Práctica 2:

Búsquedas basadas en Trayectorias Simples,

Trayectorias Múltiples

y Búsqueda Evolutiva.

Curso 2016/2017

Problema: APC

Algoritmos:

1-NN
RELIEF
ES
ILS
DE/rand/1
DE/current-to-best/1
BL
AGG-BLX
AM-(10,0.1mej)

Francisco Javier Caracuel Beltrán

76440940-A

caracuel@correo.ugr.es

Grupo 3 (Lunes 17:30-19:30)

Índice

1. Descripción del problema

El problema elegido, en este caso, es el "***Problema del Aprendizaje de Pesos en Características***", consistente en optimizar el rendimiento de un clasificador basado en el algoritmo *1-NN* (vecino más cercano), a partir de una serie de pesos asociados a las características del problema.

Este clasificador hace uso de la distancia entre el ejemplo que se quiere clasificar y todos los ejemplos de la muestra. De este modo, se le asignará la etiqueta que tenga el ejemplo cuya distancia sea menor. Para el cálculo de esta distancia, se tienen en cuenta los pesos asociados a las características que se han comentado anteriormente.

La práctica consiste, principalmente, en generar multitud de conjuntos de pesos a través de distintas técnicas y obtener, finalmente, un conjunto de pesos que consigue la tasa de acierto más elevada.

Las distintas técnicas para generar esos conjuntos de pesos y que se explican en próximas secciones son:

- Algoritmo Aleatorio.
- Algoritmo Relief.
- Algoritmo Enfriamiento Simulado.
- Algoritmo Búsqueda Local Iterativa.
- Algoritmo Evolución Diferencial con cruce aleatorio y cruce aleatorio + mejor.
- Algoritmo Aleatorio + Búsqueda Local.
- Algoritmo Genético Generacional con cruce BLX.
- Algoritmo Memético (Algoritmo Genético + Búsqueda Local).

2. Descripción de los algoritmos comunes

A continuación, se detallan todas las consideraciones comunes a los distintos algoritmos.

a) Descripción del esquema de representación de soluciones:

La solución W vendrá dada por un contenedor de números reales normalizados en [0,1] y será de tamaño N.

Se entiende N como el número de atributos con el que se codifican los ejemplos del problema que se esté resolviendo.

$$W = (w_1, w_2, w_3, \ldots, w_N) \qquad \text{donde } w_i \in [0,1]$$

w_1	w_2	...	w_{n-1}	w_n

Un 1 en la posición w_i indica que la característica $_i$ se considera completamente en el cálculo de la distancia.

Un 0 en la posición w_i indica que la característica $_i$ no se considera en el cálculo de la distancia.

Si existe algún valor intermedio, gradúa el peso que tiene éste en esa característica, ponderando su importancia.

b) Descripción en pseudocódigo de la función objetivo:

```
well = 0
notUsed = 0
for i in numInstances
{
    if label[i] == NN1(data[i])
    {
        well++;
    }
}
for i in numAttributes
{
    if w[i] < minReduction
    {
        notUsed++
    }
}
return well/numInstances*weightWell + notUsed/numAttributes*(1-weightWell)
```

c) Descripción en pseudocódigo del operador de generación de vecino/mutación:

```
w = current_solution
posMutate = randomInt(1, numAttributes)
w[posMutate] = w[posMutate] + NormalDistribution(mean, standard_deviation)
truncate(w[posMutate], 0, 1)
return w
```

*Usado en la Búsqueda Local, en los Algoritmos Genéticos, en el Enfriamiento Simulado y en la Búsqueda Local Iterativa.

d) Descripción en pseudocódigo de la generación de soluciones aleatorias:

```
wRandom = container(numAttributes)
for i in numAttributes
{
    wRandom[i] = randomFloat(0, 1)
}
return wRandom
```

e) Descripción en pseudocódigo del mecanismo de selección de los AGs:

```
selection = container(numParents)
population = currentPopulation
for i in numParents
{
    pos1 = randomInt(1, numPopulation)
    pos2
    while(pos1 == (pos2 = randomInt(1, numPopulation))){
    }
    if success(population[pos1]) >= success(population[pos2])
        selection[i] = population[pos1]
    else
        selection[i] = population[pos2]
}
return selection
```

f) Descripción en pseudocódigo del operador de cruce del AG:

- Cruce BLX:

```
w1 = chromosome1
w2 = chromosome2
for i in numAttributes
{
    cMax = max(w1[i], w2[i])
    cMin = min(w1[i], w2[i])
    I = cMax - cMin
    w1[i] = randomFloat(cMin - I*crossAlpha, cMax + I*crossAlpha)
    w2[i] = randomFloat(cMin - I*crossAlpha, cMax + I*crossAlpha)
    truncate(w1[i], 0, 1)
    truncate(w2[i], 0, 1)
}
return w1, w2
```

g) Algoritmo Búsqueda Local:

```
// Contador del número de evaluaciones
iEval = 0
// Contador del número de vecinos sin mejorar
iNeig = 0
// Mejor acierto
pF = 0
w = current_solution
while iEval<numEvalMax && iNeig<numNeig*numAttributes
{
    wNew = w
    wNew = mutate(wNew)
    if evaluate(wNew) > pF
    {
        pF = evaluate(wNew)
        w = wNew
        iNeig = 0
    }
    iEval = iEval + 1
    iNeig = iNeig + 1
}
```

Descripción en lenguaje natural:

- *iEval* es el contador que se utiliza para controlar el número de iteraciones que puede hacer.
- *numEvalMax* es el número máximo de iteraciones que tiene permitido hacer. En el caso de esta práctica, este límite es 15.000.
- *iNeig* es el contador que se utiliza para controlar el número de iteraciones que transcurren sin que una mutación mejore a la mejor solución actual.
- *numNeig* es el número que, multiplicado al número de atributos del problema que se esté tratando, ofrecerá el número máximo de iteraciones que tiene permitido hacer sin que un conjunto de mutaciones mejore a la mejor solución actual. En el caso de esta práctica es 20. Si el problema a tratar tuviera 60 atributos, el límite máximo de iteraciones que puede realizar sin mejorar sería 1.200.
- *pF* contiene la tasa de acierto de la mejor solución obtenida.

El algoritmo de Búsqueda Local realiza una serie de mutaciones partiendo de la solución que recibe.

Estará realizando mutaciones mientras que no supere el límite máximo de mutaciones (o límite máximo de evaluaciones de la mutación creada) permitidas y, además, mientras consiga una mejora de la mejor solución obtenida hasta el momento cumpliendo las condiciones descritas anteriormente.

Por cada mutación hace una llamada a la función evaluación y comprueba que la tasa de acierto de la mutación, sea mayor que la tasa de acierto de la mejor solución actual. En el caso de que la mutación mejore la tasa de acierto, se actualizarán los datos y, a partir de ese momento, la mejor solución es la mutación.

Finalmente, el algoritmo termina devolviendo la mejor solución encontrada, ya sea la mutación o la solución inicial, si ninguna mutación la ha mejorado.

3. Descripción de los algoritmos específicos

A continuación, se detalla la descripción en pseudocódigo de todas aquellas operaciones relevantes de cada algoritmo.

a) Algoritmo Genético Generacional:

```
population = currentPopulation
successPopulation = currentSuccessPopulation
numCrosses = round(probabilityOfCrossing*size(population))
numMutations = round(probabilityOfMutation*size(population)*numAttributes)
iEval = 0
while iEval < numEval
{
    newPopulation = selection()
    newSuccessPopulation = container(size(newPopulation))
    for i in numCrosses
    {
        newPopulation[i] = cross(newPopulation[i])
    }
    for i in numMutations
    {
        newPopulation[i] = mutate(newPopulation[i])
    }
    for i in size(newPopulation)
    {
        equals = false
        j = 0
        while not equals && j<size(population)
        {
            if newPopulation[i] == population[j]
                equals = true
            j++
        }
        if equals
            newSuccessPopulation[i] = successPopulation[j-1]
        else
            newSuccessPopulation[i] = evaluate(newPopulation[i])
    }
    if first(population) > first(newPopulation)
        update(first(population), last(newPopulation))
    population = newPopulation
    successPopulation = newSuccessPopulation
    iEval = iEval + size(population)
}
return bestSolution(), bestSuccess()
```

Descripción en lenguaje natural:

- *iEval* es el número de evaluaciones que se realizan. Por cada iteración se realizan tantas evaluaciones como nuevos cromosomas hay en la población. En el caso de la práctica se tiene una población con 30 cromosomas.
- *numEval* es el número máximo de evaluaciones permitidas. En el caso de la práctica se pueden hacer 15.000 evaluaciones, lo que dan lugar a un total de 500 generaciones de cromosomas nuevas.
- *population* es un contenedor donde se encuentran todos los cromosomas actuales con los que dispone el algoritmo.
- *successPopulation* es un contenedor del tamaño de la población que contiene la tasa de acierto correspondiente a cada cromosoma.
- *numCrosses* es el número de cruces que se pueden realizar en cada iteración.
- *numMutations* es el número de mutaciones que se pueden realizar en cada iteración.

El algoritmo comienza con un conjunto de cromosomas aleatorios.

En cada generación hay que cruzar y mutar una serie de cromosomas en base a una probabilidad. Calcular qué cromosomas se cruzan o mutan directamente es muy ineficiente y por eso se calcula el número exacto de cromosomas que se van a cruzar o mutar. Como la selección es aleatoria, ya se tiene en cuenta esa aleatoriedad para realizar las operaciones.

En mi práctica en particular, el método que ejecuta el Algoritmo Genético Generacional, dispone de un valor que indica el cruce que debe realizar. De este modo, cuando se realiza la llamada al cruce, será el cruce *BLX* o el *cruce aritmético* dependiendo del valor que reciba. También se tiene en cuenta el tipo de cruce que debe realizar para la selección, ya que, si es el cruce *BLX*, la selección devuelve tantos padres como tamaño tenga la población. Si es el *cruce aritmético*, la selección devolverá el doble de padres del tamaño de la población, al generar en este cruce, un solo hijo por cada dos padres.

Otro aspecto a tener en cuenta es el número de evaluaciones. Se ha comprobado que se tarda menos tiempo de ejecución en comparar si un cromosoma es igual que otro ya existente en la población anterior, que en evaluar uno de ellos. Esto hace que cuando ya se han cruzado y mutado los correspondientes cromosomas, se compare si ya existía anteriormente. En el caso de que existiera, se le aplica la tasa de acierto que tuviera el cromosoma que es igual. Si no, se realizará la evaluación.

Para conservar el elitismo de la población, antes de terminar el tratamiento de la generación, se comprueba si el mejor de la población anterior es mejor que el mejor de la población generada. En caso de que así sea, se elimina el peor de la nueva generación y se añade el mejor de la anterior.

El algoritmo termina devolviendo el mejor cromosoma generado y su tasa de acierto.

b) Algoritmo Memético:

```
    .
    .
    .
        mutate(newPopulation[randomInt(1, size(population))])
    }
    posToApplyLocalSearch = generatePosToApplyLocalSearch()
    evaluations = LocalSearch(posToApplyLocalSearch, newPopulation,
                              newSuccessPopulation)
    iEval = iEval + evaluations
    for i in size(newPopulation)
    {
        if i not posToApplyLocalSearch
            evaluate(newPopulation[i])
    }
    if first(population) > first(newPopulation)
    .
    .
    .
```

Los *Algoritmos Memético*s son *Algoritmos Genéticos* a los que se le aplica una *Búsqueda Local*. En este caso, el esquema del *Algoritmo Memético* es similar al del *Algoritmo Genético Generacional*, salvo que después de mutar los cromosomas y antes de realizar la operación de conservación del elitismo de la población, se aplica la *Búsqueda Local* a una serie de cromosomas, evaluando al finalizar los restantes a los que no se les ha aplicado la *Búsqueda Local*.

A qué serie de cromosomas se le aplica la Búsqueda Local, depende del tipo de *Algoritmo Memético* que se quiera realizar.

A grandes rasgos y suponiendo que se dispone de un contenedor que indique la posición del cromosoma al que se le va a aplicar la *Búsqueda Local* (posLS), el pseudocódigo correspondiente al método *LocalSearch(newPopulation, newSuccessPopulation)* del *Algoritmo Memético* sería:

```
numEval = 0
for i in size(newPopulation)
{
    if posLS[i] == true
    {
        newPopulation[i] = getSolutionLocalSearch(newPopulation[i])
        newSuccessPopulation[i] = getSuccessLocalSearch(newPopulation[i])
        numEval = numEval + getEvalLocalSearch(newPopulation[i])
    }
}
return numEval
```

Cabe destacar, que el número de evaluaciones permitidas es 15.000, por lo que también se deben contar las evaluaciones que se realizan en la *Búsqueda Local*, aumentando el contador de éstas cuando se termina la llamada al método.

c) Algoritmo Enfriamiento Simulado:

```
w = generateRandomSolution()
wEval = evaluate(w)
bestW = w
bestEval = wEval
t = initTemperature(wEval)
numEval = 1
success = inf
while numEval < maxEval && success != 0
{
    iNeighbors = 0
    success = 0
    while iNeighbors < maxNeighbors && success < maxSuccess
    {
        wAux = mutate(w)
        wEvalAux = evaluate(wAux)
        numEval++
        differenceEval = wEvalAux - wEval
        if differenceEval > 0 || randomInt(0,1) >= exp(differenceEval/t)
        {
            w = wAux
            wEval = wEvalAux
            success++
            if(bestW < wEval)
            {
                bestW = w
                bestEval = wEval
            }
        }
        iNeighbors++
    }
    t = coolTemperature(t)
}
return bestW, bestEval
```

El *Algoritmo Enfriamiento Simulado* solo trabaja con tres soluciones a la vez. Inicialmente genera una solución aleatoria, la segunda es utilizada para cálculos intermedios en los que comprueba si la mutación generada es mejor que la original y una tercera solución que es la mejor solución encontrada hasta el momento.

