Metaheurística

Práctica 2.b:

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

Curso 2019-2020

Javier Gálvez Obispo
javiergalvez@correo.ugr.es
Grupo 2 Jueves 17:30 – 19:30

Índice

1. Descripción del problema

Nos encontramos ante un problema de agrupamiento o clustering, donde nuestro objetivo es clasificar los datos que nos dan de acuerdo a posibles caractarísticas comunes entre ellos.

Además, en nuestro caso nos encontramos con restricciones de instancia a la hora de agrupar los datos, es decir, dada un pareja de instancias del conjuto de datos puede, o no, haber una restricción entre ellos del tipo *Must-Link*, los dos elementos deben estar en el mismo cluster, o del tipo *Cannot-Link*, si las dos instancias deben encontrarse en clusters diferentes.

Vamos a tratar estas restricciones como restricciones débiles, lo que quiere decir que nuestro objetivo será minimizar el número de restricciones que no se cumplen en vez de encontrar una solución en la que el número de restricciones incuplidas sea cero.

Nuestro objetivo es resolver este problema dados cuatro conjutos de datos diferentes junto a dos listas de restricciones por conjuto de datos, una con un 10% de restricciones y otra con un 20%.

En esta práctica utilizaremos algoritmos genéticos para resolver este problema. Vamos a implementar dos tipos de operadores de cruce, cruce uniforme (**UN**) y cruce por segmento fijo (**SF**), por lo que tendremos dos variantes por cada algoritmo genético. Implementaremos dos algoritmos genéticos generacionales (**AGG**) y dos algoritmos genéticos estacionarios (**AGE**). Además, también se han implementado tres algoritmos meméticos (**AM**) los cuales hibridan AGG-UN con búsqueda local.

2. Elementos en común entre los algoritmos

2.1. Esquema de representación de soluciones

La representación de una posible solución a nuestro problema se hace mediante un vector de longitud *N*, tamaño del conjuto de datos, en el que se almacenan enteros entre 0 y *k*, el número de clusters del problema. Entonces, en la posición *i* de nuestro vector solución *S* estará reflejado el cluster *c* al que pertenece el punto *i* del conjuto de datos. Cada vector solución se guarda junto con el valor resultado de la función objetivo en un ***struct*** *Cromosoma* que facilita el tratamiento de las soluciones.

Struct *Cromosoma* **contains**
    vector<int> S
    float f = -1 // El valor de la función objetivo se inicializa a -1
end

2.2. Función objetivo

La función objetivo se define como:

$$f = \vec{C} + (infeasibility * \lambda)$$

donde $\vec{C}$ es la desviación general de la solución S, *infeasibility* el número de restricciones incumplidas y λ es un parámetro de escalado.

2.2.1. Cálculo de la desviación general

Para calcular la desviación general de una solución primero debemos calcular los centroides μ_i (vector promedio) asociado a cada cluster c_i.

function *calcula_centroides*

Input: Solución *S*, conjunto de datos *X*, número de clusters *k*

double centroides[*k*][*X*[0].size] = [[0, …, 0], …, [0, …, 0]]
int puntos_por_cluster[*k*] = [0, 0, …, 0]

for i **in** {0, 1, …, *X*.size - 1} **do**:
    int cluster = *S*[i]
    puntos_por_cluster[cluster] += 1
    centroides[cluster] = centroides[cluster] + *X*[i]
end for
return centroides / puntos_por_cluster

Calculamos ahora la distancia media intra-cluster, siendo ésta la media de las distancias de las instancias de cada cluster con su centroide asociado.

function *calcula_media_intracluster*

Input: Solución *S*, conjunto de datos *X*, número de clusters *k*

double media_intracluster[*k*] = [0, 0, …, 0], centroides[*k*][*X*[0].size] = *calcula_centroides(S, X, k)*
int puntos_por_cluster[k] = [0, 0, …, 0]

for i **in** {0, …, *X*.size - 1} **do**:
    int cluster = *S*[i]
    puntos_por_cluster[cluster] += 1
    media_intracluster[cluster] += distancia(*X*[i], centroides[cluster])
    // distancia(x, y) calcula la distancia euclidea entre 2 puntos dados
end for
return distancia_media_intracluster / puntos_por_cluster

Y una vez tenemos las distancias medias intra-cluster ya podemos calcular la desviación general de la solución como la media de las distancias recién calculadas.

function *calcula_c*

Input: Solución *S*, conjunto de datos *X*, número de clusters *k*

double c = 0**,** media_intracluster[*k*] = *calcula_media_intracluster(S, X, k)*

```
for distancia in media_intracluster do:
        c = c + distancia
end for
return c / k
```

