Práctica Alternativa al Examen: Metaheurística Leaders and Followers

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. Introducción

Vamos a hablar acerca de la metaheurística ***Leaders and Followers*** (*Líderes y Seguidores*), la cual podemos enmarcar en el contexto de algoritmos que pretenden encontrar buenas soluciones en espacios de más de una dimensión.

Gonzalez-FernandezY, Chen S. Leaders and followers – A new metaheuristic to avoid the bias of accumulated information. In: 2015 IEEE Congress on Evolutionary Computation (CEC); 2015. p. 776–783.

La filosofía principal que sigue esta metaheurística consiste en que para lograr buenas soluciones en espacios *n*-dimensionales es más importante que la **exploración** (generación de soluciones en otros entornos distintos al de inicio) sea eficaz y que **prevalezca sobre la explotación** (capacidad de aproximar con precisión un extremo local de un entorno dado del espacio de búsqueda).

Siguiendo esta filosofía, necesitamos definir el concepto de **región de atracción**, la cual es una región del espacio de búsqueda en la cual se encuentran todos los posibles puntos de inicio en los que al aplicar búsqueda local u otro algoritmo dan como resultado un extremo local particular. Dicho de otra forma, serían regiones del espacio de búsqueda que nos garantizan la aproximación a un extremo local concreto al iniciar un algoritmo en cualquier punto de ellas.

2. Motivación

Para motivar la aparición y el uso de la metaheurística *Leaders and Followers* en diferentes problemas, vamos a fijarnos en un ejemplo concreto. Consideramos la **función de Rastringin** dada por la siguiente expresión:

$$f(x) = 10n + \sum_{i=1}^{n} [(x_i)^2 - 10\cos(2\pi x_i)],$$

donde *n* indica la dimensión y $x_i \in$ [-5.12,5.12].

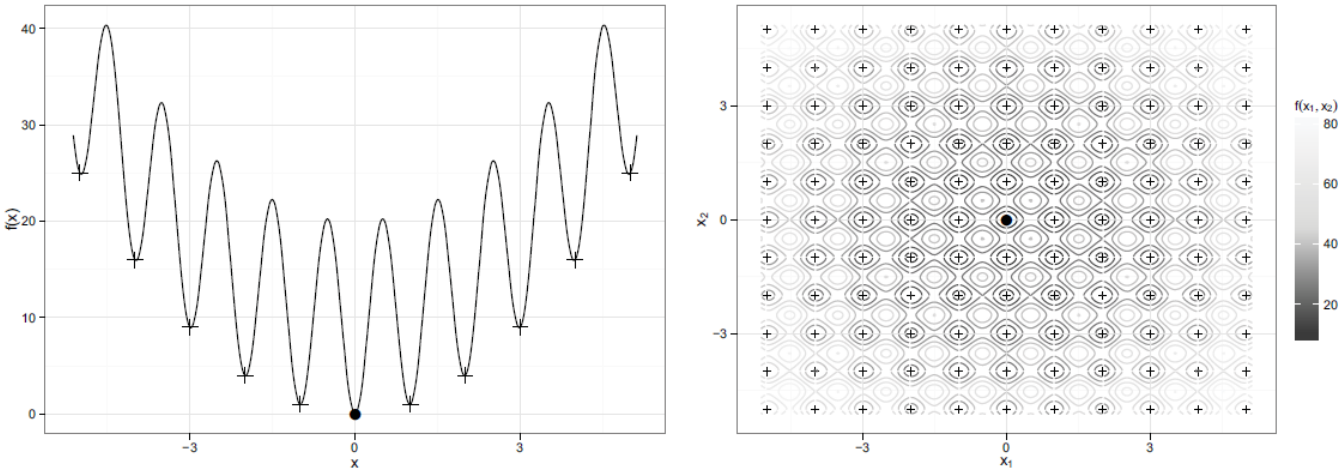


Figura 1: Función de Rastringin en una dimensión (izquierda) y en dos dimensiones (derecha). Las cruces indican óptimos locales y el círculo negro indica el óptimo global.

Esta función alcanza su mínimo global en $x = 0$, siendo por tanto $f(x) = 0$, pero tiene muchos otros extremos locales en el espacio de búsqueda, tal y como se puede apreciar en la Figura 1. Las regiones de atracción serían las regiones que rodean los extremos locales.

Fijándonos en esta función, podríamos obtener una medida de lo *alta* que está una posible solución dentro de una región de atracción calculando la diferencia entre el fitness de la solución y el valor del óptimo local. Como las regiones de atracción son convexas, la **explotación** del problema resulta **sencilla**, pero **no resulta así** para nada con respecto a la **exploración**, es decir, saltar a otras regiones de atracción *mejores*.

Con esta intuición, se realizó un **experimento** cuya **conclusión** fue que si tenemos una solución en una región de atracción que se aproxima muy bien a su correspondiente extremo local y generamos nuevos valores aleatorios en el espacio de búsqueda y los comparamos con dicha solución, entonces las conclusiones que saquemos estarán sesgadas por el conocimiento que tengamos de la región de atracción de la solución inicial, y no solo eso, sino que **cuanto más optimizada (más próxima al extremo local) esté dicha solución, más complicado será saltar a otra región de atracción**, ya que la diferencia entre la solución inicial y el extremo local al que se aproxima será muy pequeña.

En resumen, si en algún punto del experimento se consigue aproximar muy bien una solución a un extremo local, resultará muy difícil saltar a otra región de atracción con un extremo mejor, lo cual nos da a entender que **las metaheurísticas que comparan en cada iteración las nuevas soluciones generadas con la mejor solución actual podrían caer en este mismo problema**. Para evitar esto, surge la metaheurística *Leaders and Followers* que se detalla a continuación.

3. Metaheurística *Leaders and Followers*.

Desarollo e Implementación para el problema APC

En base a las conclusiones del experimento explicado anteriormente, se identifican las siguientes **premisas**, que cumplirá la metaheurística, con el fin de lograr una exploración eficaz:

1) **Se debe evitar realizar comparación directa entre los elementos generados para la exploración y la mejor solución actual.**

2) **La exploración entre regiones de atracción es más insesgada al comienzo del proceso de búsqueda.**

Por un lado, el principal argumento que secunda la primera premisa es que la mejor solución actual será muy próxima al extremo local que aproxima y, por ello, las muestras de soluciones de una región de atracción raramente estarán tan cerca del extremo local como la mejor solución conocida hasta el momento.

Por otro lado, el principal argumento en el que se basa la segunda premisa es que la *altura* en las regiones de atracción cae muy rápidamente si usamos una búsqueda aleatoria uniforme, luego, este es el motivo por el que es al comienzo del proceso de búsqueda cuando se dan las mejores condiciones para comparar regiones de atracción.

Para cumplir con estas dos premisas, **la metaheurística *Leaders and Followers* propone emplear dos poblaciones separadas**, la primera denominada ***Leaders*** (Líderes) , que se encargará de **mantener las mejores soluciones encontradas hasta el momento**, y la segunda ***Followers*** (Seguidores), que se encargará de la **búsqueda de nuevas soluciones**.