El algoritmo se basa en el uso de una variable llamada temperatura, la cual, en cada ciclo, disminuye su valor hasta un mínimo.

En cada ciclo se generan una serie de soluciones mediante mutaciones de la solución actual. Esta mutación puede mejorarla o empeorarla y podrá sustituir a la solución actual en dos casos: la solución mutada mejora a la original; mediante una probabilidad de sustituirla (cuando es peor).

En el cálculo de esta probabilidad es donde entra en juego la temperatura. En cada ciclo, la temperatura va disminuyendo y se aprovechan las características de la función exponencial.

El valor de la exponencial en 0 es 1, si crece tiende a infinito y si decrece tiende a 0, por lo que es este intervalo de valores $(-\infty, 0]$ el que interesa para calcular la probabilidad.

Teniendo la temperatura, se puede dividir la diferencia de la evaluación de la solución mutada y la original entre la propia temperatura, ofreciendo dos características:

- Cuanto mayor sea la temperatura, el resultado obtenido del cociente entre ambos será menor. A medida que la temperatura disminuya, el resultado será mayor. Teniendo en cuenta que, para hacer estos cálculos, la diferencia entre evaluaciones debe ser negativa, el resultado de la exponencial de este valor calculado tenderá a 0, por lo que a temperaturas altas, existe mucha más probabilidad de aceptar una solución peor que cuando la temperatura sea baja (ya que el resultado de la exponencial del cociente entre ambos tenderá a 1).
- Cuanto menor sea la diferencia entre soluciones, más probabilidad de aceptar soluciones peores habrá y cuanto mayor sea la diferencia, se tendrá menos probabilidad. La deducción es la misma que en el caso anterior, pero, al contrario. Si la diferencia entre evaluaciones es mayor, el resultado del cociente entre ambos será mayor, lo que hace que al ser la exponencial estrictamente creciente, se tenga menos probabilidad de aceptar soluciones peores.

Al tener la posibilidad de aceptar soluciones peores, es necesario guardar cuál ha sido la mejor solución generada en todo el proceso, que será la que finalmente se devuelva.

Se calcula la temperatura inicial como sigue:

```
initialTemperature = (mu*eval)/-ln(phi)
```

Cálculo del enfriamiento de la temperatura:

```
beta = (initialTemperature-finalTemperature) / M*initialTemperature*finalTemperature
temperature = temperature / (1 + (beta*temperature))
```

d) Algoritmo Búsqueda Local Iterativa:

```
t = calculateAttributesToChange()
w = generateRandomSolution()
wEval = LocalSearch(w)
for i in maxIterations
{
    wAux = mutate(w, t)
    evalAux = LocalSearch(wAux)
    if(evalAux > wEval)
    {
        w = wAux
        wEval = evalAux
    }
}
return w, wEval
```

La *Búsqueda Local Iterativa* tiene un mecanismo muy sencillo. Comienza generando una solución aleatoria a la que le aplica una *Búsqueda Local*, se realizan una serie de iteraciones previamente establecidas, en las que dentro de cada una:

- Realiza una mutación fuerte.
- Evalúa la solución.
- Compara la solución con la actual y guarda la mejor.

Finalmente devuelve la mejor solución obtenida. No hay problema con el elitismo porque siempre trabaja sobre la mejor solución.

La mutación realizada es muy parecida a la normal. En ella se mutan *t* características de la solución siguiendo una distribución normal establecida:

```
w = currentSolution
posMutate = generateRandomPositionsWithoutRepetitions(t, 1, numAttributes)
for i in posMutate
{
    w[i] = w[i] + NormalDistribution(mean, standardDeviation)
    truncate(w[i], 0, 1)
}
return w
```

e) Algoritmo Evolución Diferencial:

```
population = generateRandomPopulation()
pSuccess = evaluate(population)
numEval = size(population)
numParents = 2|3 // Dependiendo del tipo de cruce
while numEval < maxEval
{
    for i in size(population)
    {
        parents = getParents(population, numParents) // Dependiendo del cruce
                                                     // elige tres padres o
                                                     // dos padres y el mejor
        posW = randomInt(1, size(population)) // Se valida que no sea
                                              // un padre elegido
        w = population[posW]
        wEval = pSuccess[posW]
        wAux
        for j in numAttributes
        {
            if randomInt(0,1) <= probabilityOfCrossing
                wAux[j] = cross(w[j], parents[1][j], parents[2][j],
                                                     parents[3][j])
        }
        wEvalAux = evaluate(wAux)
        numEval++
        if wEval < wEvalAux
        {
            population[posW] = wAux
            pSuccess[posW] = wEvalAux
        }
    }
    sortPopulation()
}
return first(population), first(pSuccess)
```

El algoritmo de *Evolución Diferencial* comienza generando una población aleatoria de un tamaño determinado, la evalúa e itera hasta que consuma el máximo de evaluaciones máximas que tiene permitidas.

En cada iteración, elige tantas soluciones aleatorias como tamaño tiene la población y, para cada una de ellas, hace un cruce de ésta con una serie de soluciones de la misma población.

Al terminar cada cruce, comprueba cuál de los dos es mejor y conserva el mejor.

Finalmente devuelve la mejor solución y como conserva el elitismo, no hay que preocuparse de perder a la mejor.

El proceso de mutación y recombinación se ha realizado en el mismo paso, que se explica a continuación.

Se han implementado dos tipos de cruces:

- Aleatorio:

```
p1 = parent1
p2 = parent2
p3 = parent3
return truncate(p1 + f*(p2-p3))
```

Al realizar los cálculos, el resultado puede no ser válido, por lo que se debe controlar que esté en el intervalo [0,1].

- Mejor + aleatorio:

```
w = currentAttributeSolution
best = bestAttributeSolution
p1 = parent1
p2 = parent2
return truncate(w + f*(best-w) + f*(p1-p2))
```

Al realizar los cálculos, el resultado puede no ser válido, por lo que se debe controlar que esté en el intervalo [0,1].

4. Descripción en pseudocódigo del algoritmo de comparación

a) 1-NN:

Para hacer uso del algoritmo de comparación *1-NN* primero se debe definir el método que se encarga de calcular la *distancia euclídea* entre dos ejemplos de la muestra:

```
e1 = example1
e2 = example2
w = currentSolution
res = 0
for i in numAttributes
{
    if w[i] > minReduction
    {
        sub = e1[i] - e2[i]
        res = res + (w[i] * sub* sub)
    }
}
res = sqrt(res)
return res
```

A continuación, el algoritmo de comparación *1-NN*:

```
data = container_data
label = container_class
e = current_example
labelRes = first(label)
dMin = de(first(data), e)
for i in size(data)
{
    dAux = de(data[i], e)
    if dAux<dMin
    {
        dMin = dAux
        labelRes = label[i]
    }
}
return labelRes, dMin
```

Descripción en lenguaje natural:

- *data* es el conjunto de ejemplos de los que se compone la muestra.
- *label* es el conjunto de clases (etiquetas) que tiene cada ejemplo de la muestra.

- *e* es el ejemplo actual sobre el que se quiere calcular la distancia euclídea.
- *labelRes* contiene la etiqueta del ejemplo de la muestra cuya distancia es menor con respecto al ejemplo que se evalúa.
- *dMin* es la distancia mínima que se ha encontrado hasta el momento.

El algoritmo *1-NN* consiste en obtener la clase de un ejemplo, ya que, en principio, sería desconocida.

Devuelve una etiqueta o clase comparando con todos los ejemplos que se encuentran en la muestra. Lógicamente, el ejemplo que se quiere comprobar no puede pertenecer a la muestra porque el resultado de calcular la distancia con él mismo, sería 0.

En el caso de querer comprobar la clase de un ejemplo perteneciente a la muestra, habría que aplicar el algoritmo *1-NN leave one out*, que compara con todos los ejemplos menos consigo mismo.

El algoritmo comienza obteniendo la distancia euclídea del ejemplo desconocido con el primer ejemplo de la muestra y guardando, a su vez, la clase a la que pertenece éste.

A continuación, se recorren todos los ejemplos restantes de la muestra, calculando la distancia euclídea de cada ejemplo y comparándola con la mínima que se haya encontrado. Si en algún momento, se encuentra una distancia menor, se actualiza este valor y el de la clase, guardando la que corresponda con dicho ejemplo.

En el cálculo de la distancia euclídea, si el peso de esa característica es menor que el establecido como mínimo, no se tiene en cuenta para calcularla.

Finalmente, se devuelve la clase mínima encontrada, que será la clase del ejemplo desconocido.

b) Algoritmo Relief:

```
data = instances
w = initialize(numAttributes, 0)
for i in numInstances
{
    posEnemy = searchEnemy(data[i])
    posFriend = searchFriend(data[i])
    for j in numAttributes
    {
        enemy = data[i][j] - data[posEnemy][j]
        friend = data[i][j] - data[posFriend][j]
        w[j] = w[j] + |enemy| - |friend|
    }
}
w = normalize(w)
return w
```

Descripción en lenguaje natural:

- *data* contiene todos los ejemplos de la muestra que se está tratando.
- *w* es un contenedor de números reales inicializados en 0 y de longitud igual al número de atributos del problema.

Este algoritmo comienza inicializando un vector de pesos a 0.

Recorre todos los ejemplos de la muestra y por cada uno de ellos calcula la posición que ocupa su enemigo más cercano (otro ejemplo cuya clase es diferente al del ejemplo actual) y la posición que ocupa su amigo más cercano (otro ejemplo cuya clase es igual al del ejemplo actual).

Cuando ya ha calculado la posición de su enemigo/amigo más cercano, recorre todos los atributos de *w* y le suma la diferencia entre el ejemplo actual y su enemigo y le resta la diferencia entre el ejemplo actual y su amigo.

Para terminar, debe normalizar todos los datos de *w* porque pueden encontrarse algunos que sean negativos (se truncan a 0) o algunos que sean mayores de 1.

Devuelve *w* que contiene una solución al problema.

5. Explicación del procedimiento considerado para desarrollar la práctica

El núcleo principal de esta práctica ha sido desarrollado sin usar ningún framework, ni código proporcionado en prácticas o cualquier otro lugar.

El lenguaje de programación utilizado ha sido *C++* en el 100% de la práctica.

La aplicación se compone de seis clases principales:

- *Arff*: contiene la estructura de datos y todos los métodos necesarios para leer y realizar operaciones con la muestra.
- *Sorter*: es el clasificador de la aplicación. Hace uso de la solución que se va generando durante toda la práctica, para en base a los datos almacenados en *Arff*, clasificar un ejemplo. Contiene los *algoritmos de generación aleatoria de la solución*, *Relief*, *1-NN*, *distancia euclídea* y *función de evaluación*.
- *LocalSearch*: se encarga de aplicar la *Búsqueda Local* a una solución en concreto y devolverla. Contiene el *algoritmo de mutación* y el de *Búsqueda Local*.
- *GeneticA*: es la clase que aplica los *Algoritmos Genéticos*. Tiene dos *Algoritmos Genéticos* implementados y gestiona, dependiendo de los parámetros recibidos, el tipo de cruce que realizar (*BLX* o *cruce aritmético*) o la aplicación de la *Búsqueda Local* para convertir el *Algoritmo Genético Generacional* en *Algoritmo Memético*. Contiene el *algoritmo de Selección por Torneo Binario*, *cruce BLX*, *cruce aritmético*, *mutación*, *Algoritmo Genético Generacional* y *Algoritmo Genético Estacionario*.
- *TSA*: Trajectory based Search Algorithms. Contiene los algoritmos *Enfriamiento Simulado* y *Búsqueda Local Iterativa*. Controla los parámetros que necesita cada uno para su desarrollo.
- *DE*: Differential Evolution. Contiene el algoritmo de *Evolución Diferencial*.

Está gestionada por un archivo principal (*main.cpp*) que se encarga de lanzar todo el proceso y mostrar en pantalla la resolución de los algoritmos, publicando la tasa de acierto y tiempo en segundos que ha necesitado en la ejecución.

Para compilar todo el proyecto y crear el archivo ejecutable que lanzará la aplicación, existe un archivo Makefile.

Además, para la realización de la práctica, se hace uso de las siguientes librerías:

- *OpenMP*: la práctica está implementada haciendo uso de programación paralela a través de *OpenMP*.
- *random.h*: código disponible en la sección de código fuente de la web de la asignatura. Se hace uso de una semilla (*Seed*) que se inicializa con un valor dado por parámetro al ejecutar la aplicación para obtener los números aleatorios necesarios en el desarrollo de los algoritmos.
- *<sys/time.h>*: para medir el tiempo que requiere cada algoritmo.

