	63,89	74,03	84685,09	81,43	65,91	78,32	60954,62
Partición 2	67,53	62,50	66,53	31929,96	84,38	52,78	78,06	84705,79	75,71	61,36	72,84	60605,82
Partición 3	67,53	62,50	66,53	30966,49	78,12	58,33	74,17	85428,71	91,43	68,18	86,78	62667,75
Partición 4	62,34	75,00	64,87	28262,75	68,75	50,00	65,00	86324,65	90,00	77,27	87,45	60238,29
Partición 5	69,08	75,00	70,26	28101,57	79,69	56,94	75,14	84887,15	84,06	65,91	80,43	62643,14
Media	67,32	67,50	67,36	30082,13	77,50	56,39	73,28	85206,28	84,53	67,73	81,17	61421,92

Tabla 5.15: Resultados obtenidos por el algoritmo VNS en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	%red	Fit.	T	% clas	%red	Fit.	T	% clas	%red	Fit.	T
Partición 1	64,94	100,00	71,95	16882,44	79,69	61,11	75,97	69411,24	81,43	70,45	79,23	49581,07
Partición 2	66,23	62,50	65,49	14824,28	78,12	52,78	73,06	68676,63	85,71	75,00	83,57	43993,58
Partición 3	66,88	62,50	66,01	16470,97	81,25	47,22	74,44	78529,76	90,00	75,00	87,00	42556,02
Partición 4	67,53	62,50	66,53	17927,65	73,44	48,61	68,47	71420,45	80,00	65,91	77,18	45118,17
Partición 5	69,08	75,00	70,26	15705,82	82,81	47,22	75,69	73302,91	81,16	70,45	79,02	50079,26
Media	66,93	72,50	68,05	16362,23	79,06	51,39	73,53	72268,20	83,66	71,36	81,20	46265,62



Tabla 5.16: Resultados globales en el problema del APC

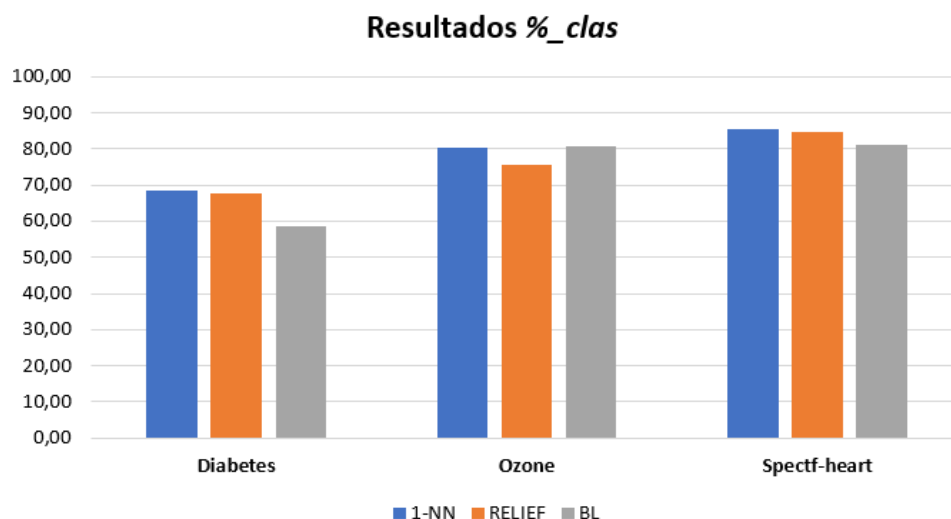
	Diabetes				Ozone				Spectf-heart			
	% clas	%red	Fit.	T	% clas	%red	Fit.	T	% clas	%red	Fit.	T
1-NN	68,36	0,00	54,69	1,09	80,31	0,00	64,25	1,26	85,38	0,00	68,31	1,06
RELIEF	67,71	5,00	55,17	9,78	75,62	6,94	61,89	5,29	84,51	24,09	72,43	4,00
BL	58,61	45,00	55,89	1028,35	80,62	12,50	67,00	7112,98	81,08	20,91	69,05	3088,19
AGG-BLX	68,36	75,00	69,69	66628,08	77,19	39,72	69,69	61244,16	84,22	44,55	76,29	45741,61
AGG-CA	68,49	62,50	67,29	67629,90	78,44	24,17	67,58	69684,03	85,38	35,45	75,40	49687,33
AGE-BLX	66,80	65,00	66,44	65892,04	77,81	30,00	68,25	62506,18	84,52	48,64	77,34	44281,22
AGE-CA	68,09	60,00	66,47	65875,02	77,50	35,00	69,00	63951,15	85,39	33,18	74,95	47506,18
AM-(10,1.0)	66,93	67,50	67,05	67126,71	78,12	39,17	70,33	65510,95	83,10	64,09	79,30	57534,08
AM-(10,0.1)	66,93	72,50	68,05	66140,59	82,50	51,94	76,39	58907,62	85,97	59,55	80,68	43543,32
AM-(10,0.1mej)	68,63	62,50	67,40	66059,92	80,00	48,33	73,67	58755,03	86,52	54,55	80,12	45210,14
BMB	65,76	75,00	67,61	29872,44	77,50	44,17	70,83	91795,27	86,22	55,91	80,16	67272,84
ES	65,37	65,00	65,29	15495,02	76,25	47,78	70,56	46149,68	85,95	60,00	80,76	26066,56
ILS	69,67	65,00	68,74	14836,62	78,44	55,28	73,81	84653,48	85,68	63,64	81,27	61939,50
ILS-ES	67,32	67,50	67,36	30082,13	77,50	56,39	73,28	85206,28	84,53	67,73	81,17	61421,92
VNS	66,93	72,50	68,05	16362,23	79,06	51,39	73,53	72268,20	83,66	71,36	81,20	46265,62

En la tabla 5.16, donde se recogen los resultados globales de la ejecución de los diferentes algoritmos para cada conjunto de datos, se destaca en **rojo**, para cada conjunto de datos, el **mejor valor de fitness** obtenido de entre todos los algoritmos, en **azul** el **segundo mejor valor de fitness** y en **verde** el **tercer mejor valor de fitness**.

## 5.2. Análisis de los Resultados Obtenidos (Práctica 1)

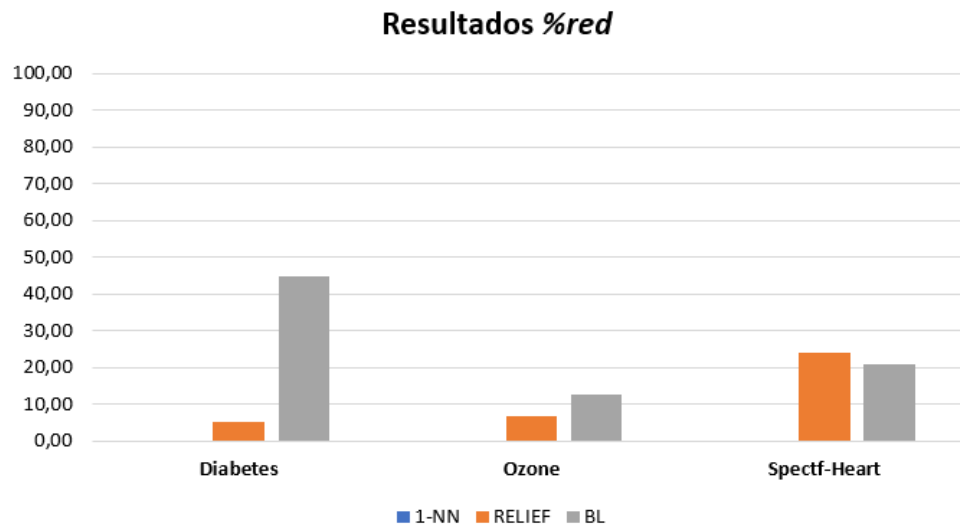
Vamos a comparar los diferentes resultados obtenidos por los diferentes algoritmos para cada conjunto de datos. Vamos a ir columna por columna (primero comparamos la tasa de clasificación media, luego la tasa reducción, después el fitness y finalmente el tiempo de ejecución).