Esta separación en dos poblaciones hace que se cumpla la primera premisa, ya que las nuevas soluciones que generen los *Followers* serán comparadas con las demás soluciones de esta misma población, sin que haya interacción con ninguna solución de la población de *Leaders*, la cual se actualizará con las soluciones de los *Followers* solo después de que se hayan realizado las búsquedas suficientes en la población de *Followers*.

Por otra parte, para hacer que se cumpla la segunda premisa, durante la ejecución del algoritmo se realizarán reinicios rápidos frecuentes en la población de los *Followers*, permitiendo así que la exploración sea más eficiente gracias a que eliminan el sesgo entre las regiones de atracción.

Teniendo en cuenta todo lo anterior, el **pseudocódigo del algoritmo *Leaders and Followers*** para minimizar una función objetivo *n*-dimensional f sería el siguiente:

```
1: L <- Inicializar los líderes con n vectores aleatorios uniformes
2: F <- Inicializar los seguidores con n vectores aleatorios uniformes
3: repeat
4:     for i:=1 to n do
5:         leader <- Tomar un líder de L
6:         follower <- Tomar un seguidor de F
7:         trial <- create_trial(leader,follower)
8:         if f(trial) < f(follower) then
9:             Sustituir follower por trial en F
10:    if median(f(F)) < median(f(L)) then
11:        L <- merge_populations(L,F)
12:        F <- Reiniciar los seguidores con vectores aleatorios uniformes
13: until El criterio de terminación se satisfaga
```

En primer lugar, las poblaciones de *Leaders* (L) y *Followers* (F) se inicializan con n soluciones aleatorias muestreadas uniformemente en el dominio del espacio de búsqueda.

El bucle principal del algoritmo comienza en la línea 3 y se ejecutará hasta que se satisfaga el criterio de terminación. Cada iteración del bucle principal comienza con una ronda de actualizaciones en los *Followers* (líneas 4-9), para lo cual se crean n pares (*leader*,*follower*) de manera que se crea un nuevo *trial* usando la función *create_trial* (línea 7) que implementa la siguiente fórmula:

$$trial_i = follower_i + \epsilon_i 2(leader_i - follower_i)\,,$$

donde ϵ_i es un parámetro que se elige aleatoriamente para cada dimensión siguiendo una distribución uniforme (0,1). Si el nuevo *trial* es mejor que *follower*, entonces reemplaza a este en la población de *Followers* (fijémonos que se compara con los *followers*, no con los *leaders*). Por lo tanto, las nuevas soluciones se generan en un hiper-rectángulo centrado en el *leader* seleccionado, tal y como se muestra en la Figura 2.

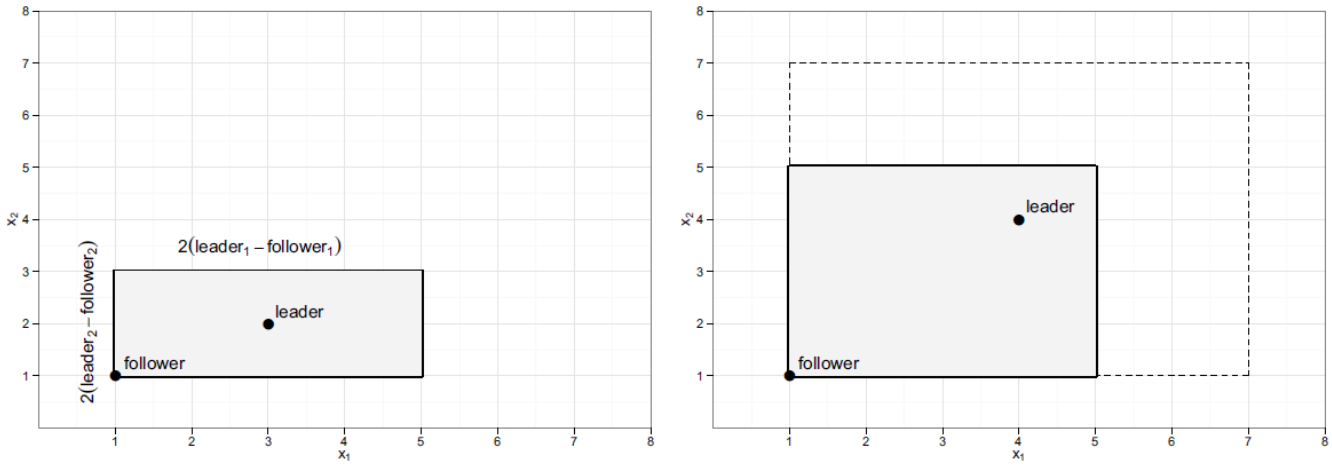


Figura 2: La región sombreada de gris indica las regiones de búsqueda de nuevas soluciones cuando no hay efectos de frontera (izquierda) y cuando están limitadas en la frontera en cada dimensión (derecha)

En la segunda parte del bucle principal (líneas 10-12) se comprueba si se ha realizado una búsqueda suficiente en el contexto de los *followers*. Si la mediana del fitness de los *followers* es mejor que la mediana del fitness de los *leaders*, entonces ambas poblaciones son comparables y es seguro fusionarlas en cuanto al riesgo de incurrir en comparaciones sesgadas.

Finalmente, la función *merge_populations* (línea 11) toma las poblaciones de *Leaders* y de *Followers* y devuelve un grupo de *n* individuos (que pasará a ser la población de *Leaders*) de la siguiente manera: mantiene el mejor *leader* hasta el momento y luego realiza un torneo binario sin reemplazamiento entre el resto de individuos para obtener los *n*-1 *leaders* restantes. Una vez actualizada la población de *Leaders*, la población de *Followers* se reinicia como soluciones muestreadas uniformemente, reintroduciendo así la oportunidad de realizar comparaciones no sesgadas al eliminar la información acumulada (recordemos la segunda premisa).

Se ha decidido **implementar esta metaheurística para el problema de Aprendizaje de Pesos en Características (Problema APC)** debido a su facilidad para adaptarla a dicho 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 $X = \{x_1, x_2, ..., x_n\}$, donde cada dato de dicha muestra está formado por un conjunto de características $\{c_1, c_2, ..., c_m,\}$ y su categoría, de forma que buscamos obtener un vector de pesos $W = \{w_1, w_2, ..., w_m\}$ 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 usamos 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.