- *<random>*: para generar el número aleatorio de la distribución normal de media 0 y desviación típica 0.3.
- *ARFF formatted file reader in C++*: para leer los archivos *.arff*. Arff Github

  Ha sido necesario readaptar levemente el código disponible de esta librería para solucionar problemas generados en la lectura de los archivos.

a) Manual de usuario:

Para compilar la aplicación, se hace uso de un archivo *Makefile*.

Se debe descomprimir el archivo *.zip* y ejecutar *make* en la raíz de los directorios generados.

Se habrá creado un archivo ejecutable en la carpeta */bin* llamado *p2*.

En la carpeta *data* se incluyen los archivos *.arff* con los que se quiera ejecutar la aplicación.

Para lanzar la aplicación se debe especificar qué archivo se desea utilizar y el valor de la semilla de inicialización de los valores aleatorios. Así, para la ejecución de esta práctica, se han utilizado los siguientes comandos a través del terminal:

- *./bin/p2 sonar.arff 1824*
- *./bin/p2 wdbc.arff 1824*
- *./bin/p2 spambase-460.arff 1824*

El archivo *Makefile* dispone de limpieza en caso de querer recompilar:

- *make clean*
- *make mr.proper*

6. Experimentos y análisis de resultados

a) Descripción de los casos del problema empleados y de los valores de los parámetros considerados en las ejecuciones de cada algoritmo:

En la realización de esta práctica se han utilizado tres tipos de problemas:

1. *Sonar*: conjunto de datos de detección de materiales mediante señales de sónar, discriminando entre objetos metálicos y rocas. 208 ejemplos con 60 características que deben ser clasificados en 2 clases.

   El fichero utilizado para obtener estos datos está nombrado como *sonar.arff*. Se utiliza la librería *ARFF formatted file reader in C++*. La clase que determina el tipo de material del ejemplo se encuentra situada al final de todos los atributos. Al cargar el archivo en la clase *Arff* se debe indicar como primer parámetro el nombre del fichero y como segundo parámetro *false*, indicando que la clase del ejemplo se encuentra al final.

2. *WDBC*: esta base de datos contiene 30 características calculadas a partir de una imagen digitalizada de una punción aspiración con aguja fina (PAAF) de una masa en la mama. Se describen las características de los núcleos de las células presentes en la imagen. La tarea consiste en determinar si un tumor encontrado es benigno o maligno (M = maligno, B = benigno). 569 ejemplos con 30 características que deben ser clasificados en 2 clases.

   El fichero utilizado para obtener estos datos está nombrado como *wdbc.arff*. Se utiliza la librería *ARFF formatted file reader in C++*. La clase que determina el tipo de tumor del ejemplo se encuentra situada al principio de todos los atributos. Al cargar el archivo en la clase *Arff* se debe indicar como primer parámetro el nombre del fichero y como segundo parámetro *true*, indicando que la clase del ejemplo se encuentra al principio.

3. *SpamBase*: conjunto de datos de detección de SPAM frente a correo electrónico seguro. 460 ejemplos con 57 características que deben ser clasificados en 2 clases.

   El fichero utilizado para obtener estos datos está nombrado como *spambase-460.arff*. Se utiliza la librería *ARFF formatted file reader in C++*. La clase que determina si el ejemplo es spam o no se encuentra situada al final de todos los atributos. Al cargar el archivo en la clase *Arff* se debe indicar como primer parámetro el nombre del fichero y como segundo parámetro *false*, indicando que la clase del ejemplo se encuentra al final.

La semilla de inicialización para la ejecución de esta prueba es *1824* para todos los problemas y se especifica al ejecutar la aplicación.

Los ficheros de datos que leerá la aplicación se encuentran en el directorio *data*, aunque también se podrán encontrar disponibles en el directorio *bin*, tal y como se indica en el guion.

b) Experimento 1. Valor de reducción dinámico:

En esta práctica se ha llevado a cabo un experimento, el cual, cambia el valor mínimo que se tiene en cuenta para calcular la tasa de reducción de las soluciones obtenidas y, por tanto, cambia el resultado de la función objetivo.

La idea nace al observar que los algoritmos, pasadas unas iteraciones/generaciones, estabilizan la tasa de acierto o reducción que van obteniendo.

Esta estabilización de la función objetivo, provoca un uso innecesario de tiempo, cómputo y número de evaluaciones, que se podrían aprovechar para mejorar los resultados.

Lo que se plantea es, si pasado un cierto número de iteraciones/generaciones, la función objetivo no aumenta, es probable que no vuelva a aumentar. Puede deberse a que se encuentre en un óptimo local y esto hace que, a simple vista, beneficie más a los algoritmos que hacen un gran uso de la explotación en lugar de la exploración. Es en ese momento en el que se aumenta ese valor mínimo que se utiliza para el cálculo de la tasa de reducción.

Al aumentar el valor mínimo de la tasa de reducción, el número de atributos que no se tienen en cuenta para clasificar el ejemplo es mayor y esto provoca:

- Aumento de la función objetivo por relación directa con la tasa de reducción. A mayor número de atributos no utilizados, mayor porcentaje de reducción y mejor resultado en la función objetivo.
- Variación desconocida en la tasa de acierto. No utilizar, sin motivo aparente, una serie de atributos de la solución puede llevar a mejorar o empeorar la tasa de acierto, sin saber exactamente qué sucederá. Esta variación de la tasa de acierto está implicada directamente en la función objetivo. Lo lógico que se puede pensar es que, al utilizar menos atributos, la precisión de la solución empeora y la tasa de acierto es peor, reduciendo el valor de la función objetivo.

Se tiene, por tanto, una teoría. Aumentando el valor mínimo que se utiliza en el cálculo de la reducción, aumentará la tasa de reducción y disminuirá la tasa de acierto. Se debería controlar cómo actúa esta técnica y encontrar un equilibrio entre ambas tasas que lleven a maximizar la función objetivo.

Se puede pensar que no es una buena técnica si la tasa de acierto disminuye, pero se cuenta con que se le ha dado el mismo peso a la tasa de acierto y a la de reducción, por lo que una solución con 90% de tasa de acierto y 10% de tasa de reducción es tan buena como una solución con 10% de tasa de acierto y 90% de tasa de reducción.

Esta técnica puede generar soluciones peores a las que se tienen, por lo que es necesario guardar la mejor solución obtenida hasta ese momento.

Los algoritmos en los que se ha aplicado este experimento son:

- Enfriamiento Simulado.
- Búsqueda Local Iterativa.
- Evolución Diferencial con ambos cruces.
- Genético Generacional con cruce BLX.

En los resultados de los datos ya se incluyen estos algoritmos con y sin el experimento para poder analizarlos.

Para encontrar un equilibrio entre la cantidad de veces que debe aumentar la tasa mínima de reducción y cuánto debe aumentar, se han lanzado bastantes ejecuciones de los algoritmos con el fin de comprobar cómo actúa a través de ensayo-error.

La idea es que, de media, alcance progresivamente 0.95 (valor máximo que puede aumentar) hacia el final del número de iteraciones/generaciones. Existen dos maneras de hacerlo: o bien hacer pocos cambios con un aumento elevado de la tasa de reducción; o bien modificar en muchas ocasiones la tasa mínima, pero con poca variabilidad.

Cada algoritmo tiene configurado sus propios valores que ofrecen un resultado positivo para el experimento.

c) Resultados obtenidos según el formato especificado:

*En la carpeta *data* que se encuentra junto con este documento, se ofrecen todos los archivos de datos de la ejecución de la práctica.

Tabla 6.c.1: Resultados obtenidos por el algoritmo 1-NN en el problema del APC

Algoritmo	Sonar				Wdbc				Spambase			
	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T
Partición 1	43,18	86,36	0	0,007227	47,44	94,87	0	0,014629	39,68	79,35	0	0,013245
Partición 2	45,12	90,24	0	0,000389	47,35	94,69	0	0,001695	41,85	83,7	0	0,00169
Partición 3	42,69	85,37	0	0,000391	46,02	92,04	0	0,00172	44,02	88,04	0	0,001615
Partición 4	39,03	78,05	0	0,000394	46,91	93,81	0	0,001719	39,68	79,35	0	0,001636
Partición 5	45,12	90,24	0	0,00039	49,12	98,23	0	0,001702	41,31	82,61	0	0,002911
Media	43,03	86,05	0,00	0,00175820	47,36	94,73	0,00	0,00429300	41,31	82,61	0,00	0,00421940

Tabla 6.c.2: Resultados obtenidos por el algoritmo Relief en el problema del APC

Algoritmo	Sonar				Wdbc				Spambase			
	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T
Partición 1	51,75	81,82	21,67	0,000462	51,20	95,73	6,67	0,001993	53,08	72,83	33,33	0,001687
Partición 2	58,01	92,68	23,33	0,000452	53,57	93,81	13,33	0,001934	60,85	76,09	45,61	0,001649
Partición 3	50,19	85,37	15	0,000454	55,68	94,69	16,67	0,001937	57,97	82,61	33,33	0,00163
Partición 4	50,69	78,05	23,33	0,000455	54,90	96,46	13,33	0,001814	59,73	82,61	36,84	0,001696
Partición 5	51,85	85,37	18,33	0,000472	55,34	97,35	13,33	0,001814	59,60	85,87	33,33	0,001619
Media	52,50	84,66	20,33	0,00045900	54,14	95,61	12,67	0,00189840	58,25	80,00	36,49	0,00165620

Tabla 6.c.3: Resultados obtenidos por el algoritmo ES en el problema del APC

Algoritmo	Sonar				Wdbc				Spambase			
	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T
Partición 1	72,66	88,64	56,67	6,01731	82,87	95,73	70	24,968	74,10	81,52	66,67	22,2928
Partición 2	76,73	95,12	58,33	5,74422	83,57	93,81	73,33	24,8942	77,32	96,74	57,89	22,2807
Partición 3	77,62	90,24	65	5,71189	86,13	95,58	76,67	24,7883	73,51	89,13	57,89	22,755
Partición 4	74,23	95,12	53,33	5,76503	87,79	95,58	80	24,4835	76,35	91,3	61,4	22,8099
Partición 5	77,56	95,12	60	5,72995	86,13	95,58	76,67	24,5853	77,44	93,48	61,4	22,7612
Media	75,76	92,85	58,67	5,79368000	85,30	95,26	75,33	24,74386000	75,74	90,43	61,05	22,57992000

Tabla 6.c.3.Exp: Resultados obtenidos por el experimento en el algoritmo ES en el problema del APC

Algoritmo	Sonar				Wdbc				Spambase			
	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T
Partición 1	73,79	90,91	56,67	5,75115	91,54	89,74	93,33	24,3232	74,31	83,7	64,91	22,3798
Partición 2	79,68	92,68	66,67	5,74952	92,35	94,69	90	24,2494	80,95	93,48	68,42	22,5315
Partición 3	80,06	95,12	65	5,75113	92,69	92,04	93,33	24,4429	83,96	83,7	84,21	22,7076
Partición 4	77,62	90,24	65	5,93177	91,91	93,81	90	24,4289	78,11	91,3	64,91	22,70330000
Partición 5	78,40	95,12	61,67	5,66286	93,57	93,81	93,33	24,3277	80,20	90,22	70,18	22,9611
Media	77,91	92,81	63,00	5,76928600	92,41	92,82	92,00	24,35442000	79,50	88,48	70,53	22,65666000

Tabla 6.c.4: Resultados obtenidos por el algoritmo ILS en el problema del APC

Algoritmo	Sonar				Wdbc				Spambase			
	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T
Partición 1	54,93	93,18	16,67	5,84126000	62,44	94,87	30	15,7941	52,71	82,61	22,81	23,4098
Partición 2	58,40	95,12	21,67	5,83124000	73,13	92,92	53,33	17,1323	58,81	91,3	26,32	23,2081
Partición 3	58,90	87,8	30	5,83883000	67,35	94,69	40	18,1819	53,13	86,96	19,3	24,1114
Partición 4	53,97	82,93	25	5,86208000	59,01	94,69	23,33	18,6096	55,76	86,96	24,56	24,0595
Partición 5	59,68	92,68	26,67	5,83615000	65,34	97,35	33,33	16,0005	58,27	90,22	26,32	23,7777
Media	57,17	90,34	24,00	5,84191200	65,45	94,90	36,00	17,14368000	55,74	87,61	23,86	23,71330000

Tabla 6.c.4..Exp: Resultados obtenidos por el experimento en el algoritmo ILS en el problema del APC

Algoritmo	Sonar				Wdbc				Spambase			
	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T
Partición 1	78,34	100	56,67	5,68112	84,53	95,73	73,33	15,2465	73,89	79,35	68,42	22,3503
Partición 2	83,46	90,24	76,67	5,73758	89,25	88,5	90	17,6759	82,16	92,39	71,93	22,3276
Partición 3	82,18	92,68	71,67	5,57782	94,01	94,69	93,33	16,8632	84,50	84,78	84,21	22,4124
Partición 4	80,25	80,49	80	5,69373	92,35	94,69	90	19,4586	70,46	84,78	56,14	22,8936
Partición 5	83,07	87,8	78,33	5,68706	84,90	96,46	73,33	17,5375	73,80	73,91	73,68	22,7042
Media	81,46	90,24	72,67	5,67546200	89,01	94,01	84,00	17,35634000	76,96	83,04	70,88	22,53762000