- Tasa de clasificación:



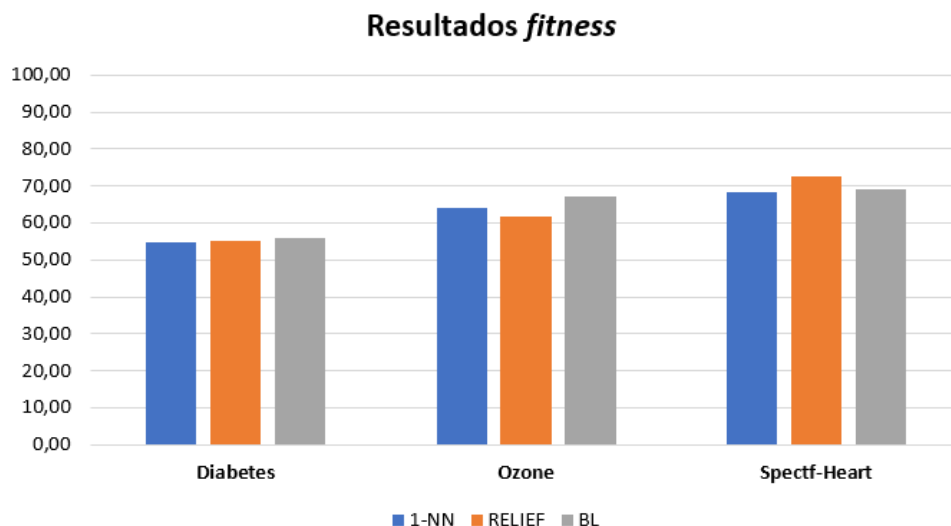
Observamos que, por lo general, la tasa de clasificación media obtenido por los tres algoritmos en cada conjunto de datos es muy **similar** (excepto en el algoritmo Greedy RELIEF para el conjunto de datos *diabetes*, la cual es notablemente inferior al resto).

- **Tasa de reducción:**



Vemos que se obtienen resultados esperados. Excepto en el conjunto de datos *spectf-heart* (donde la tasa de reducción media obtenida por el algoritmo Greedy RELIEF es ligeramente superior a la obtenida por el algoritmo de Búsqueda Local), en los otros dos conjuntos de datos **es superior la obtenida por el algoritmo de Búsqueda Local**.

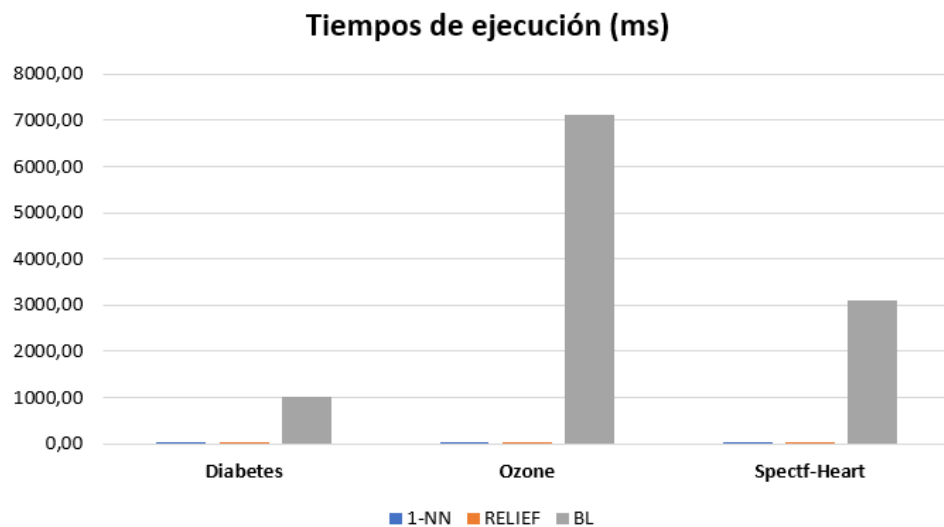
- **Fitness:**



Se puede observar que, excepto en el conjunto de datos *spectf-heart*, el **algoritmo de Búsqueda Local nos proporciona mejores resultados medios de fitness** para los diferentes conjuntos de datos. Esto se debe en gran parte, como hemos visto anteriormente, a la mayor tasa de reducción que nos proporciona este algoritmo con respecto a los otros.



- **Tiempo de ejecución:**

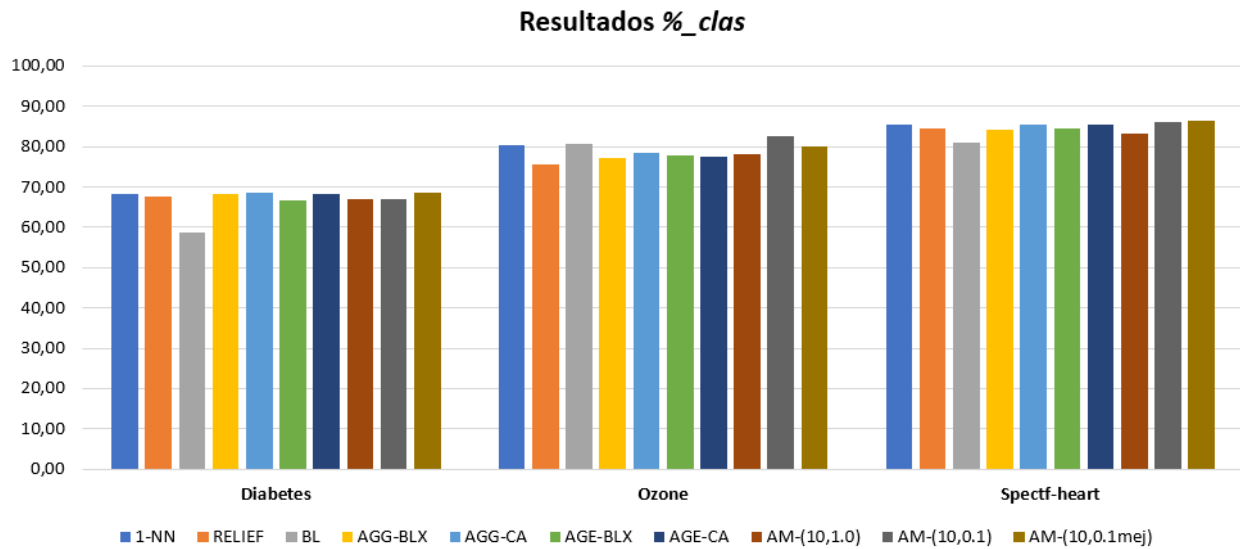


Se puede ver fácilmente que el **algoritmo de Búsqueda Local tarda de media muchísimo más** en ejecutarse con respecto a los otros algoritmos. Este es uno de los mayores inconvenientes que observamos en la Búsqueda Local, ya que, aunque nos proporcione datos muy buenos, el tiempo de ejecución puede llegar a ser desmesurado si tratamos con conjuntos de datos mucho más grandes de los que tenemos.