Realizamos dos tareas: primero obtenemos el vector de pesos (aprendizaje) y a continuación valoramos la calidad del mismo (validación). Usamos 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. Se aprende un clasificador utilizando el 80% de los datos disponibles (4 particiones de las 5) y validaremos con el 20% restante (la partición restante). Así obtenemos 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 mide 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 →** $\boldsymbol{tasa\_clas = 100 * \frac{Nº\ de\ instancias\ bien\ clasificadas\ en\ T}{Nº\ de\ instancias\ en\ T}}$

- **Simplicidad →** $\boldsymbol{tasa\_red = 100 * \frac{Nº\ de\ valores\ w_i < 0.1}{Nº\ de\ características}}$

Utilizamos una agregación sencilla que combine ambos objetivos en un único valor. Así, la **función objetivo** es $\boldsymbol{F(w) = \alpha * tasa\_clas(w) + (1 - \alpha) * tasa\_red(w)}$. El valor de α pondera la importancia entre el acierto y la reducción de características de la solución encontrada (el clasificador generado). Usamos α = 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 han ejecutado sobre **tres conjuntos de datos normalizados distintos: *diabetes*, *ozone* y *spectf-heart***.

Una vez explicado en qué consiste el problema APC, pasamos a la explicación del desarrollo e implementación de la metaheurística *Leaders and Followers* a dicho problema.

Para representar las soluciones, empleamos un ***struct*** que representa una **solución** (un *leader* o un *follower*). En este contexto, una solución será un vector de pesos (una posible solución) ya evaluado, es decir, con un valor otorgado por la función objetivo (*fitness*).

Para representar una **población** de soluciones (un conjunto de soluciones) utilizamos el tipo de dato ***multiset*** (contenedor asociativo y ordenado que admite repetidos). Es importante permitir repetidos, pues podemos tener una población con varias soluciones iguales. Definimos la siguiente relación de orden entre cromosomas: *Si S_1 y S_2 son dos soluciones, diremos que $S_1 < S_2$ si $S_1.fitness < S_2.fitness$.*

```
// ************************************************************************
// ************************ Estructura de Datos ************************
// ************************************************************************

// Leader o Follower
struct LF {
  vector<double> w;  // Vector de pesos
  double fitness;     // Valor de la función objetivo para w
};

// Comparación de Leaders/Followers
struct LFComp {
  bool operator()(const LF& lhs, const LF& rhs) const {
    return lhs.fitness < rhs.fitness;
  }
};

// Población
typedef multiset<LF, LFComp> PoblacionLF;
```

De esta forma, siguiendo el pseudocódigo dado anteriormente, la implementación de la metaheurística *Leaders and Followers* es la siguiente:

```
vector<double> LeadersAndFollowers(const vector<Ejemplo>& entrenamiento) {

  int num_iteraciones = 0;

  PoblacionLF leaders;
  PoblacionLF followers;

  vector<double> w(entrenamiento[0].num_caracts);

  int tamanio_poblacion = w.size();

  leaders = inicializarPoblacionLF(entrenamiento);
  followers = inicializarPoblacionLF(entrenamiento);

  while(num_iteraciones < MAX_ITER_LEADERS_AND_FOLLOWERS) {

    for(int i = 0; i < tamanio_poblacion; ++i) {
      auto it_l = leaders.rbegin();
      advance(it_l, i);
      LF leader = *it_l;

      auto it_f = followers.rbegin();
      advance(it_f, i);
      LF follower = *it_f;

      LF trial = createTrial(leader, follower, entrenamiento);

      if(trial.fitness > follower.fitness)
        follower = trial;
    }

    if( calcularMediana(leaders) > calcularMediana(followers) ) {
      leaders = mergePopulations(leaders, followers);
      followers = inicializarPoblacionLF(entrenamiento);
    }

    num_iteraciones++;
  }

  w = leaders.rbegin()->w;

  return w;
}
```

En dicha implementación cabe destacar que como criterio de parada se ha establecido el número de iteraciones y que los signos < que aparecían en el pseudocódigo de la metaheurística se han sustituido por signos >, ya que dicho pseudocódigo era para encontrar mínimos de una función, pero en el problema APC queremos encontrar máximos.

Las funciones auxiliares que aparecen el código principal de la metaheurística (destacadas en la imagen anterior con un rectángulo rojo) se han implementado como sigue:

```
PoblacionLF inicializarPoblacionLF(const vector<Ejemplo>& entrenamiento) {

  PoblacionLF p;

  uniform_real_distribution<double> distribucion_uniform_real(0.0, 1.0);

  int tamanio_poblacion = entrenamiento[0].num_caracts;

  for(int i = 0; i < tamanio_poblacion; ++i) {
    LF individuo;

    individuo.w.resize(tamanio_poblacion);

    for(int j = 0; j < individuo.w.size(); ++j)
      individuo.w[j] = Random::get(distribucion_uniform_real);  // Inicializar vector de pesos

    individuo.fitness = fitness(tasaClasificacionLeaveOneOut(entrenamiento, individuo.w), tasaReduccion(individuo.w));    // Calcular fitness

    p.insert(individuo);
  }

  return p;
}
```

```
LF createTrial(const LF& leader, const LF& follower, const vector<Ejemplo>& entrenamiento) {

  LF trial;

  trial.w.resize(leader.w.size());

  uniform_real_distribution<double> distribucion_uniform_real(0.0, 1.0);

  double epsilon = Random::get(distribucion_uniform_real);

  for(int i = 0; i < trial.w.size(); ++i) {
    trial.w[i] = follower.w[i] + epsilon * 2 * (leader.w[i] - follower.w[i]);

    if(trial.w[i] > 1.0)
      trial.w[i] = 1.0;
    if(trial.w[i] < 0.0)
      trial.w[i] = 0.0;
  }

  trial.fitness = fitness(tasaClasificacionLeaveOneOut(entrenamiento, trial.w), tasaReduccion(trial.w));

  return trial;
}
```

```cpp
double calcularMediana(const PoblacionLF& p) {

  vector<double> v_fitness(p.size());

  for(int i = 0; i < p.size(); ++i) {
    auto it = p.begin();
    LF individuo = *it;
    v_fitness.push_back(individuo.fitness);
  }

  double mediana;

  if(v_fitness.size() % 2 == 0)
    mediana  = ( v_fitness[v_fitness.size() / 2] + v_fitness[ (v_fitness.size() / 2) - 1] ) / 2.0;
  else
    mediana = v_fitness[v_fitness.size() / 2];

  return mediana;
}
```

```cpp
PoblacionLF mergePopulations(const PoblacionLF& leaders, const PoblacionLF& followers) {
  PoblacionLF p;

  PoblacionLF copia_leaders = leaders;
  PoblacionLF copia_followers = followers;

  auto it = copia_leaders.rbegin();
  LF mejor_leader = *it;
  p.insert(mejor_leader);
  copia_leaders.erase(std::next(it).base());

  // Torneo binario para obtener los n-1 individuos restantes de la nueva población
  while(p.size() < leaders.size()) {
    LF ganador;

    if (!copia_leaders.empty() && !copia_followers.empty()) {
      uniform_int_distribution<int> distribucion_uniform_int_leaders(0, copia_leaders.size() - 1);
      uniform_int_distribution<int> distribucion_uniform_int_followers(0, copia_followers.size() - 1);

      auto it_l = copia_leaders.begin();
      auto it_f = copia_followers.begin();

      advance(it_l, Random::get(distribucion_uniform_int_leaders));
      advance(it_f, Random::get(distribucion_uniform_int_followers));

      if (it_l->fitness > it_f->fitness)
        ganador = *it_l;
      else
        ganador = *it_f;

      p.insert(ganador);

      copia_leaders.erase(it_l);
      copia_followers.erase(it_f);
    }
    else if (!copia_leaders.empty()) {
      auto it_l = copia_leaders.begin();
      ganador = *it_l;

      p.insert(ganador);
      copia_leaders.erase(it_l);
    }
    else if (!copia_followers.empty()) {
      auto it_f = copia_followers.begin();
      ganador = *it_f;

      p.insert(ganador);
      copia_followers.erase(it_f);
    }
  }

  return p;
}
```