En el código implementado estas funciones están unidas en una única función

2.2.2. Cálculo de la *infeasibility*

Para calcular la *infeasibility* debemos contar el número de restricciones incumplidas en la solución S, es decir, las veces que se dan los hechos:

- dos instancias en cluster distintos con una restricción *Must-Link*
- dos instancias en un mismo cluster con una restricción *Cannot-Link*

```
function calcula_infeasibility

Input: Solución S, conjuto de restricciones R.

int restricciones_incumplidas = 0

for i in {0, 1, …, S.size - 2} do:
        for j in {i+1, i+2, …, S.size - 1} do:
        if (R_ij == 1) do:
            Si X[i] y X[j] no están en el mismo cluster => restricciones_incumplidas += 1
        elif (R_ij == -1) do:
            Si X[i] y X[j] están en el mismo cluster => restricciones_incumplidas += 1
        end for
end for
return restricciones_incumplidas
```

2.2.3. Cálculo de λ

Definimos λ como el cociente entre la distancia de los puntos más alejados del conjuto y el número de restricciones presentes en el problema.

```
function calcula_lambda

Input: Conjuto de datos X, conjuto de restricciones R.

double max_dist = 0
int num_restricciones = 0
for i in {0, 1, …, X.size - 2} do:
        for j in {i+1, i+2,  …, X.size - 1} do:
            double distancia = distancia(X[i], X[j])
            if (distancia > max_dist) max_dist = distancia
            if (R[i][j] != 0) num_restricciones += 1
        end for
end for
```

return lambda = max_dist / num_restricciones

Entonces, en el código de la función objetivo sólo tenemos que llamar a éstas funciones que acabamos de definir

function *calcula_f*

Input: Solución *S*, Conjuto de datos *X*, conjuto de restricciones *R*, número de clusters *k*.
return *calcula_c(S, X, k) + calcula_infeasibility(S, R) * calcula_lambda(X, R)*

En el código implementado solo se llama una vez a *calcula_lambda(X, R)* por conjunto de datos.

2.3. Generación de soluciones aleatorias

Para generar una solución aleatoria generamos *N*, tamaño del conjuto de datos, número aleatorios entre 0 y *k*, el número de clusters del problema y comprobamos que ninguno de los cluster esté vacio.

function *generar_solución_aleatoria*

Input: Tamaño del conjuto de datos *N*, clusters *k*.

Bool cluster_vacio = true
int S[*N*]

while cluster_vacio == true **do**:
    cluster_vacio = false
    int cuenta[k] = [0, …, 0]

    for i **in** {0, 1, …, *N* – 1} **do**:
        int cluster = *random_int*(0, *k*)
        S[i] = cluster
        cuenta[cluster] += 1
    end for

    if 0 **in** cuenta **then** cluster_vacio = true
end while
return S

En los algoritmos de ésta práctica trabajamos con un conjunto de soluciones, para ello definimos otra función *generar_población* que llamará a *generar_solución_aleatoria* un número *n* de veces y además, transformará las soluciones a *Cromosomas*.

function *generar_población*

Input: Tamaño de la población *n*, Tamaño del conjuto de datos *N*, clusters *k*.

Cromosoma poblacion[*n*]

```
for cromosoma in poblacion do:
        cromosoma.S = generar_solución_aleatoria(N, k)
        // Las soluciones se evalúan posteriormente en otra función
end for
return poblacion
```

3. Operadores de los algoritmos géneticos

3.1. Selección

El operador de selección es utilizado por todos los algoritmos genéticos implementados. Recibe como parámetro el número de cromosomas a seleccionar, *n*, y realiza torneos binarios hasta que obtiene los *n* cromosomas. El resultado que devuelve es una nueva población con los cromosomas ganadores.

```
function seleccion
```

Input: Número de cromosomas a elegir *n*, Población *P*.

Cromosoma poblacion_elegida[*n*]

for i **in** {0, 1, …, *n*} **do**:
        int x = *random_int*(0, *P*.size), y = *random_int*(0, *P*.size)
        // Torneo binario
        if *P*[x].f < *P*[y].f **then** poblacion_elegida[i] = *P*[x] **else** poblacion_elegida[i] = *P*[y]
end for
return poblacion_elegida

3.2. Cruce

Se emplean dos operadores de cruce distintos, cruce uniforme y cruce por segmento fijo, para generar los nuevos individuos. Estos operadores reciben como parámetros dos padres y devuelve un nuevo cromosoma generado a partir de estos.
Definimos una función auxiliar que recibe como parámetros una población y el número total de cruces a realizar. Ésta llama a las funciones de cruce dos veces por cada pareja seleccionada y devuelve una nueva población con todos los hijos generados.