### **5.3. Análisis de los Resultados Obtenidos (Práctica 2)**

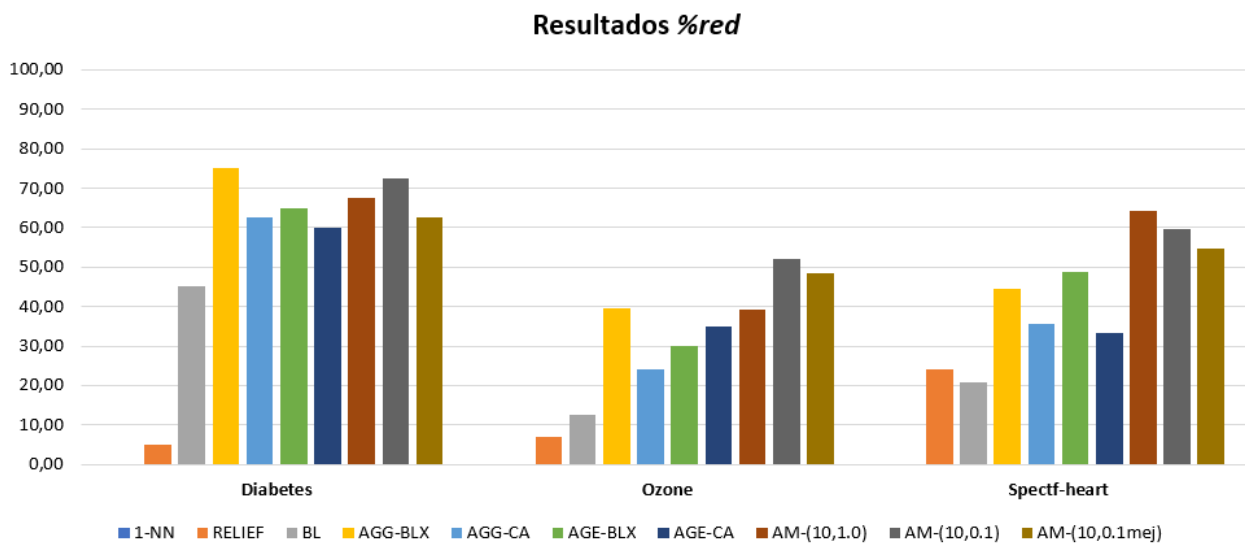
De igual manera que hicimos en la práctica 1, vamos a comparar los diferentes resultados obtenidos por los diferentes algoritmos para cada conjunto de datos. Vamos a ir columna por columna (primero comparamos la tasa de clasificación media, luego la tasa reducción, después el fitness y finalmente el tiempo de ejecución).

- Tasa de clasificación:



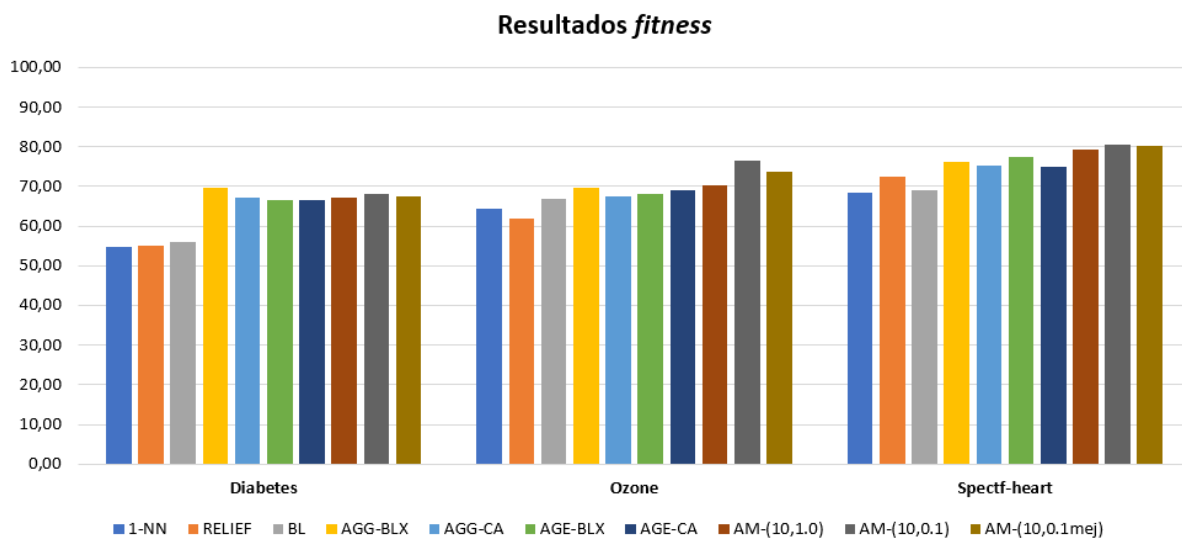
Observamos que, por lo general, la tasa de clasificación media obtenido por los diferentes algoritmos en cada conjunto de datos es muy **similar**. No hay gran mejora con respecto al algoritmo Greedy RELIEF o el algoritmo de Búsqueda Local implementados en la primera práctica (en algunos casos, incluso empeora). Podría destacarse que los **algoritmos meméticos** (en especial, **AM-(10,0.1)**) suelen ser mejores (o iguales como mínimo) que el resto.

- Tasa de reducción:



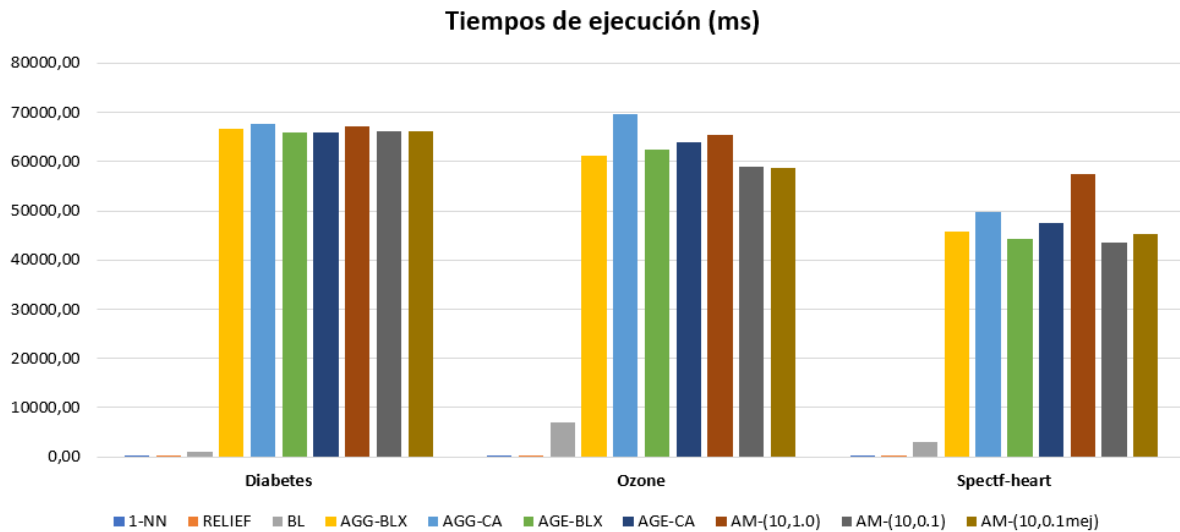
Con respecto a la tasa de reducción media obtenido por los diferentes algoritmos en cada conjunto de datos, sí que observamos variaciones de un algoritmo a otro. Para el conjunto de datos *diabetes*, es el algoritmo **AGG-BLX** el que tiene una mayor tasa de reducción mientras que para los conjuntos de datos *ozone* y *spectf-heart* son los **algoritmos meméticos** los que muestran un mayor valor de dicha tasa. Lo que se observa claramente es que todos los algoritmos implementados en esta segunda práctica superan a los de la primera práctica con respecto al valor de la tasa de reducción.