4. Propuesta de Mejora mediante Hibridación

Como propuesta de mejora se propone una hibridación de dicha metaheurística con una búsqueda local. Como ya se comentó, en la metaheurística *Leaders and Followers* prevalece la exploración sobre la explotación, por lo que hibridar dicha metaheurística con una búsqueda local (que es un algoritmo de explotación) es probable que proporcione una mejora en los resultados.

De esta forma, se ha implementado la siguiente búsqueda local de baja intensidad (llamada de baja intensidad debido a su criterio de terminación):

```
int busquedaLocalBajaIntensidad(const vector<Ejemplo>& entrenamiento, LF& individuo) {

  normal_distribution<double> distribucion_normal(0.0, SIGMA);

  vector<int> indices;     // Se utiliza para seleccionar la componente de w a mutar
  double mejor_fitness;

  int num_iteraciones = 0;

  // Inicializamos el vector de índices
  for(int i = 0; i < individuo.w.size(); ++i)
    indices.push_back(i);

  Random::shuffle(indices);

  mejor_fitness = individuo.fitness;

  // Búsqueda en el vecindario del primero mejor
  while(num_iteraciones < individuo.w.size() * COEF_MAX_VECINOS_BAJA_INTENSIDAD ) {

    // Seleccionamos la componente de w para mutar
    int comp_mutada = indices[num_iteraciones % individuo.w.size()];

    // Mutación Normal de w
    LF individuo_mutado = individuo;
    individuo_mutado.w[comp_mutada] += Random::get(distribucion_normal);

    // Truncamos el peso de la componente mutada si fuera necesario
    if(individuo_mutado.w[comp_mutada] > 1)
      individuo_mutado.w[comp_mutada] = 1;
    else if(individuo_mutado.w[comp_mutada] < 0)
      individuo_mutado.w[comp_mutada] = 0;

    // Vemos si el fitness dado por individuo_mutado mejora mejor_fitness, en cuyo caso actualizamos dicho valor
    individuo_mutado.fitness = fitness(tasaClasificacionLeaveOneOut(entrenamiento, individuo_mutado.w), tasaReduccion(individuo_mutado.w));

    num_iteraciones++;

    if(individuo_mutado.fitness > mejor_fitness) {
      individuo = individuo_mutado;
      mejor_fitness = individuo_mutado.fitness;
    }

    // Actualizamos el vector de índices si ya se ha recorrido entero sin encontrar mejora
    if(num_iteraciones % individuo.w.size() == 0)
      Random::shuffle(indices);
  }

  return num_iteraciones;
}
```

Se ha decidido aplicar dicha búsqueda local al individuo *trial* que se crea en el algoritmo, ya que dicha parte es la parte principal de explotación de la metaheurística por lo que al aplicarle la búsqueda local estaríamos intensificándola. El código de la metaheurística *Leaders and Followers* con dicha mejora sería el siguiente:

```
vector<double> LeadersAndFollowersBL(const vector<Ejemplo>& entrenamiento) {

  int num_iteraciones = 0;

  PoblacionLF leaders;
  PoblacionLF followers;

  vector<double> w(entrenamiento[0].num_caracts);

  int tamanio_poblacion = w.size();

  leaders = inicializarPoblacionLF(entrenamiento);
  followers = inicializarPoblacionLF(entrenamiento);

  while(num_iteraciones < MAX_ITER_LEADERS_AND_FOLLOWERS) {

    for(int i = 0; i < tamanio_poblacion; ++i) {
      auto it_l = leaders.rbegin();
      advance(it_l, i);
      LF leader = *it_l;

      auto it_f = followers.rbegin();
      advance(it_f, i);
      LF follower = *it_f;

      LF trial = createTrial(leader, follower, entrenamiento);

      num_iteraciones += busquedaLocalBajaIntensidad(entrenamiento, trial);

      if(trial.fitness > follower.fitness)
        follower = trial;
    }

    if( calcularMediana(leaders) > calcularMediana(followers) ) {
      leaders = mergePopulations(leaders, followers);
      followers = inicializarPoblacionLF(entrenamiento);
    }

    num_iteraciones++;
  }

  w = leaders.rbegin()->w;

  return w;
}
```

5. Propuesta de Mejora del Diseño

Como propuesta de mejora de mejora sobre el diseño/comportamiento de la metaheurística se ha decidido (entre varias opciones que se han barajado y testeado) modificar la función *createTrial* de la siguiente forma:

```
LF createTrial(const LF& leader, const LF& follower, const vector<Ejemplo>& entrenamiento) {

  LF trial;

  trial.w.resize(leader.w.size());

  uniform_real_distribution<double> distribucion_uniform_real(0.0, 1.0);

  double epsilon = Random::get(distribucion_uniform_real);

  for(int i = 0; i < trial.w.size(); ++i) {
    trial.w[i] = follower.w[i] + epsilon * 2 * (leader.w[i] - follower.w[i]);

    if(trial.w[i] > 1.0)
      trial.w[i] = 1.0;
    if(trial.w[i] < 0.0)
      trial.w[i] = 0.0;
  }

  trial.fitness = fitness(tasaClasificacionLeaveOneOut(entrenamiento, trial.w), tasaReduccion(trial.w));

  return trial;
}
```

```
LF createTrialModificado(const LF& leader, const LF& follower, const vector<Ejemplo>& entrenamiento, double mutacion) {

  LF trial;

  trial.w.resize(leader.w.size());

  normal_distribution<double> distribucion_normal(0.0, mutacion);

  for(int i = 0; i < trial.w.size(); ++i) {
    double promedio = (leader.w[i] + follower.w[i]) / 2.0;
    double delta = Random::get(distribucion_normal);

    trial.w[i] = promedio + delta;

    if(trial.w[i] > 1.0)
      trial.w[i] = 1.0;
    if(trial.w[i] < 0.0)
      trial.w[i] = 0.0;
  }

  trial.fitness = fitness(tasaClasificacionLeaveOneOut(entrenamiento, trial.w), tasaReduccion(trial.w));

  return trial;
}
```

La función ***createTrial*** se basa en la exploración y explotación generando una nueva solución de prueba combinando los pesos de un *leader* y un *follower*, utilizando un factor de exploración *epsilon*. El valor *epsilon* se selecciona aleatoriamente entre 0 y 1 para introducir aleatoriedad en la exploración. La nueva solución generada, *trial*, se ajusta para asegurarse de que los valores de peso estén entre 0 y 1. Finalmente, se calcula el fitness de dicha solución *trial*.