3.2.1. Cruce uniforme

En el cruce uniforme generamos un nuevo cromosoma combinando la mitad de los genes de un padre, seleccionados de forma aleatoria, con la mitad de los genes del otro padre.

```
function cruce_uniforme
```

Input: Cromosoma *P1*, Cromosoma *P2*.

Cromosoma hijo = **new** *Cromosoma*
bool cluster_vacio = true

```
while cluster_vacio == true do:
        cluster_vacio = false
        int cuenta[k] = [0, …, 0]
        // Generamos una lista con P1.size / 2 enteros en el intervalo [0, P1.size)
        int lista_genes[P1.size / 2] = random_int(0, P1.size, P1.size / 2)

        for i in {0, 1, …, P1.size – 1} do:
            if i in lista_genes then hijo.S[i] = P1.S[i] else hijo.S[i] = P2.S[i]
            cuenta[hijo.S[i]] += 1
        end for

        if 0 in cuenta then cluster_vacio = true
end while
return hijo
```

3.2.2. Cruce por segmento fijo

En el cruce por segmento fijo generamos un nuevo cromosoma copiando un segmento continuo de genes de uno de los padres y el resto de genes por asignar se combinan de forma uniforme.

```
function cruce_por_segmento_fijo
Input: Cromosoma P1, Cromosoma P2.

Cromosoma hijo = new Cromosoma
bool cluster_vacio = true
int genes = P1.size

while cluster_vacio == true do:
        cluster_vacio = false
        int cuenta[k] = [0, …, 0]

        int punto_de_inicio = random_int(0, genes), rango = random_int(0, genes)
        int genes_fuera_segmento = genes – rango
        int genes_p1 = 0, genes_p2 = 0 // Cuenta para la parte uniforme

        int aux = (punto_de_inicio + rango) % genes
        if punto_de_inicio + rango > genes then
            * CONDICIÓN * =  (i >= punto_de_inicio or i < aux)
        else
            * CONDICIÓN * =  (i >= punto_de_inicio and i < (punto_de_inicio + rango))
        end if

        for i in {0, 1, …, P1.size – 1} do:
            // Segmento fijo // * CONDICIÓN * se sustituye por la asignación de arriba
            if * CONDICIÓN * then hijo.S[i] = P1.S[i]
```

```
            // Combinación uniforme
            else
                if genes_p1 > genes_fuera_segmento / 2 then hijo.S[i] = P2.S[i]
                elif genes_p2 > genes_fuera_segmento / 2 then hijo.S[i] = P1.S[i]
                else elegir de forma aleatoria el padre del que copiar el gen y sumar 1 a genes_pX
            end else
        cuenta[hijo.S[i]] += 1
        end for

        if 0 in cuenta then cluster_vacio = true
end while
return hijo
```

3.3. Mutación

Es necesario definir dos operadores de mutación, uno basado en la esperanza matemática y otro basado en una probabilidad. Ésto se debe a que en los algoritmos **AGG** el número de genes que pueden mutar es mucho mayor que en el caso de los **AGE** ya que, en éste último, sólo mutan dos cromosomas. Para los **AGG** usaremos la mutación basada en la esperanza matemática para reducir la cantidad de números aleatorios a generar, mientras que en el caso de los **AGE** utilizaremos la mutación basada en una probabilidad dada.

3.3.1. Mutación basada en la esperaza matemática

En vez de generar un número aleatorio por cada gen para ver si éste muta o no, ésta versión del operador de mutación recibe el número de genes a mutar, *n*, y genera sólo esa cantidad de números aleatorios.

function *mutacion_esperanza_matematica*

Input: Población *P*, Número de genes a mutar *n*, Número de clusters *k*.

```
int genes_mutados = 0

while genes_mutados < n do:
        int gen = random_int(0, P.size * P[0].S.size)
        int x = gen / P[0].S.size, y = gen % P[0].S.size

        int antiguo_cluster = P[x].S[y], nuevo_cluster = antiguo_cluster
        while nuevo_cluster == antiguo_cluster do:
            nuevo_cluster = random_int(0, k)
        end while
        P[x].S[y] = nuevo_cluster
        if antiguo_cluster not in P[x].S then P[x].S[y] = antiguo_cluster
        else P[x].f = -1, genes_mutados += 1 // f = -1 para recalcular f
end while
return P
```

3.3.2. Mutación con probabilidad

En este caso tenemos como parámetro la probabildad de que mute un gen. Calculamos la probabilidad de que mute un cromosoma como el producto de la probabildad de que mute un gen por el número de genes en un cromosoma. Generamos un número aleatorio para ver si muta el cromosoma y, en caso afirmativo, generamos otro número aleatorio para ver que gen mutar.