- **Fitness:**



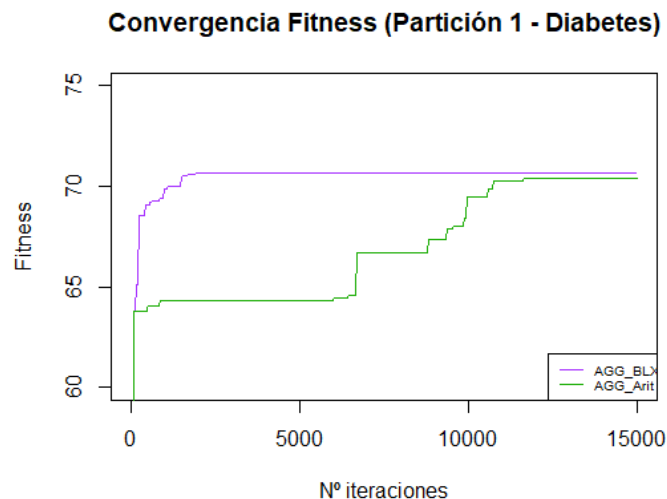
Se puede observar que, por lo general, los algoritmos desarrollados en esta segunda práctica ofrecen un mayor fitness con respecto a los de la primera práctica. También observamos que, para el conjunto de datos *diabetes*, es el algoritmo **AGG-BLX** el que tiene un mayor fitness mientras que para los conjuntos de datos *ozone* y *spectf-heart* son los algoritmos meméticos los que muestran un mayor fitness, en especial, el algoritmo **AM-(10,0.1)**. Esto último se debe en gran parte, como hemos visto anteriormente, a la mayor tasa de reducción que nos proporcionan estos dos algoritmos con su respectivo conjunto de datos donde destacan.

- **Tiempo de ejecución:**

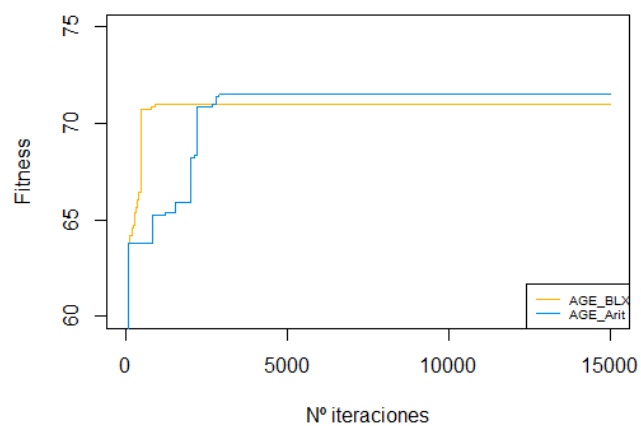


Se puede ver fácilmente que **los algoritmos desarrollados en esta segunda práctica tardan mucho más que los de la primera práctica**. Cabe destacar también que, entre los algoritmos de esta segunda práctica, **AM(10,0.1)** y **AM(10,0.1mej)** son los que ofrecen un menor tiempo de ejecución (aunque tampoco una diferencia exagerada).

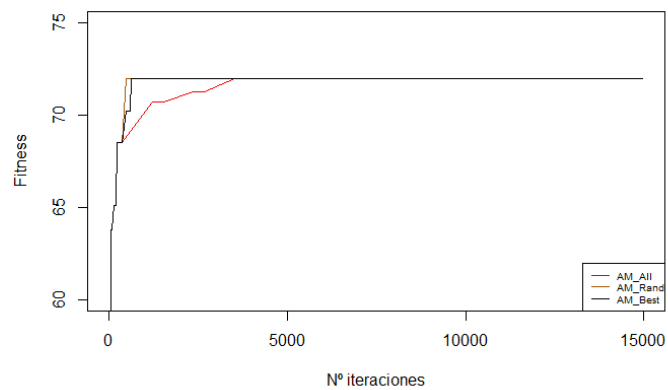
- **Convergencia:**



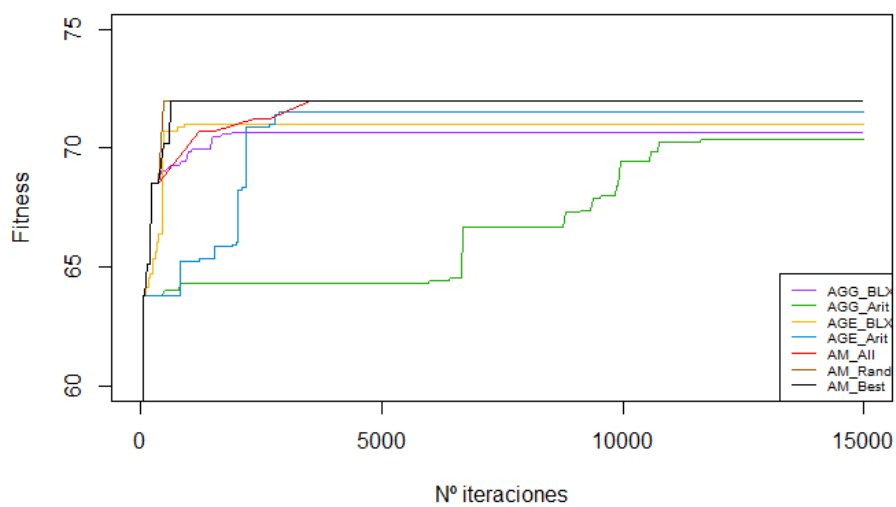
**Convergencia Fitness (Partición 1 - Diabetes)**



**Convergencia Fitness (Partición 1 - Diabetes)**



**Convergencia Fitness (Partición 1 - Diabetes)**



Se ha decidido hacer también un análisis comparativo de la velocidad de convergencia de los diferentes algoritmos desarrollados en esta segunda práctica para la primera partición del conjunto de datos *diabetes*. Para ello, se ha hecho uso de la librería *ofstream* de C++ para realizar salida de datos (num\_iteraciones y fitness, concretamente, a un fichero .txt) y, tras tener los ficheros de datos, hemos realizado un script en el lenguaje de programación R para obtener las diferentes gráficas (el código de este script se encuentra en la próxima imagen).