Por otro lado, la función ***createTrialModificado*** introduce una modificación en la forma en la que se genera la nueva solución de prueba. Se utiliza un factor de mutación, *mutacion*, que determina la magnitud de la perturbación aplicada a la solución promedio de *leader* y *follower*. Se usa una distribución normal de media 0 y desviación típica *mutacion* para generar una perturbación aleatoria, *delta*, alrededor del promedio. La nueva solución generada, *trial*, se obtiene sumando el promedio y la perturbación *delta*. Al igual que en *createTrial*, la nueva solución generada, *trial*, se ajusta para asegurarse de que los valores de peso estén entre 0 y 1. Finalmente, se calcula el fitness de dicha solución *trial*.

En términos de exploración y explotación, se puede decir que:

- *createTrial* tiene un enfoque más equilibrado entre exploración y explotación a lo largo de toda la ejecución del algoritmo, inclinándose más hacia la explotación controlada con cierta exploración. La combinación de pesos con *epsilon* introduce exploración al permitir que la nueva solución se aleje de las soluciones existentes. La influencia de la exploración se controla mediante dicho factor *epsilon*. Esta función también se beneficia de la explotación al utilizar los pesos de *leader* y *follower* como base para la nueva solución.

- *createTrialModificado* tiene un enfoque más adaptativo entre exploración y explotación a lo largo de toda la ejecución del algoritmo, comenzando con una preferencia hacia la exploración y terminando más enfocado a la explotación. La introducción de la perturbación aleatoria mediante la distribución normal amplía la búsqueda en el espacio de soluciones. La magnitud de la perturbación está controlada por el factor *mutacion*. Valores más altos de *mutacion* favorecen la exploración al introducir mayores perturbaciones y aumentar la diversidad de las soluciones de prueba. Valores más bajos de *mutacion* promueven la explotación al generar soluciones de prueba más similares a las soluciones existentes. Se aplica un esquema de reducción lineal en el algoritmo principal de la metaheurística para disminuir gradualmente el valor de dicho factor *mutacion* en cada iteración multiplicándolo por un *factor_ajuste*.

De esta forma, el código de la metaheurística *Leaders and Followers* con dicha mejora de diseño/comportamiento sería el siguiente:

```cpp
vector<double> LeadersAndFollowersModificado(const vector<Ejemplo>& entrenamiento) {

  int num_iteraciones = 0;

  PoblacionLF leaders;
  PoblacionLF followers;

  vector<double> w(entrenamiento[0].num_caracts);

  int tamanio_poblacion = w.size();

  leaders = inicializarPoblacionLF(entrenamiento);
  followers = inicializarPoblacionLF(entrenamiento);

  double mutacion_inicial = 0.1;
  double mutacion_minima = 0.01;
  double factor_ajuste = 0.9;

  double mutacion = mutacion_inicial;

  while(num_iteraciones < MAX_ITER_LEADERS_AND_FOLLOWERS) {

    for(int i = 0; i < tamanio_poblacion; ++i) {
      auto it_l = leaders.begin();
      advance(it_l, i);
      LF leader = *it_l;

      auto it_f = followers.rbegin();
      advance(it_f, i);
      LF follower = *it_f;

      LF trial = createTrialModificado(leader, follower, entrenamiento, mutacion);

      if(trial.fitness > follower.fitness)
        follower = trial;
    }

    if( calcularMediana(leaders) > calcularMediana(followers) ) {
      leaders = mergePopulations(leaders, followers);
      followers = inicializarPoblacionLF(entrenamiento);
    }

    mutacion *= factor_ajuste;

    if(mutacion < mutacion_minima)
      mutacion = mutacion_minima;

    num_iteraciones++;
  }

  w = leaders.rbegin()->w;

  return w;
}
```

6. 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, iomanip, set...* Además, hacemos uso también del archivo ***random.hpp*** proporcionado por el profesor de prácticas 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.

- ***pAlternativa.h/pAlternativa.cpp*** → Contienen los algoritmos implementados en esta práctica final alternativa al examen (Leaders and Followers, Leaders and Followers hibridado con Búsqueda Local y Leaders and Followers Modificado), así como una función para ejecutar cada 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 *practicaAlt_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 ***./practicaAlt_MH {semilla}.***

7. Resultados Obtenidos

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).

A continuación, se muestran las tablas de los resultados obtenidos para cada uno de los algoritmos de esta práctica 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 LAF en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	*%red*	*Fit.*	T	*% clas*	*%red*	*Fit.*	T	*% clas*	*%red*	*Fit.*	T
Partición 1	68,18	12,50	57,05	1189,46	73,44	18,06	62,36	29566,10	75,71	25,00	65,57	12679,30
Partición 2	66,23	25,00	57,99	1133,20	75,00	16,67	63,33	15355,79	85,71	15,91	71,75	6743,85
Partición 3	66,23	37,50	60,49	1194,59	81,25	19,44	68,89	29470,93	92,86	25,00	79,29	12819,39
Partición 4	68,18	12,50	57,05	1156,71	73,44	8,33	60,42	15273,54	80,00	27,27	69,45	7004,80
Partición 5	67,11	37,50	61,18	1150,66	87,50	20,83	74,17	29521,04	85,51	20,45	72,50	6697,56
Media	67,19	25,00	58,75	1164,92	78,12	16,67	65,83	23837,48	83,96	22,73	71,71	9188,98

Tabla 5. 2 : Resultados obtenidos por el algoritmo LAF-BL en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	*%red*	*Fit.*	T	*% clas*	*%red*	*Fit.*	T	*% clas*	*%red*	*Fit.*	T
Partición 1	68,18	12,50	57,05	1047,91	76,56	25,00	66,25	94524,12	80,00	15,91	67,18	25050,03
Partición 2	70,13	25,00	61,10	964,14	81,25	16,67	68,33	95176,00	85,71	15,91	71,75	24536,92
Partición 3	66,88	37,50	61,01	980,73	82,81	15,28	69,31	94296,82	91,43	13,64	75,87	26127,99
Partición 4	72,73	37,50	65,68	927,29	75,00	18,06	63,61	96106,07	80,00	15,91	67,18	24974,86
Partición 5	65,79	25,00	57,63	916,16	82,81	13,89	69,03	97034,01	79,71	25,00	68,77	24810,50
Media	68,74	27,50	60,49	967,25	79,69	17,78	67,31	95427,40	83,37	17,27	70,15	25100,08

Tabla 5. 3 : Resultados obtenidos por el algoritmo LAF-MOD en el problema del APC