Tabla 6.c.5: Resultados obtenidos por el algoritmo DE/rand/1 en el problema del APC

Algoritmo	Sonar				Wdbc				Spambase			
	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T
Partición 1	92,50	100	85	5,13513000	94,58	99,15	90	23,0358	91,81	92,39	91,23	21,0059
Partición 2	95,00	100,00	90,00	5,0732	93,68	97,35	90	23,4103	92,02	94,57	89,47	24,7205
Partición 3	92,50	100,00	85,00	5,02471000	94,90	96,46	93,33	23,4245	93,23	93,48	92,98	21,2494
Partición 4	93,34	100,00	86,67	5,09326000	94,90	96,46	93,33	22,9527	93,36	90,22	96,49	21,1731
Partición 5	93,40	95,12	91,67	5,17421000	94,90	96,46	93,33	23,3782	93,23	93,48	92,98	21,1166
Media	93,35	99,02	87,67	5,10010200	94,59	97,18	92,00	23,24030000	92,73	92,83	92,63	21,85310000

Tabla 6.c.5.Exp: Resultados obtenidos por el experimento en el algoritmo DE/rand/1 en el problema del APC

Algoritmo	Sonar				Wdbc				Spambase			
	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T
Partición 1	92,50	100	85	5,12696	94,15	98,29	90	23,1667	91,81	92,39	91,23	20,7024
Partición 2	95,00	100	90	5,25475	94,12	98,23	90	23,0807	92,69	92,39	92,98	21,0549
Partición 3	91,67	100	83,33	5,0454	94,90	96,46	93,33	22,9524	91,60	90,22	92,98	21,1936
Partición 4	93,34	100	86,67	5,05388	94,90	96,46	93,33	22,9665	93,02	91,3	94,74	21,1946
Partición 5	94,17	100	88,33	5,23475	94,80	92,92	96,67	22,6186	92,36	93,48	91,23	21,1339
Media	93,33	100,00	86,67	5,14314800	94,57	96,47	92,67	22,95698000	92,29	91,96	92,63	21,05588000

Tabla 6.c.6: Resultados obtenidos por el algoritmo DE/current-to-best/1 en el problema del APC

Algoritmo	Sonar				Wdbc				Spambase			
	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T
Partición 1	76,06	95,45	56,67	5,15983	87,44	94,87	80	22,8237	74,52	85,87	63,16	21,0339
Partición 2	86,67	100	73,33	5,14047	89,36	92,04	86,67	22,8886	82,04	95,65	68,42	21,2387
Partición 3	80,90	95,12	66,67	5,14821	93,57	93,81	93,33	22,6855	80,08	93,48	66,67	21,3404
Partición 4	74,29	90,24	58,33	5,16742	86,23	99,12	73,33	23,2732	82,92	95,65	70,18	21,2186
Partición 5	81,28	97,56	65	5,09369	92,01	97,35	86,67	22,7071	82,16	92,39	71,93	21,3068
Media	79,84	95,67	64,00	5,14192400	89,72	95,44	84,00	22,87562000	80,34	92,61	68,07	21,22768000

Tabla 6.c.6.Exp: Resultados obtenidos por el experimento en el algoritmo DE/current-to-best/1 en el problema del APC

Algoritmo	Sonar				Wdbc				Spambase			
	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T
Partición 1	89,09	93,18	85	5,10747	93,64	90,6	96,67	22,3908	84,75	78,26	91,23	20,9496
Partición 2	93,34	100	86,67	5,10554	92,24	91,15	93,33	22,9206	88,72	96,74	80,7	21,1711
Partición 3	84,74	87,8	81,67	5,17481	94,46	95,58	93,33	23,1127	84,29	82,61	85,96	21,3742
Partición 4	89,68	92,68	86,67	5,10913	91,46	92,92	90	23,0531	89,18	92,39	85,96	21,2453
Partición 5	89,29	90,24	88,33	5,09575	91,02	92,04	90	22,9739	87,00	88,04	85,96	21,0586
Media	89,22	92,78	85,67	5,11854000	92,56	92,46	92,67	22,89022000	86,79	87,61	85,96	21,15976000

Tabla 6.c.7: Resultados obtenidos por el algoritmo BL en el problema del APC (a partir de 1-NN)

Algoritmo	Sonar				Wdbc				Spambase			
	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T
Partición 1	43,18	86,36	0	0,521469	47,44	94,87	0	1,09049	39,68	79,35	0	1,97881
Partición 2	47,56	95,12	0	0,56521	47,35	94,69	0	1,12452	41,85	83,7	0	1,97549
Partición 3	42,69	85,37	0	0,519105	46,02	92,04	0	1,14167	44,57	89,13	0	2,12724
Partición 4	39,03	78,05	0	0,522823	46,91	93,81	0	1,24918	39,68	79,35	0	2,23418
Partición 5	45,12	90,24	0	0,516881	49,12	98,23	0	1,12478	41,85	83,7	0	2,41517
Media	43,51	87,03	0,00	0,52909760	47,36	94,73	0,00	1,14612800	41,52	83,05	0,00	2,14617800

Tabla 6.c.8: Resultados obtenidos por el algoritmo AGG-BLX en el problema del APC

Algoritmo	Sonar				Wdbc				Spambase			
	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T
Partición 1	65,76	93,18	38,33	6,33582	84,10	94,87	73,33	19,8441	74,60	91,3	57,89	20,2624
Partición 2	72,95	97,56	48,33	6,25489	80,68	94,69	66,67	19,7375	68,34	94,57	42,11	20,923
Partición 3	70,57	87,8	53,33	6,38885	83,68	97,35	70	20,0095	73,94	93,48	54,39	20,4306
Partición 4	68,40	95,12	41,67	6,24873	87,79	95,58	80	19,7122	70,63	95,65	45,61	20,5431
Partición 5	72,95	97,56	48,33	6,38308	84,01	94,69	73,33	19,5312	74,14	95,65	52,63	20,5693
Media	70,12	94,24	46,00	6,32227400	84,05	95,44	72,67	19,76690000	72,33	94,13	50,53	20,54568000

Tabla 6.c.8.Exp: Resultados obtenidos por el experimento en el algoritmo AGG-BLX en el problema del APC

Algoritmo	Sonar				Wdbc				Spambase			
	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T
Partición 1	84,09	93,18	75	6,25524	90,30	90,6	90	19,7686	77,90	89,13	66,67	20,2431
Partición 2	85,00	100	70	6,08138	88,68	97,35	80	19,8772	78,99	91,3	66,67	20,2588
Partición 3	78,01	92,68	63,33	6,33243	91,91	93,81	90	19,6315	78,65	92,39	64,91	20,492
Partición 4	77,95	97,56	58,33	6,55411	90,24	93,81	86,67	19,4728	75,81	90,22	61,4	20,461
Partición 5	85,00	100	70	6,24208	93,23	96,46	90	19,6521	75,69	93,48	57,89	20,4748
Media	82,01	96,68	67,33	6,29304800	90,87	94,41	87,33	19,68044000	77,41	91,30	63,51	20,38594000

Tabla 6.c.9: Resultados obtenidos por el algoritmo AM-(10,0,1mej) en el problema del APC

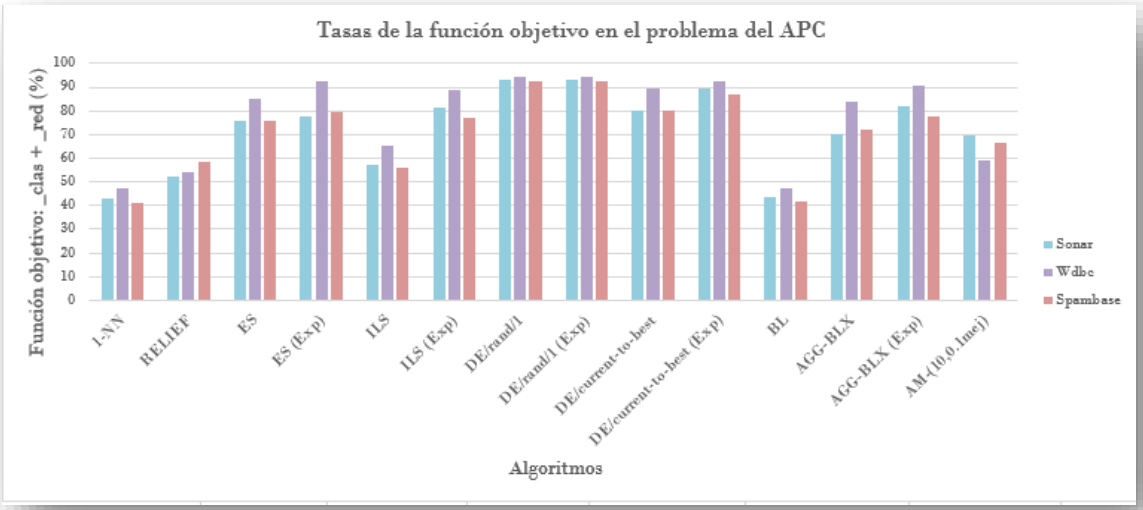
Algoritmo	Sonar				Wdbc				Spambase			
	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T
Partición 1	62,96	90,91	35	6,47382	57,44	94,87	20	12,5225	63,20	91,3	35,09	22,1818
Partición 2	73,78	97,56	50	6,37505	56,13	95,58	16,67	14,1247	65,41	86,96	43,86	23,2944
Partición 3	70,90	95,12	46,67	6,44851	56,46	92,92	20	14,2467	69,55	93,48	45,61	22,9147
Partición 4	69,29	90,24	48,33	6,43308	61,57	96,46	26,67	13,9554	66,71	91,3	42,11	22,9444
Partición 5	70,06	95,12	45	6,45879	62,79	95,58	30	15,2063	67,67	96,74	38,60	23,35640000
Media	69,40	93,79	45,00	6,43785000	58,88	95,08	22,67	14,01112000	66,51	91,96	41,05	22,93834000

Tabla 6.c.10: Resultados globales para el problema del APC

Algoritmo	Sonar				Wdbc				Spambase			
	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T	Agr.	%_clas	%_red	T
1-NN	43,03	86,05	0,00	0,00175820	47,36	94,73	0,00	0,00429300	41,31	82,61	0,00	0,00421940
RELIEF	52,50	84,66	20,33	0,00045900	54,14	95,61	12,67	0,00189840	58,25	80,00	36,49	0,00165620
ES	75,76	92,85	58,67	5,79368000	85,30	95,26	75,33	24,74386000	75,74	90,43	61,05	22,57992000
ES (Exp)	77,91	92,81	63,00	5,76928600	92,41	92,82	92,00	24,35442000	79,50	88,48	70,53	22,65666000
ILS	57,17	90,34	24,00	5,84191200	65,45	94,90	36,00	17,14368000	55,74	87,61	23,86	23,71330000
ILS (Exp)	81,46	90,24	72,67	5,67546200	89,01	94,01	84,00	17,35634000	76,96	83,04	70,88	22,53762000
DE/rand/1	93,35	99,02	87,67	5,10010200	94,59	97,18	92,00	23,24030000	92,73	92,83	92,63	21,85310000
DE/rand/1 (Exp)	93,33	100,00	86,67	5,14314800	94,57	96,47	92,67	22,95698000	92,29	91,96	92,63	21,05588000
DE/current-to-best	79,84	95,67	64,00	5,14192400	89,72	95,44	84,00	22,87562000	80,34	92,61	68,07	21,22768000
DE/current-to-best (Exp)	89,22	92,78	85,67	5,11854000	92,56	92,46	92,67	22,89022000	86,79	87,61	85,96	21,15976000
BL	43,51	87,03	0,00	0,52909760	47,36	94,73	0,00	1,14612800	41,52	83,05	0,00	2,14617800
AGG-BLX	70,12	94,24	46,00	6,32227400	84,05	95,44	72,67	19,76690000	72,33	94,13	50,53	20,54568000
AGG-BLX (Exp)	82,01	96,68	67,33	6,29304800	90,87	94,41	87,33	19,68044000	77,41	91,30	63,51	20,38594000
AM-(10,0.1mej)	69,40	93,79	45,00	6,43785000	58,88	95,08	22,67	14,01112000	66,51	91,96	41,05	22,93834000

d) Análisis de resultados:

- Función objetivo global de cada algoritmo en el problema del APC:



De los datos se puede destacar el algoritmo Genético Generacional con cruce BLX. En la práctica anterior se determinó que era el mejor algoritmo de todos los existentes y en esta ha sido ampliamente superado por el algoritmo de Evolución Diferencial. Se analizará por separado la tasa de acierto y de reducción para identificar por qué su rendimiento no es tan bueno.

Con respecto al algoritmo de Evolución Diferencial, es el que mejor tasa ofrece para los tres conjuntos de datos. Entre los dos tipos de cruces que se han aplicado, el cruce aleatorio mejora claramente al cruce aleatorio + mejor.

Los algoritmos 1-NN, Relief y Búsqueda Local siguen siendo los peores de todos los implementados, siguiéndolos de cerca la Búsqueda Local Iterativa.