A primera vista, en la última gráfica de convergencia que recoge a todos los algoritmos de esta segunda práctica, se puede observar que el algoritmo **AGG-CA es el que más tarda en converger** de todos, y detrás de él le sigue el algoritmo AM\_All pero este si está un poco más igualado con respecto al resto de algoritmos. Lo que es claro es que **AM-(10,0.1) es el que converge más rápido** de todos, seguido de AM-(10,0.1mej) (dando estos dos algoritmos los mejores fitness para esa partición con respecto al resto).

```

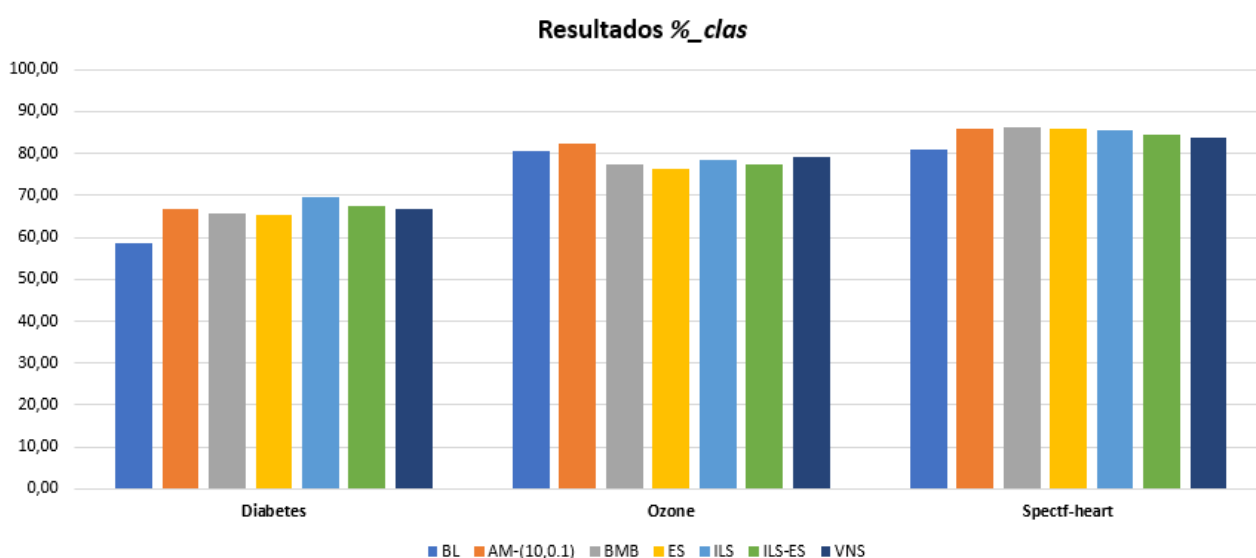
1  # Leer archivo de texto
2  datosAGG_BLX <- read.table("convergenciaAGG_BLX.txt", header = FALSE, col.names = c("x", "y"), sep = " ")
3  datosAGG_Arit <- read.table("convergenciaAGG_Arit.txt", header = FALSE, col.names = c("x", "y"), sep = " ")
4
5  datosAGE_BLX <- read.table("convergenciaAGE_BLX.txt", header = FALSE, col.names = c("x", "y"), sep = " ")
6  datosAGE_Arit <- read.table("convergenciaAGE_Arit.txt", header = FALSE, col.names = c("x", "y"), sep = " ")
7
8  datosAM_All <- read.table("convergenciaAM_All.txt", header = FALSE, col.names = c("x", "y"), sep = " ")
9  datosAM_Rand <- read.table("convergenciaAM_Rand.txt", header = FALSE, col.names = c("x", "y"), sep = " ")
10 datosAM_Best <- read.table("convergenciaAM_Best.txt", header = FALSE, col.names = c("x", "y"), sep = " ")
11
12 # Crear gráfico
13 plot(datosAGG_BLX$x, datosAGG_BLX$y, type = "l",
14       xlim = c(0, 15000), ylim = c(60, 75),
15       xlab = "Nº iteraciones", ylab = "Fitness",
16       main = "Convergencia Fitness (Partición 1 - Diabetes)",
17       col = rgb(168, 51, 255, maxColorValue = 255))
18
19 lines(datosAGG_Arit$x, datosAGG_Arit$y, type = "l",
20       col = rgb(30, 174, 8, maxColorValue = 255))
21
22
23 lines(datosAGE_BLX$x, datosAGE_BLX$y, type = "l",
24       col = rgb(255, 178, 0, maxColorValue = 255))
25
26 lines(datosAGE_Arit$x, datosAGE_Arit$y, type = "l",
27       col = rgb(0, 140, 225, maxColorValue = 255))
28
29 lines(datosAM_All$x, datosAM_All$y, type = "l",
30       col = rgb(240, 0, 0, maxColorValue = 255))
31
32 lines(datosAM_Rand$x, datosAM_Rand$y, type = "l",
33       col = rgb(166, 86, 0, maxColorValue = 255))
34
35 lines(datosAM_Best$x, datosAM_Best$y, type = "l",
36       col = rgb(0, 0, 0, maxColorValue = 255))
37
38 # Agregar leyenda al gráfico
39 legend("bottomright",
40       legend = c("AGG_BLX", "AGG_Arit", "AGE_BLX", "AGE_Arit", "AM_All", "AM_Rand", "AM_Best"),
41       col = c(rgb(168, 51, 255, maxColorValue = 255),
42               rgb(30, 174, 8, maxColorValue = 255),
43               rgb(255, 178, 0, maxColorValue = 255),
44               rgb(0, 140, 225, maxColorValue = 255),
45               rgb(240, 0, 0, maxColorValue = 255),
46               rgb(166, 86, 0, maxColorValue = 255),
47               rgb(0, 0, 0, maxColorValue = 255)),
48       lty = 1,
49       cex = 0.6)

```

## 5.4. Análisis de los Resultados Obtenidos (Práctica 3)

De igual manera que hicimos en las prácticas 1 y 2, vamos a comparar los diferentes resultados obtenidos por los diferentes algoritmos para cada conjunto de datos. Vamos a ir columna por columna (primero comparamos la tasa de clasificación media, luego la tasa reducción, después el fitness y finalmente el tiempo de ejecución).

- Tasa de clasificación:



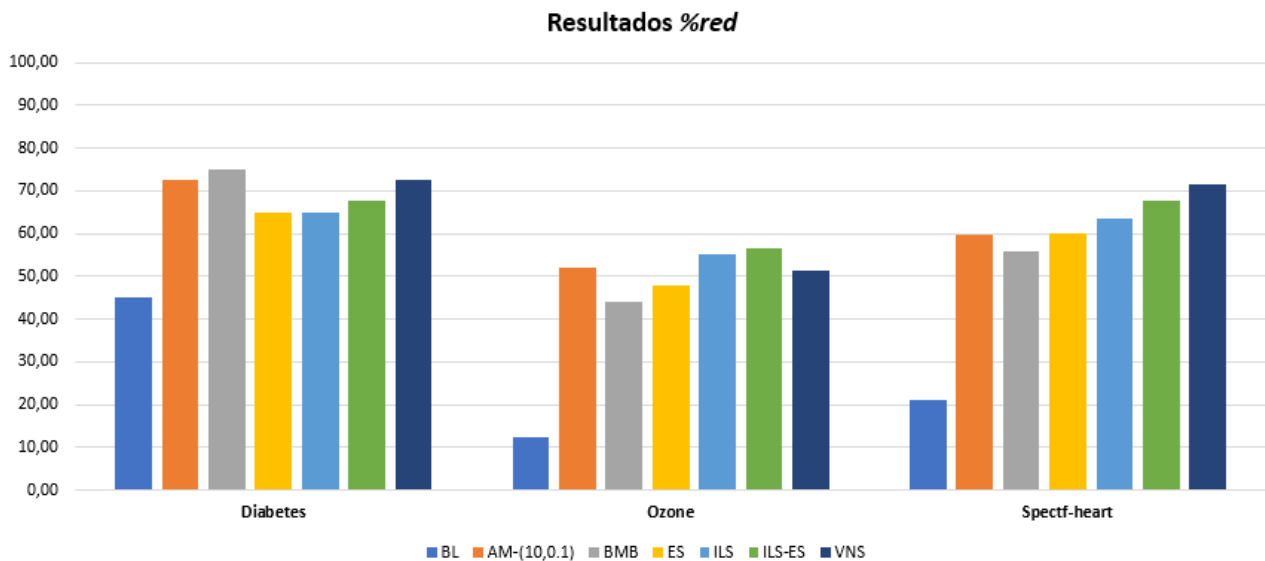
Observamos que, entre los algoritmos desarrollados en esta práctica, **el algoritmo ILS es el que obtiene mayor tasa de clasificación en general, seguido del algoritmo VNS.**

Con respecto al algoritmo de **Búsqueda Local** implementado en la primera práctica, vemos que **todos los algoritmos lo supera en tasa de clasificación** (excepto en el conjunto de datos *ozone*, donde la Búsqueda Local supera a todos).