	Diabetes				Ozone				Spectf-heart			
	% clas	*%red*	*Fit.*	T	*% clas*	*%red*	*Fit.*	T	*% clas*	*%red*	*Fit.*	T
Partición 1	68,18	12,50	57,05	1382,32	82,81	19,44	70,14	33040,16	81,43	18,18	68,78	13715,32
Partición 2	64,94	12,50	54,45	1220,78	81,25	23,61	69,72	31532,66	88,57	15,91	74,04	7452,00
Partición 3	62,99	50,00	60,39	1225,75	78,12	20,83	66,67	32026,38	88,57	27,27	76,31	13674,21
Partición 4	64,29	62,50	63,93	2260,72	75,00	13,89	62,78	17316,35	87,14	13,64	72,44	7506,73
Partición 5	65,79	62,50	65,13	2243,07	82,81	13,89	69,03	31301,63	82,61	15,91	69,27	13783,93
Media	65,24	40,00	60,19	1666,53	80,00	18,33	67,67	29043,44	85,66	18,18	72,17	11226,44

Tabla 5. 4 : Resultados globales en el problema del APC

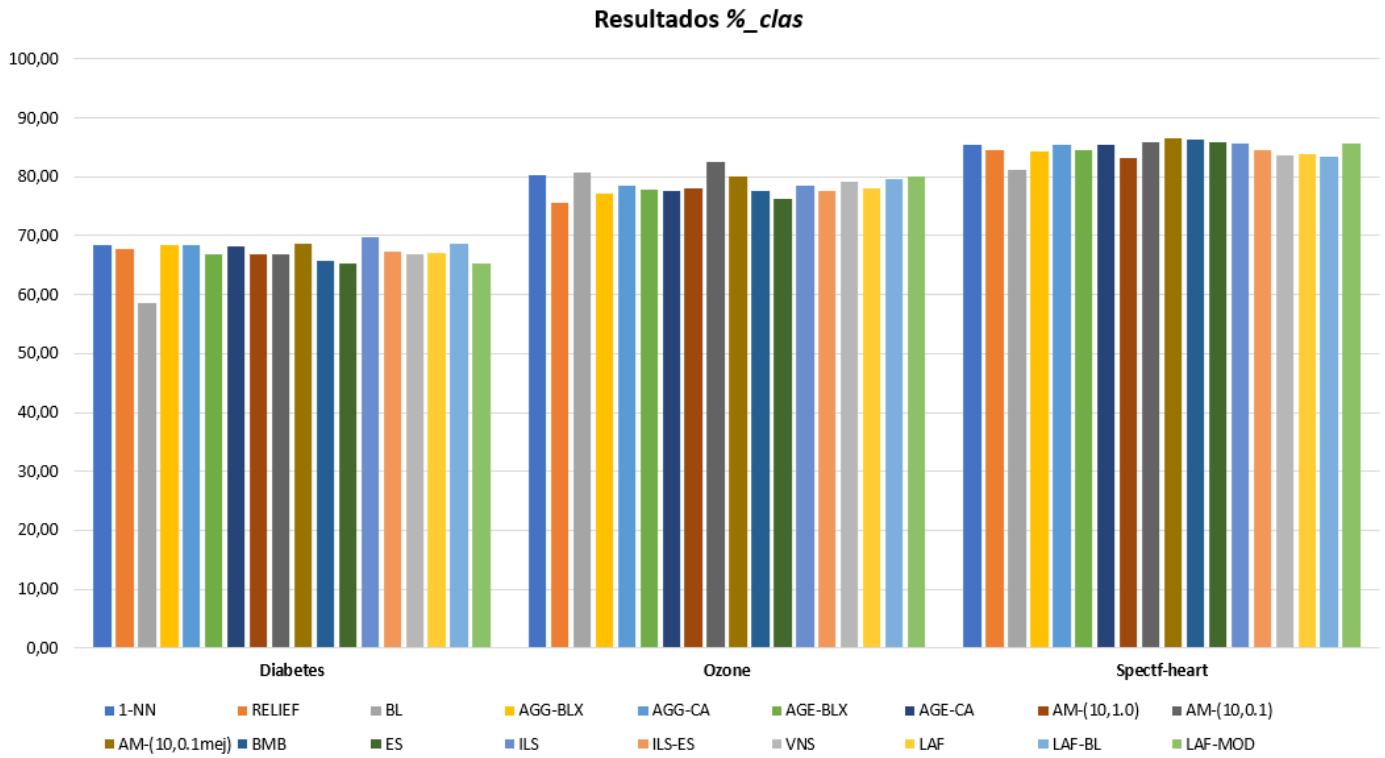
	Diabetes				Ozone				Spectf-heart			
	% clas	*%red*	*Fit.*	T	*% clas*	*%red*	*Fit.*	T	*% clas*	*%red*	*Fit.*	T
1-NN	68,36	0,00	54,69	1,09	80,31	0,00	64,25	1,26	85,38	0,00	68,31	1,06
RELIEF	67,71	5,00	55,17	9,78	75,62	6,94	61,89	5,29	84,51	24,09	72,43	4,00
BL	58,61	45,00	55,89	1028,35	80,62	12,50	67,00	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**	16362,62	79,06	51,39	73,53	72268,20	83,66	71,36	**81,20**	46265,62
LAF	67,19	25,00	**58,75**	1164,92	78,12	16,67	**65,83**	23837,48	83,96	22,73	**71,71**	9188,98
LAF-BL	68,74	27,50	**60,49**	967,25	79,69	17,78	**67,31**	95427,40	83,37	17,27	**70,15**	25100,08
LAF-MOD	65,24	40,00	**60,19**	1666,53	80,00	18,33	**67,67**	29043,44	85,66	18,18	**72,17**	11226,44

En la tabla 5.4, donde se recogen los resultados globales de la ejecución de los diferentes algoritmos (todos los que se han desarrollado en las diferentes prácticas) 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**. También se destacan en **negro** los **valores de fitness de los algoritmos de la metaheurística *Leaders and Followers***.