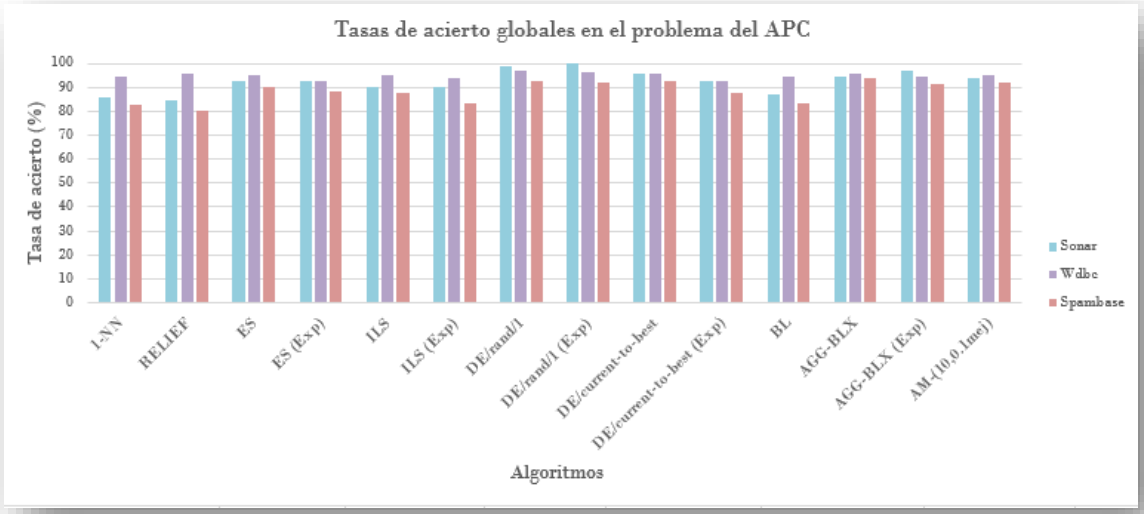
El algoritmo de Enfriamiento Simulado ofrece tasas muy parecidas al algoritmo Genético Generacional pese a ser un tipo de algoritmo diferente.

Una primera impresión del experimento es que ha sido muy positivo. El peor resultado que obtiene es que iguala a la versión original, mejorando aproximadamente un 10% en todos los algoritmos. Se podría destacar la Búsqueda Local Iterativa, que mejora aproximadamente un 20%-25%. Esta mejora puede confirmar el pensamiento inicial que indicaba que los algoritmos que abusan de la explotación serían los mayores beneficiados.

Tras analizar la tasa de acierto, reducción, tiempos y relación %/tiempo, se podrá ver la evolución de los algoritmos en cada iteración/generación para comprobar con claridad cómo funciona el experimento.

Se detallan por separado las tasas que han influido en el resultado de la función objetivo.

- Tasas de acierto globales en el problema del APC:



Sorprendentemente, en esta ejecución de los conjuntos de datos para esta práctica se han obtenido tasas de aciertos muy elevadas.

Incluso los algoritmos 1-NN y Relief, aunque obtienen los peores resultados de todos los algoritmos, mantienen una tasa aceptable que solo es, aproximadamente, un 10%-15% peor que el resto. La Búsqueda Local sigue este mismo razonamiento al utilizar de base el algoritmo 1-NN.

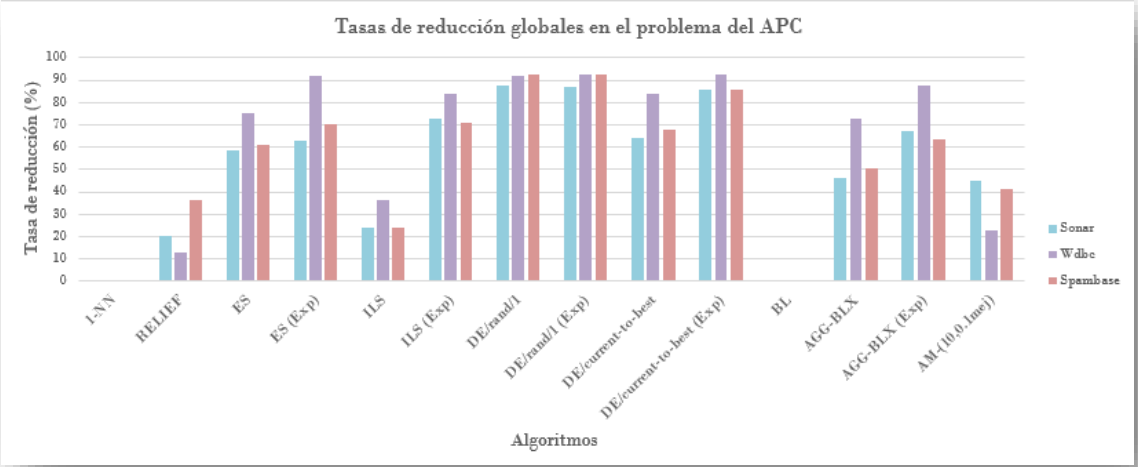
Se ha obtenido 100% de acierto en algunas particiones del algoritmo Evolución Diferencial para Sonar, lo que hace pensar que es perfecto. La realidad es que se vuelve a encontrar con el problema de que son ejemplos demasiado sencillos y que los algoritmos de calidad no encuentran mucho problema en llegar a óptimos.

En la tasa de acierto, el algoritmo Genético Generacional sí se encuentra entre los mejores, como ya se pudo ver en la práctica 1, por lo que el origen de sus peores resultados en la función objetivo radica en la tasa de reducción.

Con respecto a la experimentación, se puede apreciar una disminución general de todos los algoritmos con respecto al original. Aunque esta disminución de la tasa de acierto no es muy relevante, confirma que con el uso de esta técnica se pierde precisión en la solución.

Se han hecho comparaciones con los algoritmos comunes utilizados en ambas prácticas, asignando las particiones de la práctica primera y las particiones de esta práctica segunda y las tasas de acierto obtenidas han sido parecidas, variando aproximadamente un 3%-4%. Se entiende que, las particiones están correctamente creadas, por lo que no encuentro el motivo aparente por el que con el algoritmo de Evolución Diferencial se obtengan tasas de acierto tan elevadas.

- Tasas de reducción globales para el problema del APC:



En la práctica anterior no se encontraba la tasa de reducción, lo que hacía que fuera más difícil realizar un análisis más exhaustivo de los resultados.

El valor inicial de la tasa de reducción es muy pequeño. Esto hace que normalmente sean necesarios la mayor parte de los atributos de la solución para comprobar su calidad. Algoritmos que hagan uso de una mayor explotación verán sacrificada su tasa de reducción, ya que no serán capaces de generar soluciones cuyos atributos sean inferiores a la tasa mínima. Es el caso del algoritmo de Búsqueda Local (perjudicado en parte por 1-NN) o en su defecto, ILS, que obtiene un valor muy pobre.

Algoritmos que abusen de la exploración pueden obtener mayores tasas de reducción y, en este punto, es donde el algoritmo Genético Generacional se ha visto afectado. No ha sido tan agresivo con la generación de nuevas poblaciones como si ha sido el algoritmo de Evolución Diferencial, que obtiene nuevamente los mejores resultados de todos los algoritmos.

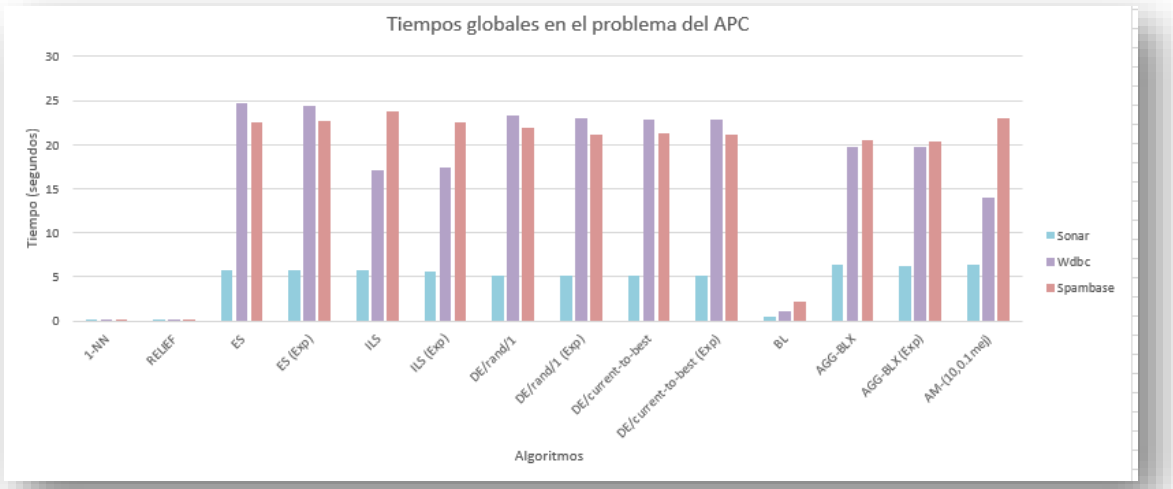
El algoritmo de Enfriamiento Simulado se ve perjudicado por el hecho de explotar más que explorar conforme avanza en su ejecución.

El experimento ha sido ideado pensando en aumentar considerablemente la tasa de reducción, como así lo refleja la gráfica en los algoritmos que hacen uso de él. La tasa de reducción ha aumentado, aproximadamente, un 10-20% en los algoritmos con los que se ha experimentado.

Mención aparte merece el algoritmo de Búsqueda Local Iterativa, que mejora un 40%-50% la tasa de reducción sobre el algoritmo original. A continuación, se detallará la evolución que ha sufrido este algoritmo (y a los que se le ha aplicado el experimento) para comprobar por qué se ha producido la mejora.

Se ha encontrado un equilibrio entre la disminución de la tasa de acierto y el aumento la tasa de reducción, siendo la mejora de la tasa de reducción bastante superior a la disminución de la tasa de acierto que, directamente mejora los resultados de la función objetivo.

- Tiempos globales en el problema del APC:



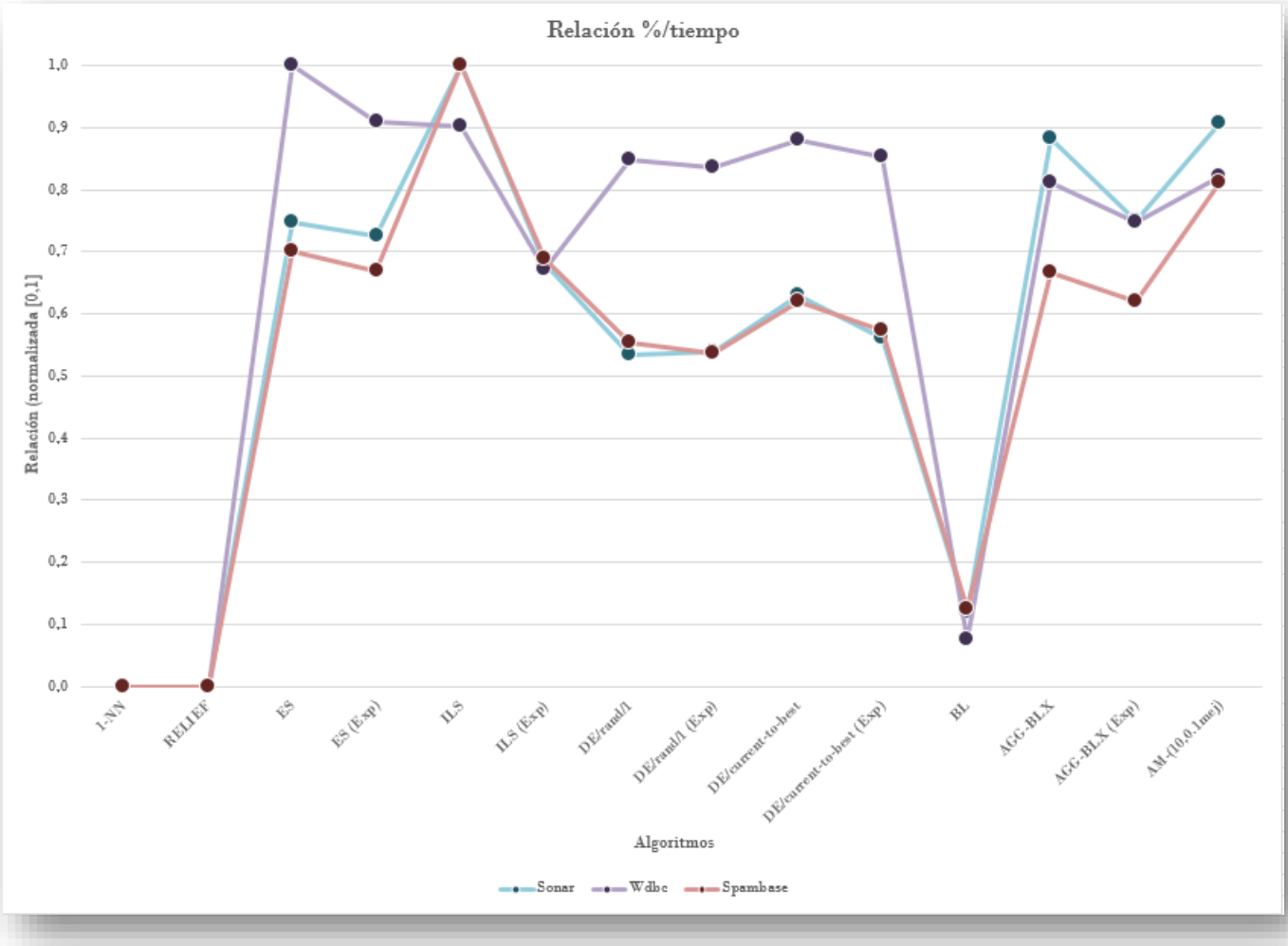
En estos datos se puede apreciar como el problema Sonar es el más sencillo de todos y, debido a esto, el más rápido en ejecutarse. WDBC se encuentra prácticamente en el mismo nivel de complejidad que Spambase, aunque quizás, este último sea el más complejo y laborioso en cuanto a computación.