Finalmente, con respecto al algoritmo **AM-(10,0.1)**, se observa que **como mínimo, los algoritmos de esta práctica igualan a dicho algoritmo (en general, lo superan, excepto en el conjunto de datos *ozone*, siendo AM-(10,0.1) el algoritmo con mayor tasa de clasificación de entre todos los considerados).**

En función de los conjuntos de datos, para el conjunto de datos **diabetes**, el algoritmo que tiene una mayor tasa de clasificación es el algoritmo **ILS**, para **ozone** es **AM-(10,0.1)** y para **spectf-heart** es **BMB**.

- Tasa de reducción:



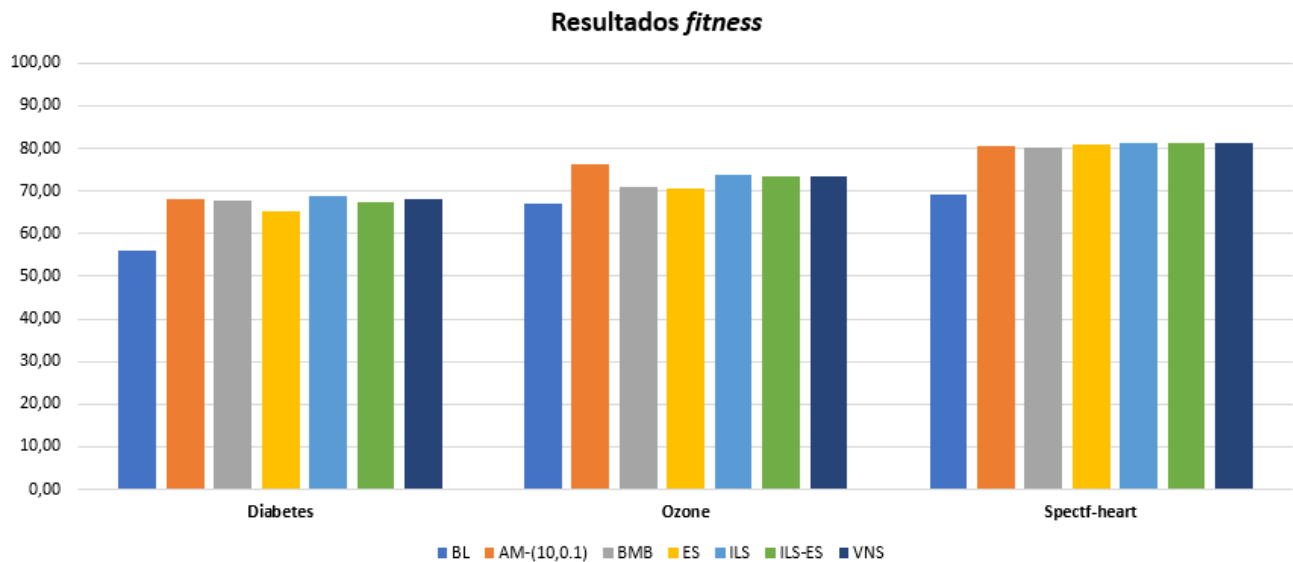
Para comenzar, considerando únicamente los algoritmos desarrollados en esta práctica, **dependiendo del conjunto de datos considerado un algoritmo tiene más tasa de reducción que los otros** (en el conjunto de datos *diabetes*, el que más tasa de reducción proporciona es el algoritmo **BMB**, en *ozone* es **ILS-ES** y en *spectf-heart* es **VNS**). Lo que sí parece apreciarse de forma constante es que el algoritmo **ES** suele ser el penúltimo en el ranking con una tasa de reducción más baja que el resto.

En comparación ahora con el algoritmo de **Búsqueda Local** implementado en la primera práctica, vemos que **todos los algoritmos lo supera en tasa de reducción** ya que la Búsqueda Local proporciona la tasa de reducción más baja de entre todos los algoritmos considerados.

Finalmente, comparando la tasa de reducción los algoritmos de esta práctica con la del algoritmo **AM-(10,0.1)**, observamos que **hay algunos que no lo superan o como máximo lo igualan** (como sucede con el algoritmo **ES**) y **hay otros que, dependiendo del conjunto de datos, sí que lo superan** (como sucede, por ejemplo, con el algoritmo **ILS**).



- **Fitness:**



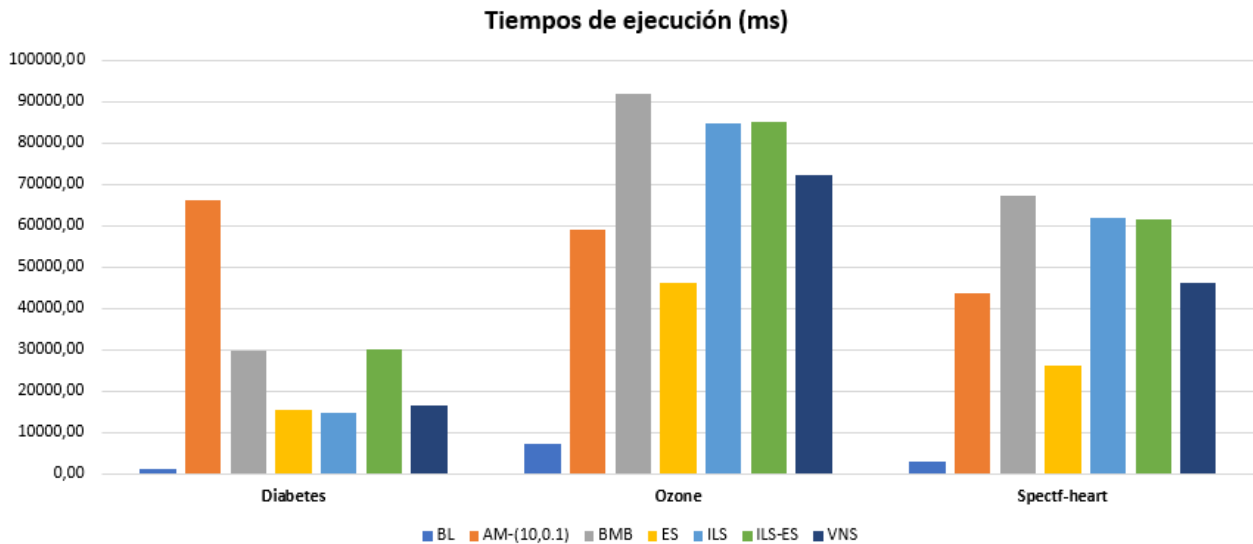
Se observa que, entre los algoritmos desarrollados en esta práctica, **el algoritmo ILS es el que obtiene mayor fitness en general, seguido del algoritmo VNS.**

Con respecto al algoritmo de **Búsqueda Local** implementado en la primera práctica, vemos que **todos los algoritmos lo supera en fitness en todos los conjuntos de datos.**

Finalmente, con respecto al algoritmo **AM-(10,0.1)**, se observa que, **en general, los algoritmos o casi lo igualan o lo superan por poco** (excepto en el conjunto de datos ozone, donde se ve claramente que AM-(10,0.1) supera a todos).