8. Análisis de los Resultados Obtenidos

Vamos a comparar los diferentes resultados obtenidos por los diferentes algoritmos de la metaheurística *Leaders and Followers* para cada conjunto de datos con el resto de algoritmos. 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

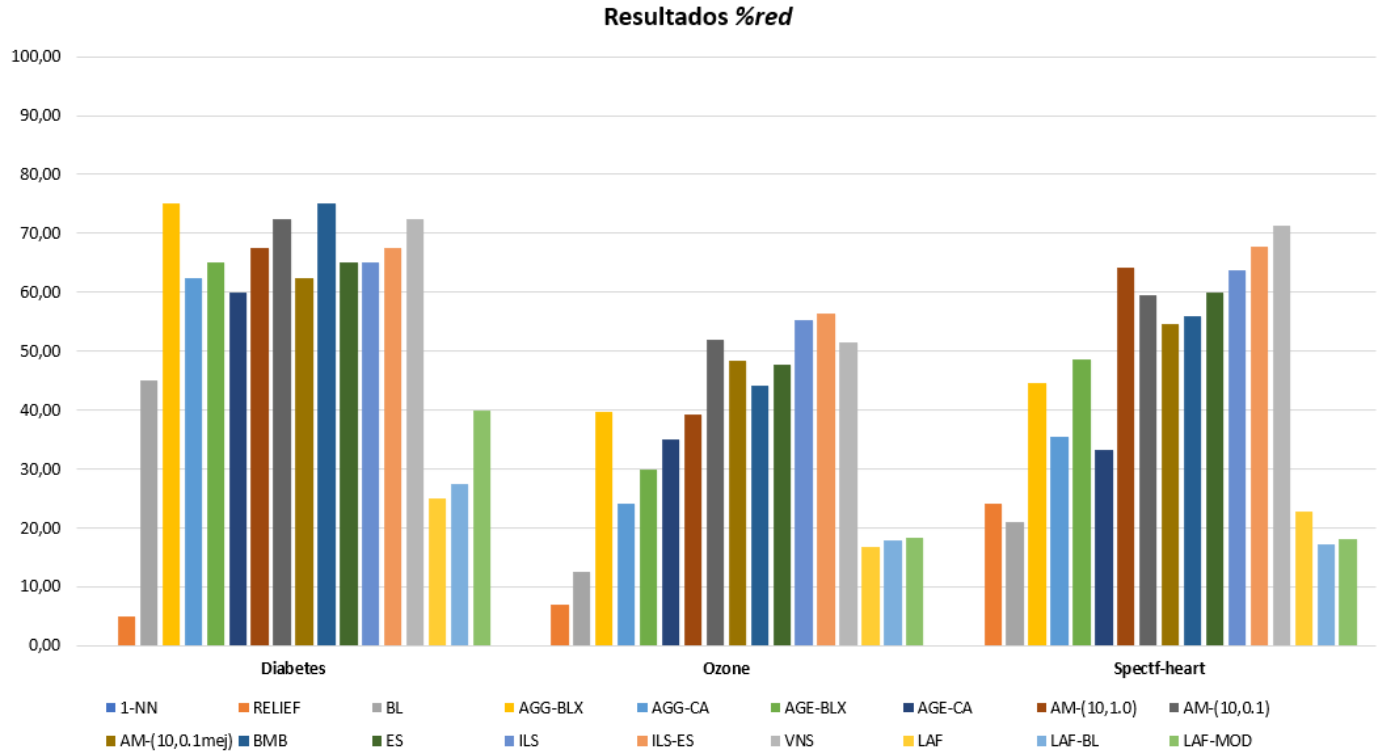


- Observamos que, entre los algoritmos desarrollados sobre la metaheurística *Leaders and Followers*, **el algoritmo LAF-MOD es el que obtiene mayor tasa de clasificación en general, seguido del algoritmo LAF-BL.**

- Con respecto a los **algoritmos implementados en las prácticas**, vemos que, en general, **LAF, LAF-BL y LAF-MOD los igualan más o menos en tasa de clasificación** excepto a los algoritmos meméticos AM-(10,1.0), AM-(10,0.1) y AM-(10,0.1mej), los cuales son los más destacados.

- 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 **AM-(10,0.1mej)**.

Tasa de Reducción

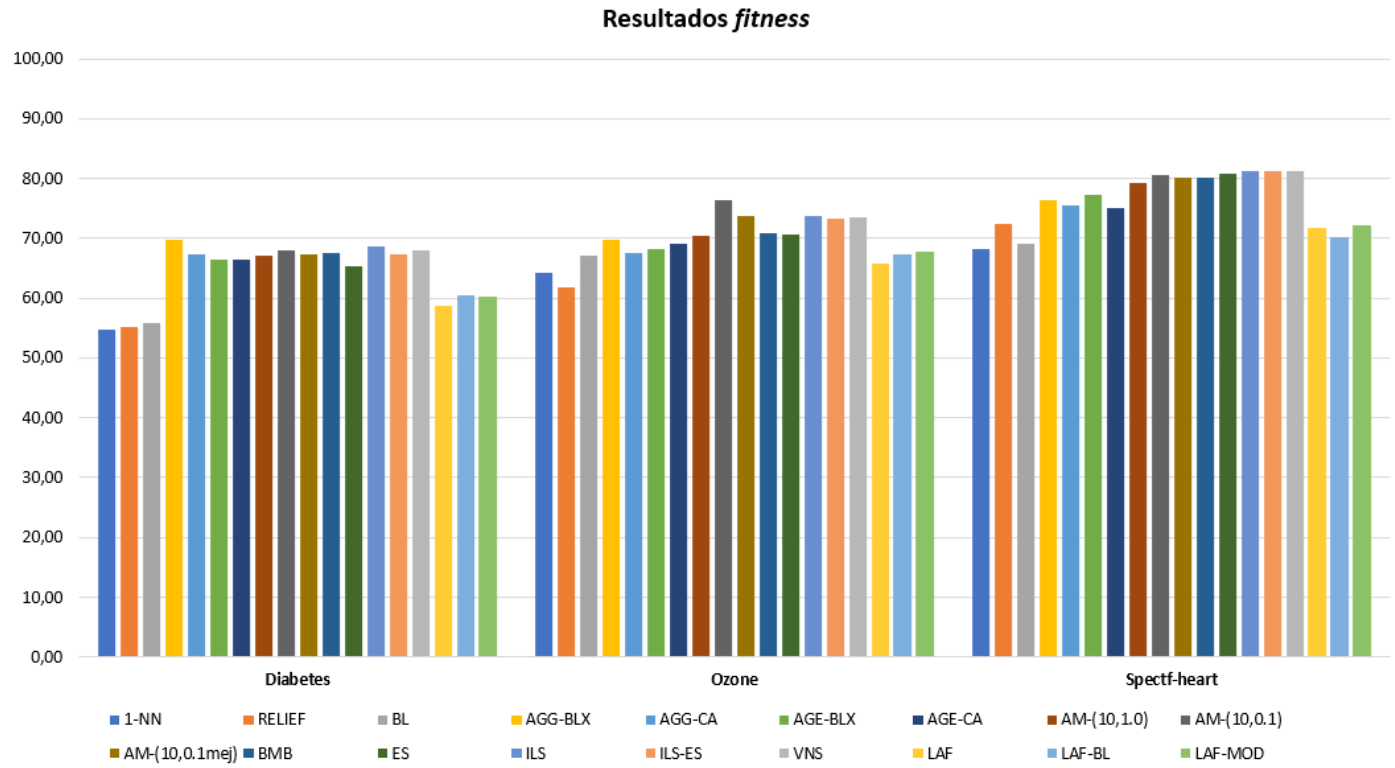


- Observamos que, entre los algoritmos desarrollados sobre la metaheurística *Leaders and Followers*, **el algoritmo LAF-MOD es el que obtiene mayor tasa de reducción en el conjunto de datos diabetes, en el conjunto de datos ozone los tres algoritmos tienen una tasa de reducción muy similar y en el conjunto de datos spectf-heart el algoritmo LAF es el que tiene mayor tasa de reducción.**

- Con respecto a los **algoritmos implementados en las prácticas**, vemos que, en general, **LAF, LAF-BL y LAF-MOD tienen una tasa de reducción mucho más inferior,** excepto para los algoritmos de la práctica 1 (1-NN, Greedy RELIEF y BL), siendo como mínimo iguales.