El algoritmo 1-NN y Relief apenas tienen carga de cómputo ya que utilizan operaciones muy sencillas en sus cálculos.

A partir de la Búsqueda Local, se puede apreciar un incremento del tiempo de cálculo. Al tener que realizar 15.000 evaluaciones de los cromosomas, junto a las mutaciones, hace que el tiempo aumente.

El tiempo total que ha tardado en ejecutarse completamente la aplicación, ha sido 44 minutos y 48 segundos. Si se tiene en cuenta que también se han contabilizado los experimentos, el tiempo real se reduce prácticamente a la mitad, lo que indica que los algoritmos son muy rápidos en comparación con algunos de los utilizados en la práctica anterior.

- Relación Tiempo/Resultado:



Para comprobar qué algoritmos ofrecen mejor relación tiempo/resultado, se puede consultar en la gráfica anterior.

A medida que el algoritmo se acerque a 1, la relación será peor. Esto nos indica que el algoritmo 1-NN y Relief obtienen la mejor relación.

Es tarea del cliente decidir qué algoritmo implementar, valorando la tasa mínima de acierto que está dispuesto a aceptar o el tiempo máximo de que dispone.

Teniendo en cuenta la tasa de acierto, se puede apreciar como el Algoritmo de Evolución Diferencial con cruce aleatorio es la mejor opción para el conjunto Sonar. El experimento del Algoritmo de Evolución Diferencial con cruce aleatorio es el idóneo para el conjunto Spambase, junto con el experimento de la Búsqueda Local Iterativa para el conjunto Wdbc, lo que hace que sea una grata sorpresa haber conseguido los mejores resultados con la experimentación en estos conjuntos.

- Evolución de los algoritmos originales comparados con el experimento:

A continuación, se detalla cómo ha evolucionado la función objetivo, tasa de acierto y tasa de reducción en cada iteración/generación de los algoritmos.

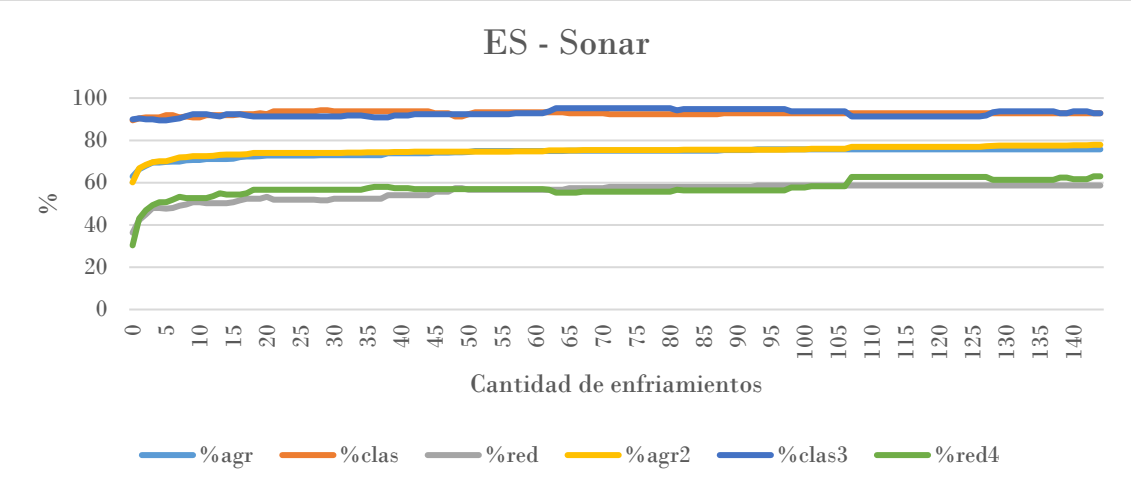
Debido a la aleatoriedad de los algoritmos, se han truncado aquellas iteraciones/generaciones que eran superiores a la mínima para cada algoritmo, con el fin de establecer una relación directa en los gráficos.

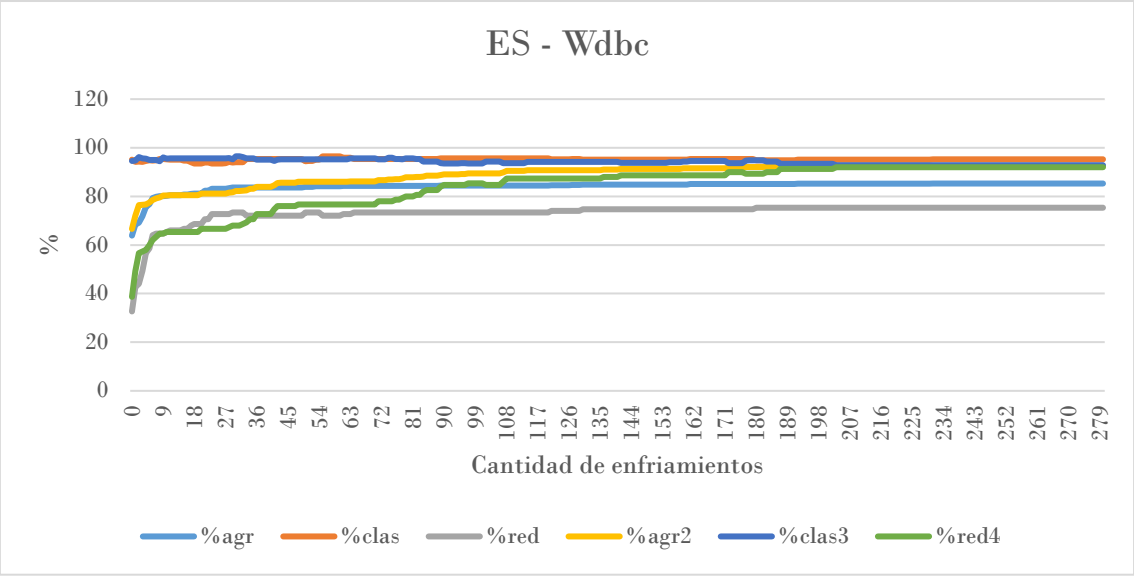
Para facilitar la comprensión de las gráficas, se definen los valores de las leyendas:

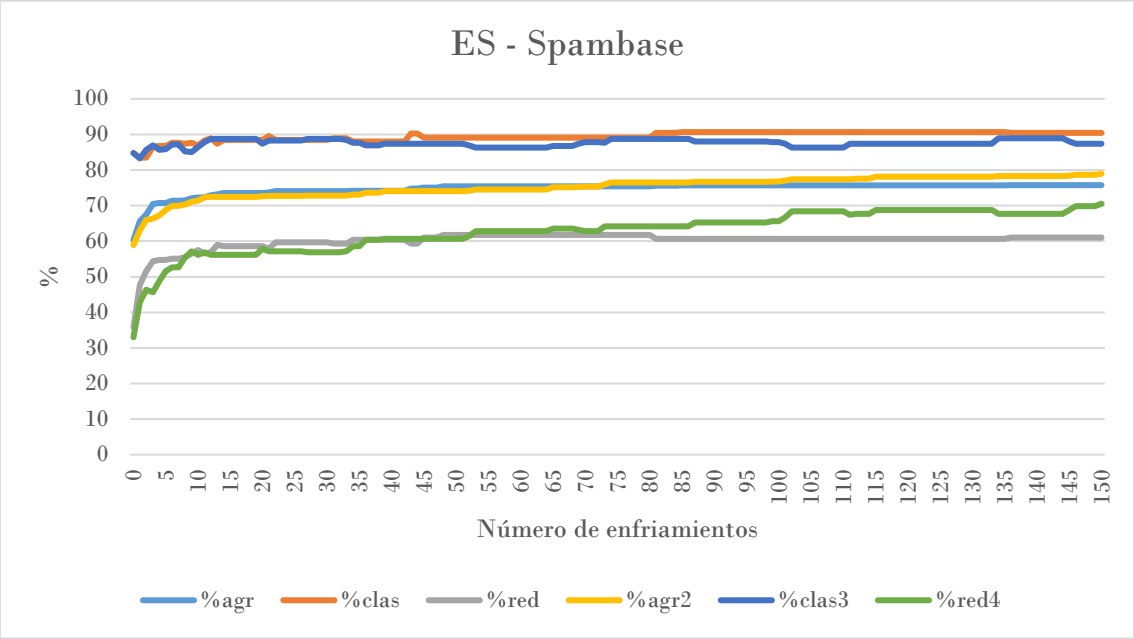
- %agr : función objetivo del algoritmo original.
- %clas: tasa de acierto del algoritmo original.
- %red: tasa de reducción del algoritmo original.
- %agr2: función objetivo del experimento del algoritmo.
- %clas3: tasa de acierto del algoritmo original.
- %red4: tasa de reducción del algoritmo original.

Para mayores consultas, se ofrecen los ficheros de datos originales ubicados en la carpeta *data*.

a) Enfriamiento Simulado:



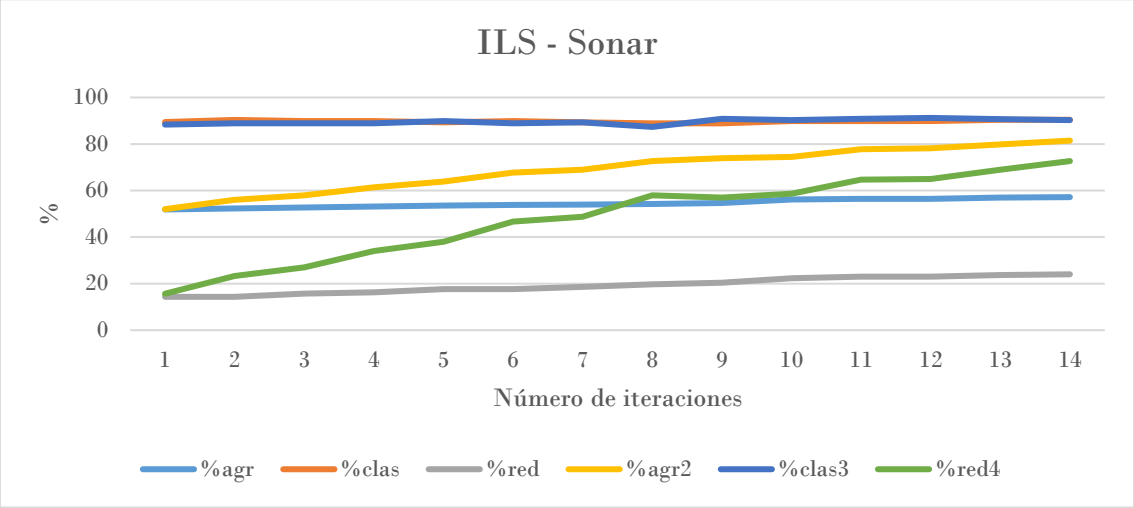


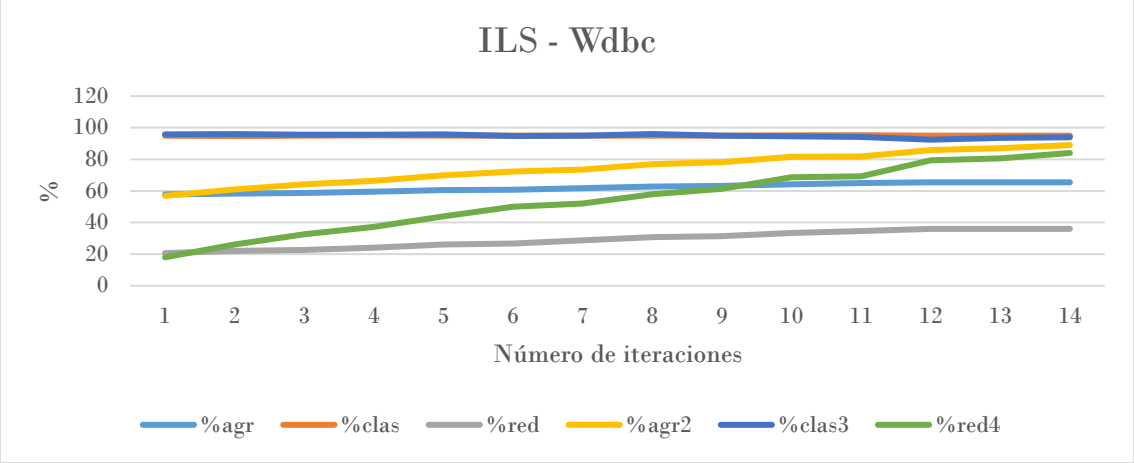


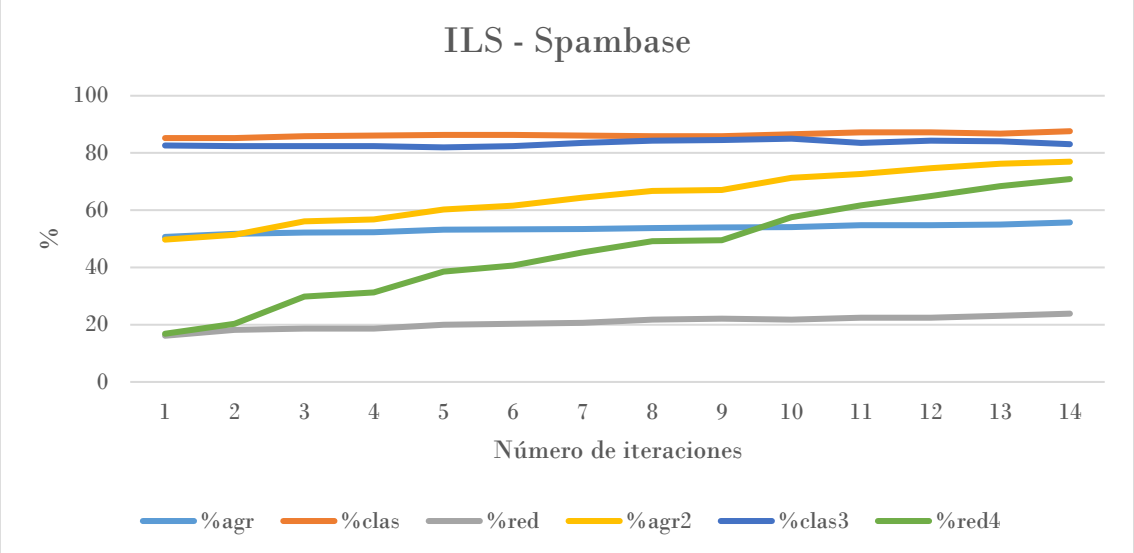
En los tres conjuntos de datos se ve perfectamente reflejado el motivo de haber elegido el experimento realizado. En muy pocos enfriamientos, el algoritmo de Enfriamiento Simulado se encuentra en una zona de óptimo local y, como a medida que avanza en su ejecución, la temperatura disminuye, no se encuentra una gran cantidad de nuevas soluciones.

En cada conjunto de datos, aunque la evolución no sea exactamente igual en el experimento, tiende a comportarse parecida. Las líneas pertenecientes al experimento son muy irregulares, con pequeñas alteraciones en cada enfriamiento. Al perder precisión, la tasa de acierto disminuye levemente, pero la tasa de reducción si va mejorando. Gracias a esto, la función de evaluación es levemente superior.

b) Búsqueda Local Iterativa:



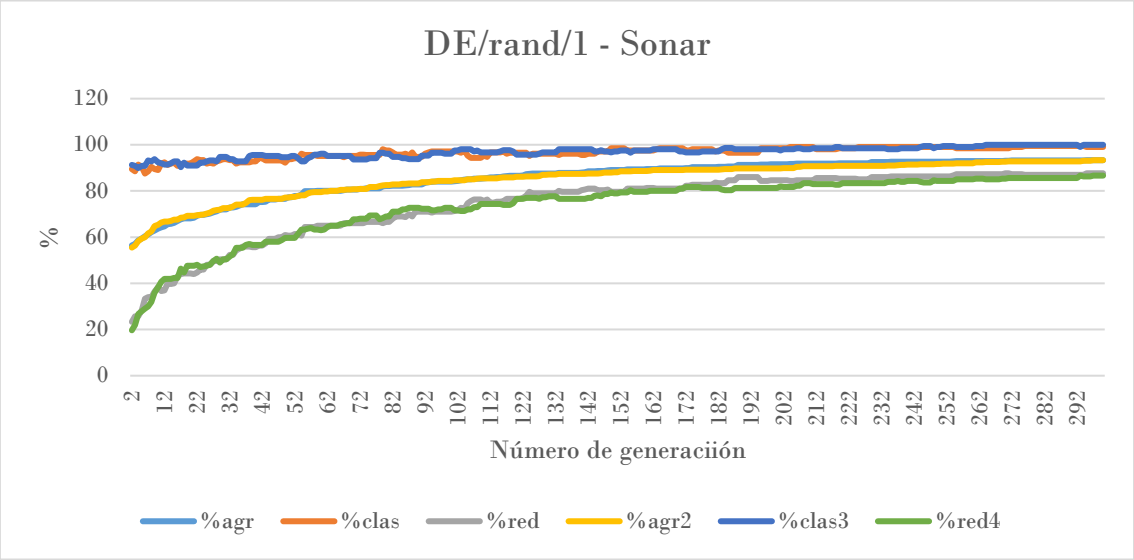


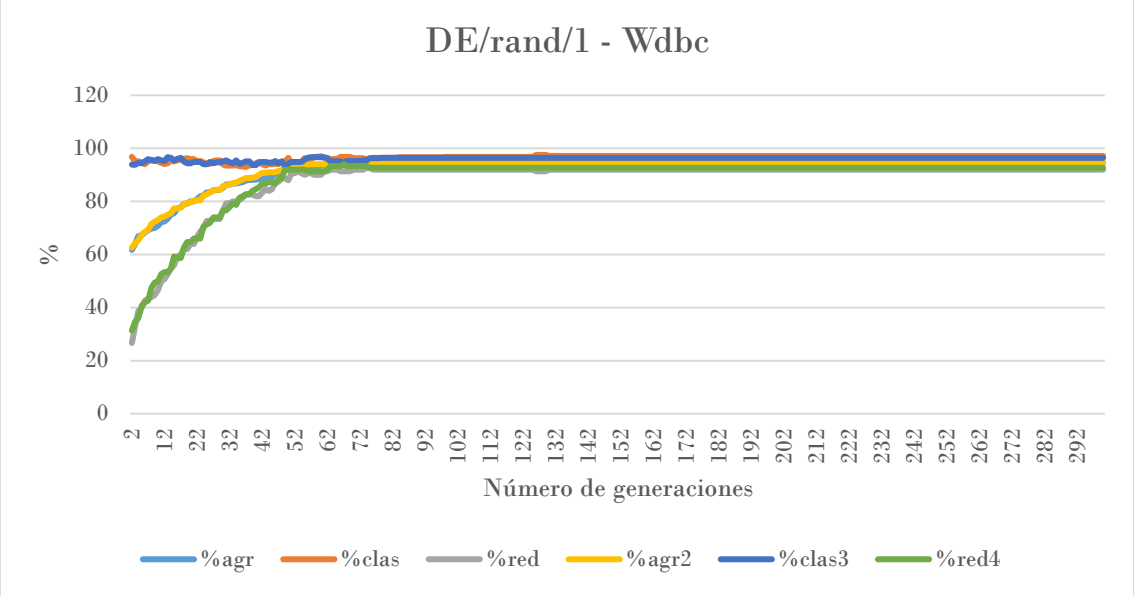


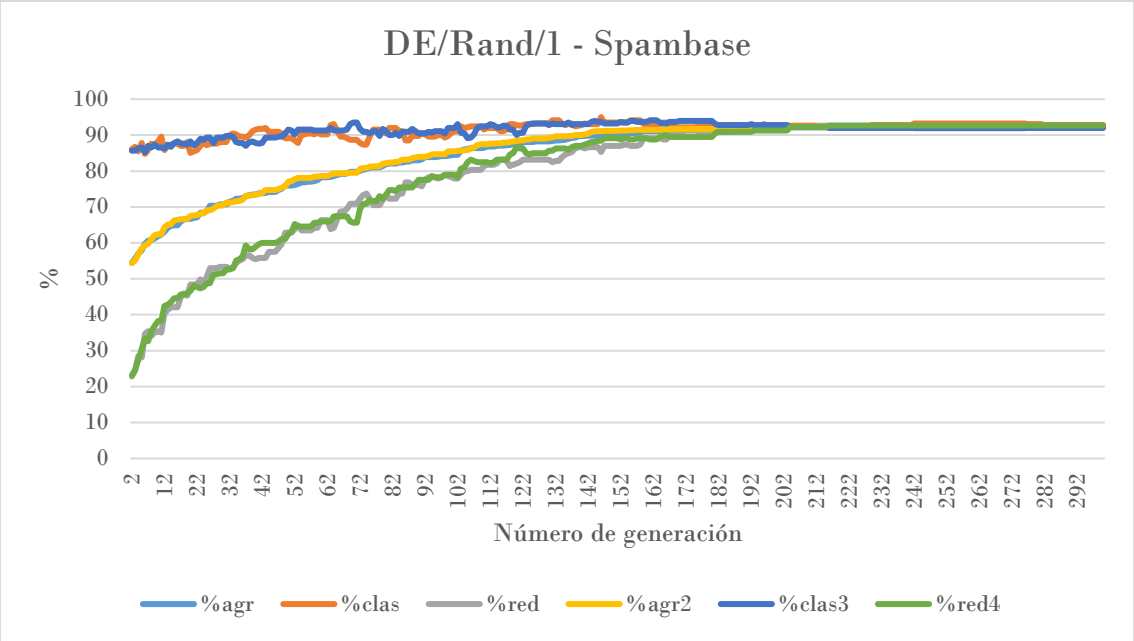
El algoritmo de Búsqueda Local Iterativa se mantiene estable desde el comienzo de su ejecución, con leves mejoras. Esto es una pérdida de evaluaciones que el experimento aprovecha perfectamente.

La tasa de acierto en el experimento es la misma que con el algoritmo original, lo que sumado al gran aumento de la tasa de reducción al utilizar en cada iteración menos atributos, arrastra la mejora de la función objetivo hacia un valor mucho mejor.

c) Evolución Diferencial con cruce aleatorio:





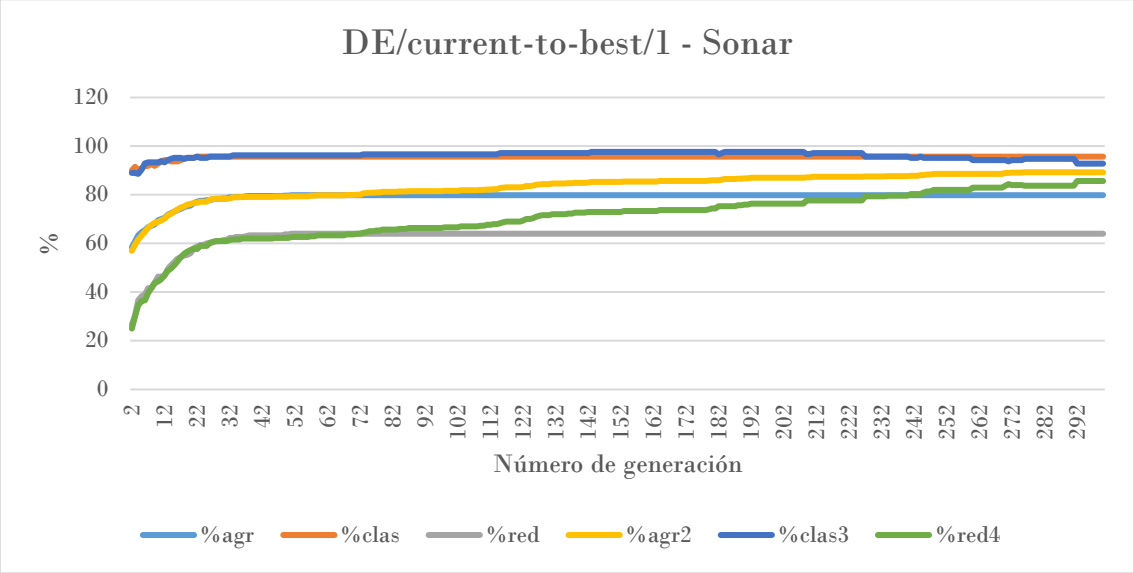


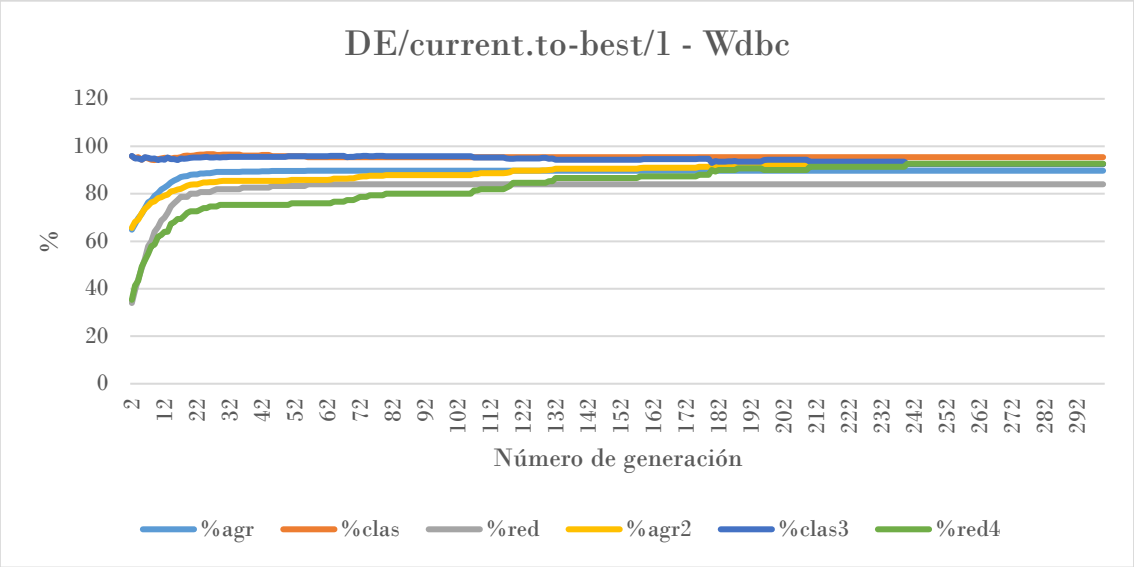
El algoritmo de Evolución Diferencial experimenta una mejora levemente más progresiva que los anteriores algoritmos.