En función de los conjuntos de datos, para el conjunto de datos **diabetes**, el algoritmo que tiene una mayor tasa de clasificación es el algoritmo **ILS**, para **ozone** es **AM-(10,0.1)** y para **spectf-heart** es **ILS**.

- **Tiempo de ejecución:**

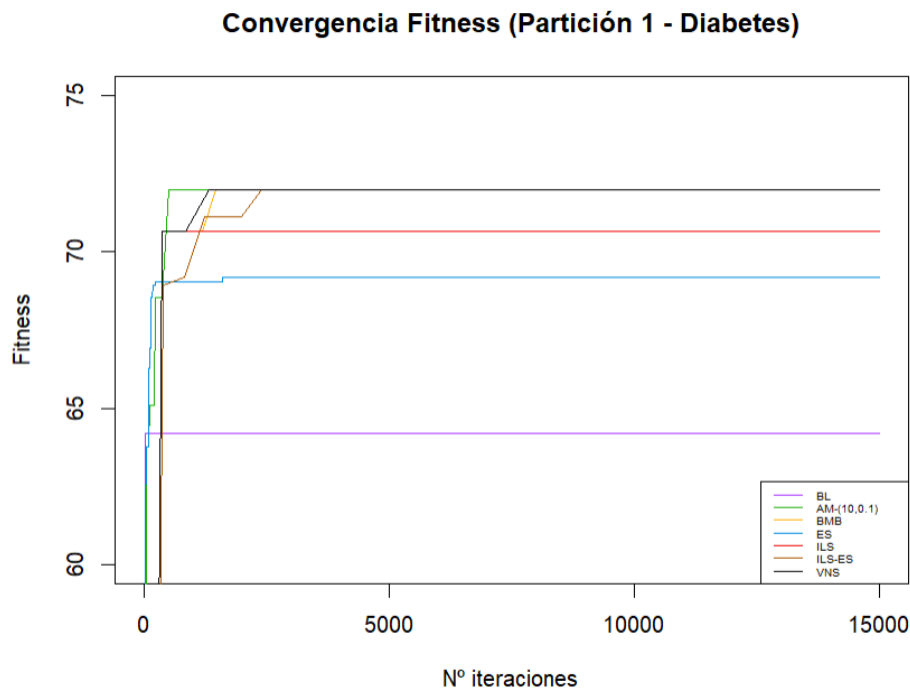


Con respecto al tiempo de ejecución, entre los algoritmos desarrollados en esta práctica, **es el algoritmo BMB es el que tiene un mayor tiempo de ejecución en general.**

Con respecto al algoritmo de **Búsqueda Local** implementado en la primera práctica, vemos que **todos los algoritmos lo superan en tiempo de ejecución con una gran diferencia.**

Finalmente, con respecto al algoritmo **AM-(10,0.1)**, se observa que, **en general, los algoritmos desarrollados en esta práctica suelen tardar más (excepto el algoritmo ES que siempre tarda menos en todos los conjuntos de datos y excepto en el conjunto de datos *diabetes*, siendo AM-(10,0.1) el que tarda más entre todos).**

- **Convergencia:**



Al igual que se hizo en la práctica 2, se ha decidido hacer también un análisis comparativo de la velocidad de convergencia de los diferentes algoritmos desarrollados en esta tercera práctica para la primera partición del conjunto de datos *diabetes*. Para ello, se ha hecho uso de la librería *ofstream* de C++ para realizar salida de datos (num\_iteraciones y fitness, concretamente, a un fichero .txt) y, tras tener los ficheros de datos, hemos realizado un script en el lenguaje de programación R para obtener las diferentes gráficas (el código de este script se encuentra en la próxima imagen).

A primera vista, se puede observar que el algoritmo **ILS-ES es el que más tarda en converger** de todos, y detrás de él le sigue el algoritmo ES. Lo que es claro es que el algoritmo de Búsqueda Local (BL) **es el que converge más rápido** de todos (aunque es el que peor fitness proporciona), seguido de AM-(10,0.1)

```

1  # Leer archivo de texto
2  datosBL <- read.table("convergenciaBL.txt", header = FALSE, col.names = c("x", "y"), sep = " ")
3  datosAM_Rand <- read.table("convergenciaAM_Rand.txt", header = FALSE, col.names = c("x", "y"), sep = " ")
4  datosBMB <- read.table("convergenciaBMB.txt", header = FALSE, col.names = c("x", "y"), sep = " ")
5  datosES <- read.table("convergenciaES.txt", header = FALSE, col.names = c("x", "y"), sep = " ")
6  datosILS <- read.table("convergenciaILS.txt", header = FALSE, col.names = c("x", "y"), sep = " ")
7  datosILS_ES <- read.table("convergenciaILS_ES.txt", header = FALSE, col.names = c("x", "y"), sep = " ")
8  datosVNS <- read.table("convergenciaVNS.txt", header = FALSE, col.names = c("x", "y"), sep = " ")
9
10 # Crear gráfico
11 plot(datosBL$x, datosBL$y, type = "l",
12       xlim = c(0, 15000), ylim = c(60, 75),
13       xlab = "Nº iteraciones", ylab = "Fitness",
14       main = "Convergencia Fitness (Partición 1 - Diabetes)",
15       col = rgb(168, 51, 255, maxColorValue = 255))
16
17 lines(datosAM_Rand$x, datosAM_Rand$y, type = "l",
18       col = rgb(30, 174, 8, maxColorValue = 255))
19
20
21 lines(datosBMB$x, datosBMB$y, type = "l",
22       col = rgb(255, 178, 0, maxColorValue = 255))
23
24 lines(datosES$x, datosES$y, type = "l",
25       col = rgb(0, 140, 225, maxColorValue = 255))
26
27 lines(datosILS$x, datosILS$y, type = "l",
28       col = rgb(240, 0, 0, maxColorValue = 255))
29
30 lines(datosILS_ES$x, datosILS_ES$y, type = "l",
31       col = rgb(166, 86, 0, maxColorValue = 255))
32
33 lines(datosVNS$x, datosVNS$y, type = "l",
34       col = rgb(0, 0, 0, maxColorValue = 255))
35
36 # Agregar leyenda al gráfico
37 legend("bottomright",
38       legend = c("BL", "AM-(10,0.1)", "BMB", "ES", "ILS", "ILS-ES", "VNS"),
39       col = c(rgb(168, 51, 255, maxColorValue = 255),
40             rgb(30, 174, 8, maxColorValue = 255),
41             rgb(255, 178, 0, maxColorValue = 255),
42             rgb(0, 140, 225, maxColorValue = 255),
43             rgb(240, 0, 0, maxColorValue = 255),
44             rgb(166, 86, 0, maxColorValue = 255),
45             rgb(0, 0, 0, maxColorValue = 255)),
46       lty = 1, cex = 0.5)

```

## 5.5. Conclusiones (Práctica 1)

En general, la **Búsqueda Local** nos proporciona mejores resultados (aunque sean óptimos locales, los resultados son suficientemente buenos) en comparación con los otros algoritmos debido a su gran tasa de reducción, pero a cambio de tener un tiempo de ejecución mucho mayor con respecto a los otros algoritmos (aunque siguen siendo tiempos de ejecución viables). Si tuviésemos que reducir el número de características sin importar si la ejecución tarda más (sin llegar a ser un tiempo desmesurado), la mejor opción sería la Búsqueda Local. Sin embargo, si tuviésemos un conjunto de datos muy grande en el que no sea muy necesario la reducción de características, no elegiría la Búsqueda Local por su gran tiempo de ejecución.