- En función de los conjuntos de datos, para el conjunto de datos ***diabetes***, los algoritmos que empatan en una mayor tasa de reducción son los algoritmos **AGG-BLX** y **BMB**, para ***ozone*** es **ILS-ES** y para ***spectf-heart*** es **VNS.**

Fitness

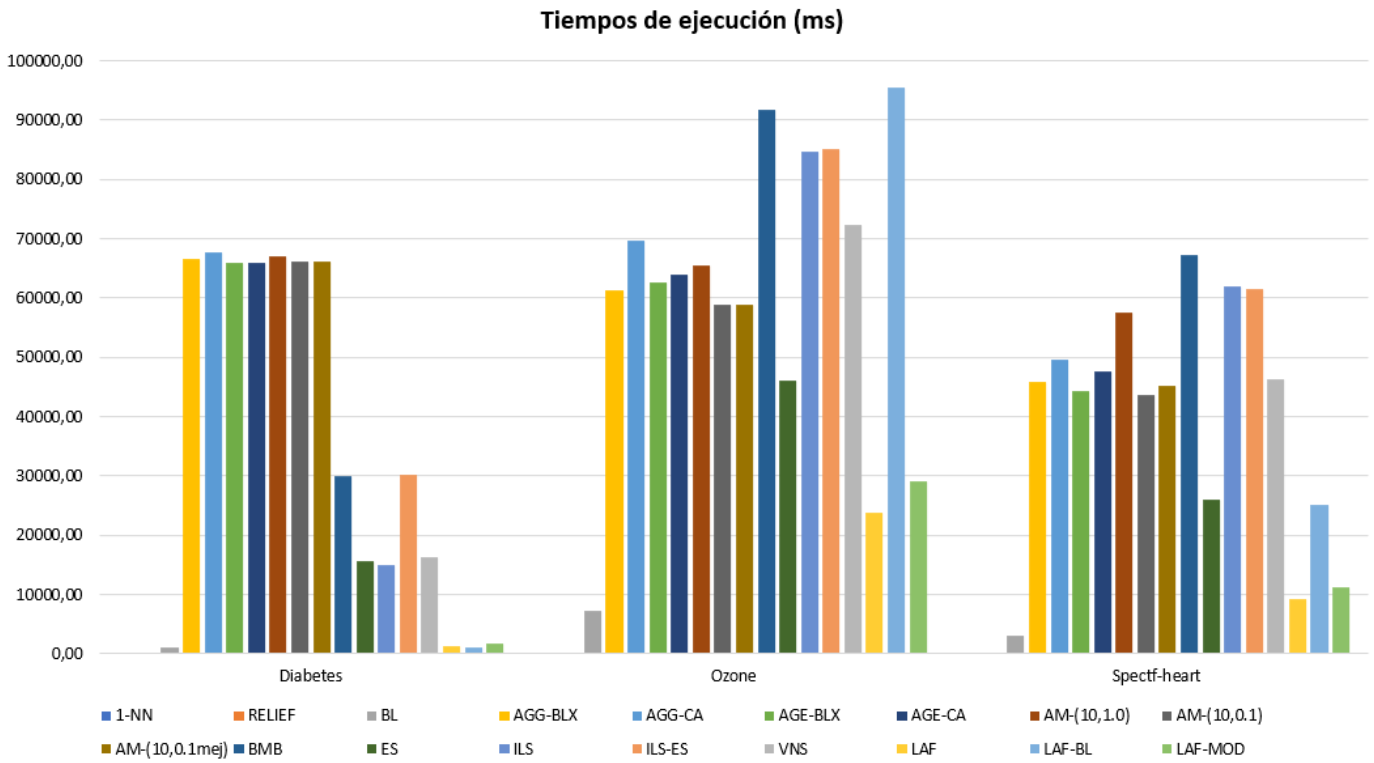


- Observamos que, entre los algoritmos desarrollados sobre la metaheurística *Leaders and Followers*, **el algoritmo LAF-MOD es el que obtiene mayor fitness en general, seguido del algoritmo LAF-BL.**

- Con respecto a los **algoritmos implementados en las prácticas**, vemos que, en general, **LAF, LAF-BL y LAF-MOD tienen un fitness mucho más inferior,** excepto para los algoritmos de la práctica 1 (1-NN, Greedy RELIEF y BL), siendo como mínimo iguales.

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

Tiempos de Ejecución



- Observamos que, entre los algoritmos desarrollados sobre la metaheurística *Leaders and Followers*, **el algoritmo LAF-BL es el que tiene un mayor tiempo de ejecución en general, y los algoritmos LAF y LAF-MOD tienen un tiempo de ejecución similar.**

- Con respecto a los **algoritmos implementados en las prácticas**, vemos que, en general, **LAF, LAF-BL y LAF-MOD tienen un tiempo de ejecución mucho menor,** excepto el algoritmo LAF-BL en el conjunto de datos *ozone*, siendo este el mayor de entre todos los algoritmos.

- En función de los conjuntos de datos, para el conjunto de datos ***diabetes***, el algoritmo que tiene un mayor tiempo de ejecución es el algoritmo **AGG-CA**, para ***ozone*** es **LAF-BL** y para ***spectf-heart*** es **BMB.**

9. Conclusiones

En primer lugar, cabe destacar que **los algoritmos basados en la metaheurística Leaders and Followers no generan muy buenos resultados para el problema APC en comparación con los otros algoritmos desarrollados en las prácticas.** Sí que se observa que, **como mínimo, estos algoritmos igualan en general a los algoritmos desarrollados en la práctica 1 (1-NN, Greedy RELIEF y Búsqueda Local)**. También se observa que se produce una **mejora en los resultados al hibridar el algoritmo base de la metaheurística *Leaders and Followers* con la Búsqueda Local**, y que se consiguen **resultados aún mejores con la propuesta de mejora del diseño/comportamiento de la metaheurística**.

En conclusión, considerando todos los algoritmos desarrollados, 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 el código de los algoritmos basados en la metaheurística *Leaders and Followers* es bastante sencillo e intuitivo. El objetivo de dicha metaheurística era intentar corregir el principal problema que planteaba la búsqueda local: el estancamiento prematuro en óptimos locales. Tras realizar un estudio de la convergencia de los algoritmos basados en dicha metaheurística, podemos concluir con seguridad, que se ha logrado cumplir dicho objetivo. Es probable que, a costa de un aumento considerable en el tiempo de ejecución, se logren obtener mejores resultados para el problema APC con dichos algoritmos. **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**.