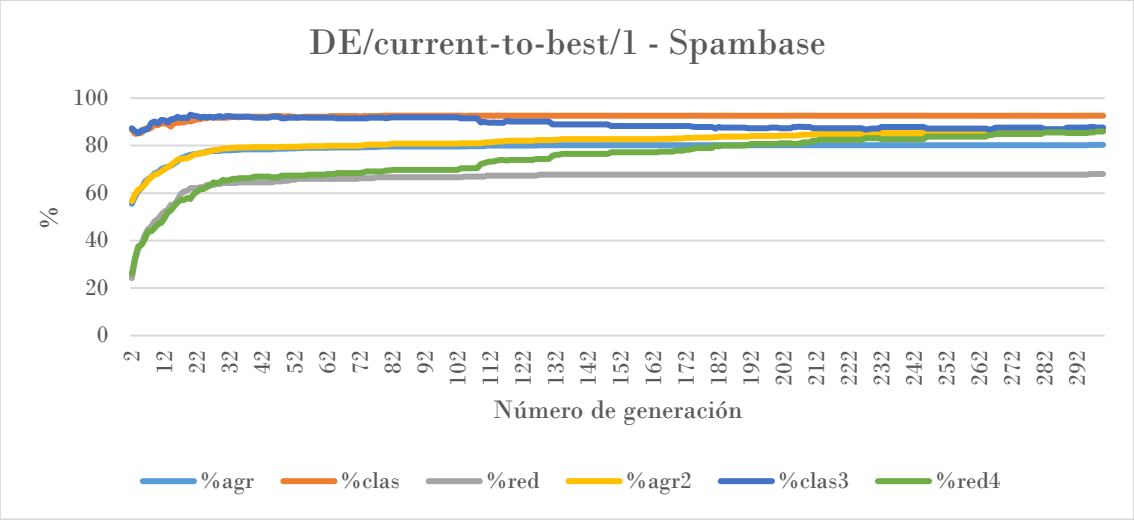
Hay dos factores por los que el experimento no tiene éxito:

- El algoritmo original es bueno y consigue tasas muy altas, por lo que hay poco que mejorar.
- Mejora progresivamente, utilizando más generaciones. El experimento solo tiene sentido cuando el algoritmo se estanca y tras varias generaciones no hay mejora. Si hay mejora, no entra en funcionamiento, por lo que se tiene el algoritmo original.

d) Evolución Diferencial con cruce mejor + aleatorio:

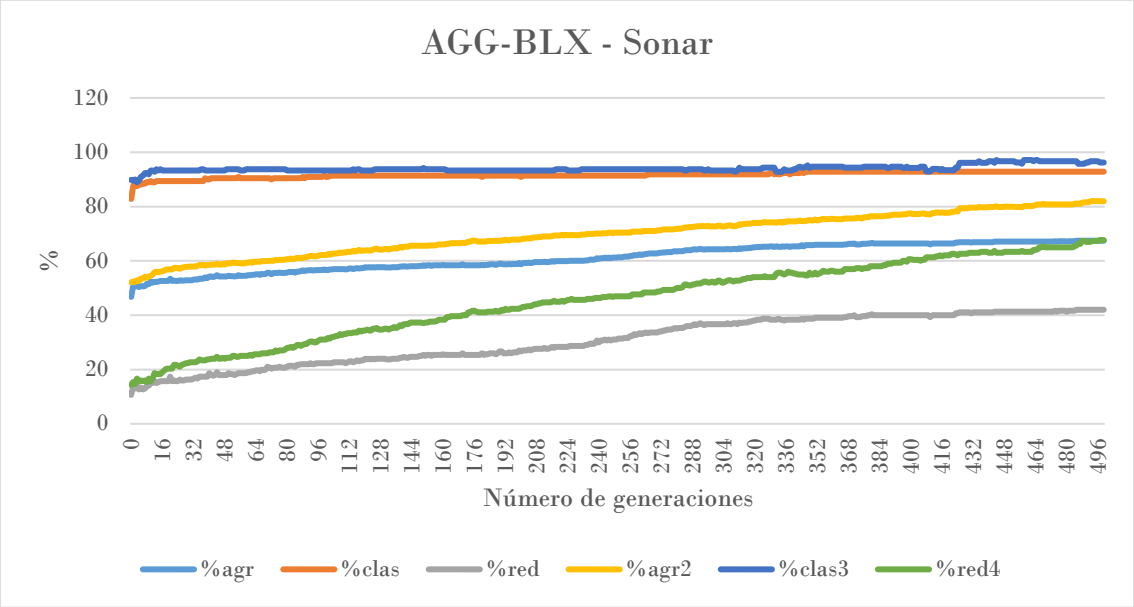


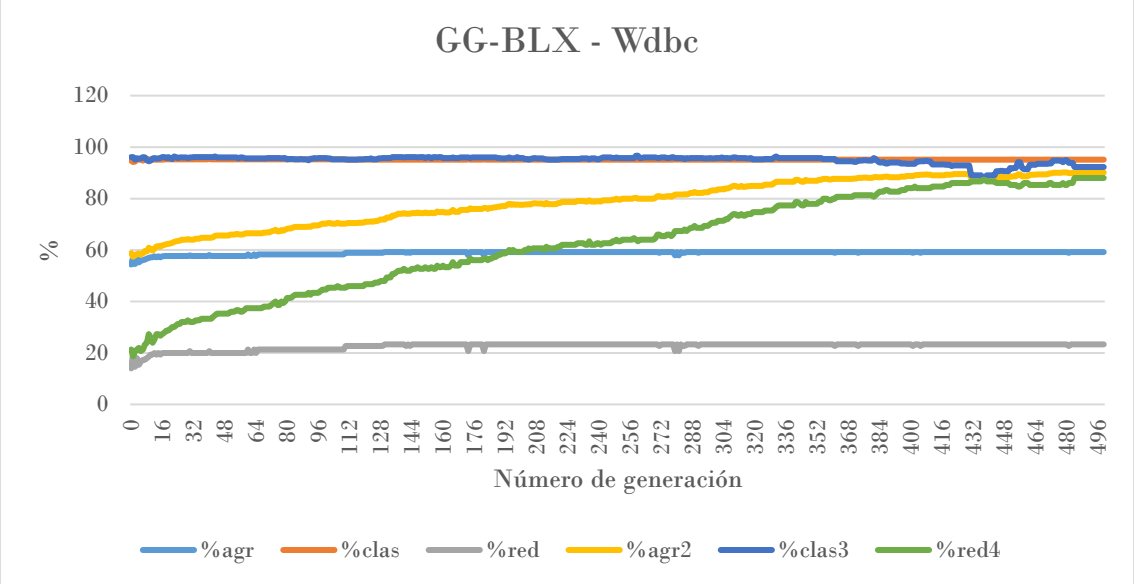


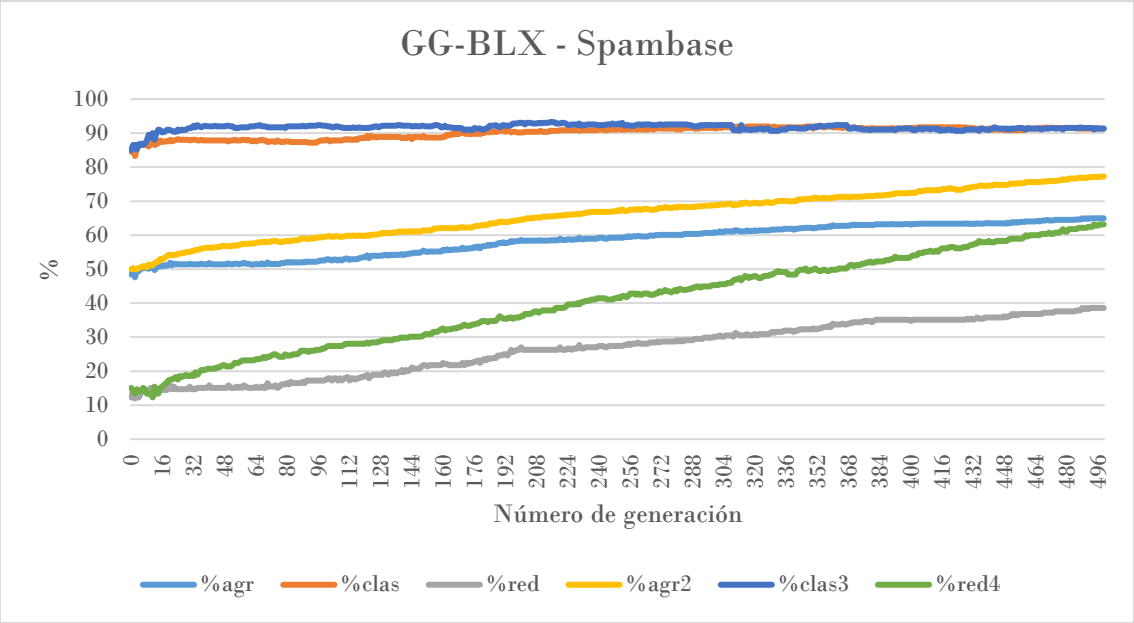


El tipo de cruce forma parte fundamental del algoritmo. En el cruce aleatorio, el hijo que se genera está relacionado con la población en general. En el cruce mejor + aleatorio tiene en cuenta la mejor solución, por lo que ya está condicionado a explotar en esa zona más que explorar por toda la población. Este hecho se encuentra reflejado en una estabilización más rápida de las tasas, lo que hace que cuando no se detecta mejora, se le aplique la modificación a la tasa de reducción. Si lo anterior se suma a un bajo nivel en la tasa de reducción, da lugar a la mejora global durante todas las generaciones (debido a la mejora de la tasa de reducción), llegando a ofrecer un buen resultado de la función objetivo.

e) Algoritmo Genético Generacional con cruce BLX:







Como ya se ha visto anteriormente, la tasa de acierto del Algoritmo Genético Generacional es bastante buena. El problema de este algoritmo es que consigue una tasa de reducción muy baja que, calculando la función objetivo, no ofrece los resultados que se esperan.

En los conjuntos Sonar y Spambase se consigue mejorar poco a poco la función objetivo, lo que hace que el experimento no sea tan efectivo. En el conjunto Wdbc, las tasas se estabilizan, lo que provoca un uso más intenso del experimento y lleva a unas tasas muy altas.

7. Referencias bibliográficas

La información principal para realizar esta práctica ha sido consultada en la web de la asignatura:

- Tema 2. Modelos de Búsqueda: Entornos y Trayectorias vs. Poblaciones.
- Tema 3. Metaheurísticas Basadas en poblaciones.
- Tema 5. Metaheurísticas Basadas en Trayectorias.
- Seminario 2. Problemas de optimización con técnicas basadas en trayectorias simples: Búsqueda Local.
- Seminario 4.
- Guion B: APC.
- Generador de números pseudoaleatorios.
- Tablas e instancias para el problema del Aprendizaje de Pesos en Características (APC)

La biblioteca para leer los ficheros Arff se encuentra en:

- https://github.com/teju85/ARFF

Otras fuentes consultadas:

- http://www.cplusplus.com/reference/vector/vector/vector/
- http://www.cplusplus.com/reference/algorithm/sort/
- https://totaki.com/poesiabinaria/2014/01/concurrencia-posix-threads-y-variables-compartidas-en-c/
- http://stackoverflow.com/questions/10874214/measure-execution-time-in-c-openmp-code
- http://stackoverflow.com/questions/496448/how-to-correctly-use-the-extern-keyword-in-c
- http://stackoverflow.com/questions/4940259/lambdas-require-capturing-this-to-call-static-member-function
- http://stackoverflow.com/questions/9712413/c-stack-smashing-detected
- https://es.wikipedia.org/wiki/Funci%C3%B3n_de_densidad_de_probabilidad
- http://www.cplusplus.com/reference/random/normal_distribution/
- http://ieeexplore.ieee.org/abstract/document/996017/
- http://www.cleveralgorithms.com/nature-inspired/evolution/differential_evolution.html
- http://www.dii.unipd.it/~alotto/didattica/corsi/Elettrotecnica%20computazionale/DE.pdf
- http://wwwae.ciemat.es/~cardenas/curso_AE/DE_cursoAE.pdf
- https://es.wikipedia.org/wiki/Evoluci%C3%B3n_diferencial
- https://en.wikipedia.org/wiki/Iterated_local_search
- http://www.net2plan.com/teaching/files/PyGRC/P3/P3_planif_main.pdf
- https://en.wikipedia.org/wiki/Simulated_annealing