Con respecto al **algoritmo Greedy RELIEF**, podemos observar que el fitness que proporciona es muy similar con respecto al proporcionado por el clasificador 1-NN para el conjunto de datos *diabetes* e incluso peor con respecto al del clasificador 1-NN para el conjunto de datos *ozone* (al ser un algoritmo voraz, no está garantizada siempre una mejor solución). Sin embargo, para el conjunto de datos de *spectf-heart*, este produce una gran mejora del fitness con respecto al clasificador 1-NN. Al permitir que la tasa de reducción no sea 0 (lo cual es lo que hace que varíe con respecto al clasificador 1-NN, donde la tasa de reducción siempre es 0), podemos aumentar la simplicidad (aunque no tanto como con la Búsqueda Local) y, por tanto, mejorar el valor de la función objetivo. Una de las grandes ventajas de este algoritmo, como se puede observar es que su tiempo de ejecución es ínfimo (similar al del clasificador 1-NN).

**En conclusión, el algoritmo Greedy RELIEF nos proporciona resultados similares o ligeramente superiores a los del clasificador 1-NN y su tiempo de ejecución es ínfimo (parecido al del clasificador 1-NN), mientras que el algoritmo de Búsqueda Local nos proporciona destacablemente resultados mejores con respecto al clasificador 1-NN y al Greedy RELIEF pero una de sus grandes desventajas es el considerable aumento que produce del tiempo de ejecución.**

## 5.6. Conclusiones (Práctica 2)

En primer lugar, con respecto a los **algoritmos genéticos generacionales**, observamos que el **AGG-CA es un poco peor que el AGG-BLX** (tampoco hay una gran diferencia). Es posible que esto sea así porque el cruce BLX introduce capacidad de exploración de las soluciones y porque el cruce aritmético es determinista, mientras que el cruce BLX tiene un factor aleatorio que establece un equilibrio entre explotación y exploración, consiguiendo así una convergencia más rápida a soluciones mejores. En cuanto a la comparación con el algoritmo de búsqueda local de la primera práctica, vemos que ambos proporcionan mejores resultados de fitness a cambio de tardar entre 6 y 8 veces más que la búsqueda local.

En segundo lugar, con respecto a los **algoritmos genéticos estacionarios**, observamos que el **AGE-BLX y el AGE-CA ofrecen resultados similares (destacando AGE-BLX un poco más con respecto a AGE-CA)** en el conjunto de datos *spectf-heart*. Observamos también que el fitness dado por AGE-CA es muy similar al de AGE-CA, mientras que, en líneas generales, el fitness de AGE-BLX es ligeramente mayor al de AGE-BLX. En cualquier caso, en la implementación de los algoritmos estacionarios tenemos una probabilidad de mutación relativamente baja (al ser la población intermedia muy reducida), por lo que la diversidad introducida en la población está limitada. Esto hace que no se consiga mejorar mucho las soluciones y se observen largos períodos de estancamiento al converger prematuramente a óptimos locales. En este caso, el tiempo de ejecución es similar al de los algoritmos genéticos generacionales por lo que es mucho mayor con respecto al de la búsqueda local, pero a cambio obtenemos una mejora significativa en el fitness.

En tercer lugar, **los algoritmos meméticos (en especial AM-(10,0.1)) ofrecen una mejora notable del fitness con respecto al resto de algoritmos considerados** (tanto los de esta segunda práctica como los de la anterior). Al hibridar un algoritmo genético con alta capacidad de exploración, como es el AGE-BLX, con un algoritmo que explota muchas soluciones, como es la búsqueda local, conseguimos una rápida convergencia inicial, con capacidad de mejorar las soluciones y escapar de óptimos locales. El hecho de emplear una búsqueda local de baja intensidad hace que el tiempo de ejecución de estos algoritmos sea incluso un poco inferior al de los algoritmos genéticos. Un detalle a observar es que las soluciones para AM-(10,0.1) y AM-(10,0.1mej) se estancan muy rápido, por debajo de las 2500 iteraciones. Probablemente esto sea así por las iteraciones de la búsqueda local de baja intensidad que no conducen a mejora, por lo que podría pensarse en un criterio de parada alternativo de la búsqueda local que contemple estas situaciones y consiga disminuir el tiempo de ejecución de los algoritmos. Entre los tres algoritmos meméticos, cabe destacar **AM-(10,0.1)** por tener, en general, un menor tiempo de ejecución y ofrecer un mayor valor de fitness.

**En conclusión, si hubiera que elegir un solo algoritmo de entre todos, la elección más razonable sería AM-(10,0.1) para nuestra situación.** Por lo general, los algoritmos meméticos mejoran a los genéticos (ya que los meméticos se basan en los genéticos añadiendo búsqueda local), pero nunca podemos dar las cosas por hecho ya que hay muchos factores a tener en cuenta (como son el sesgo que puedan tener los conjuntos de datos, la semilla utilizada, etc.). Sí parece preferible usar estos algoritmos con respecto a los de la primera práctica, ya que aunque aumente el tiempo de ejecución de forma considerable, los valores de fitness mejoran notablemente.

## 5.7. Conclusiones (Práctica 3)

En primer lugar, cabe destacar que **todos los algoritmos de esta práctica son mejores con diferencia que la Búsqueda Local** de la primera práctica. En segundo lugar, parece observarse que **estos mismos algoritmos suelen, como mínimo, igualar al algoritmo AM-(10,0.1)**, el cual es el que mejor resultados nos proporcionó en la segunda práctica. En tercer lugar, se observa que, aunque sea por una diferencia muy pequeña, de cuestión de centésimas, **es el algoritmo ILS el que proporciona mejores resultados en general**.

En conclusión, considerando todos los algoritmos desarrollados en las tres prácticas, la mejor elección para nuestra situación sería el algoritmo ILS ya que es el que, en general, nos proporciona mejores resultados en un tiempo razonable (aunque supera considerablemente el tiempo de ejecución de la búsqueda local de la primera práctica, merece la pena). Cabe destacar también que los algoritmos de búsqueda basadas en trayectorias no necesitan estructuras de datos auxiliares como los basados en poblaciones, y el código es bastante sencillo e intuitivo. El objetivo de los algoritmos implementados en esta práctica era intentar corregir el principal problema que planteaba la búsqueda local: el estancamiento prematuro en óptimos locales. Podemos concluir con seguridad, que se ha logrado cumplir dicho objetivo. Tras implementar todos los algoritmos de las diferentes prácticas, la conclusión a la que podemos llegar es que es complicado encontrar una metaheurística que se adapte perfectamente a todos los conjuntos de datos y casos del problema, por lo que, si buscamos las mejores soluciones a un problema en un tiempo razonable, no deberíamos asegurar el mejor funcionamiento de una metaheurística u otra a priori, sino que deberíamos de probar siempre las diferentes metaheurísticas aplicadas al problema y ver cuál de ellas se adapta mejor.