---

function *mutacion_probabilidad*

Input: Población *P*, Probabilidad de que un gen mute *Pm*, Número de clusters *k*.

double probabilidad_mutar_cromosoma = *Pm* * *P*[0].S.size

for cromosoma **in** *P* **do:**
    if *random_float*(0, 1) <= probabilidad_mutar_cromosoma **then**
        int gen = *random_int*(0, cromosoma.S.size)

        int antiguo_cluster = cromosoma.S[gen], nuevo_cluster = antiguo_cluster
        while nuevo_cluster == antiguo_cluster **do:**
            nuevo_cluster = *random_int*(0, *k*)
        end while
        cromosoma.S[gen] = nuevo_cluster

        if antiguo_cluster **not** in cromosoma.S **then** cromosoma.S[y] = antiguo_cluster
        else P[x].f = -1 // f = -1 para recalcular f
    end if

end while
return *P*

3.4. Elitismo y reemplazamiento

En los **AGG** la nueva población generada reemplaza a la población inicial. Es por esto que realizamos elitismo, que consiste en mantener el mejor cromosoma de la población inicial en la nueva generación. En el caso de los **AGE** no hay preocuparse por mantener la mejor solución hallada ya que los dos hijos generados se enfrentan en un torneo binario con los peores cromosomas de la población inicial.

3.4.1. Elitismo

---

function *elitismo*

Input: Población inicial *P*, Población nueva *P_nueva*.

Ordenar *P* de menor a mayor f
Ordenar *P_nueva* de menor a mayor f
if *P*[0] **not in** *P_nueva* **then** *P_nueva*[*P_nueva.size* – 1] = *P*[0]
return *P_nueva*

3.4.2. Reemplazamiento

function *reemplazamiento*

Input: Población inicial *P*, Población nueva *P_nueva.*

Ordenar *P* de menor a mayor f
Ordenar *P_nueva* de menor a mayor f // *P_nueva* solo tiene dos cromosomas (**AGE**)
int n = *P*.size

if *P_nueva*[1].f < *P*[n-1].f **then** *P*[n-1] = *P_nueva*[1]
elif *P_nueva*[0].f < *P*[n-2].f **then** *P*[n-2] = *P_nueva*[0]
return *P*

3.5. Evaluación

Después de generar una población es necesario evaluar la función objetivo para cada cromosoma de ésta. El operador de evaluación recibe como parámetro la población, la modifica y devuelve el número de evaluaciones de la función objetivo que ha realizado.

function *evaluación*

Input: Población *P*, Conjuto *de datos X, conjuto de restricciones R*, número de clusters *k*, evaluaciones realizadas *evals.*

for cromosoma **in** *P* **do**:
    if cromosoma.f == -1 **then**
        calcula_f(cromosoma.S, *X, R, k*)
        evals += 1
    end if
end for
return *P, evals*

4. Pseudocódigo de los algoritmos

4.1. Algoritmos genéticos generacionales

Implementamos dos algoritmos genéticos generacionales que se diferencian en el tipo de cruce que utilizan.

function *agg-un*

Input: Tamaño de la población *n*, Probabilidad de cruce *Pc*, Probabilidad de mutar un gen *Pm*, número máximo de evaluaciones *max_evals*, Conjuto de datos *X*, conjuto de restricciones *R*, número de clusters *k*.

Cromosoma población[*n*] = *generar_población*(*n*, *X*.size, *k*)
int evaluaciones_hechas = 0

int n_cruces = *Pc* * *n* / 2, n_mutaciones = *Pm* * *n* * *X*.size

población, evaluaciones_hechas = *evaluación*(población, *X, R, k,* evaluaciones_hechas)
while evaluaciones_hechas < *max_evals* **do**:
    Cromosoma nueva_población= *seleccion*(población.size, población)
    Cromosoma hijos = *aux_cruce_uniforme*(nueva_población, n_cruces)
    for i in {0, 1, …, n_cruces} do
        nueva_población[i] = hijos[i]
    end for
    nueva_población = *mutacion_esperanza_matematica*(nueva_población, n_mutaciones, *k*)
    nueva_población = *elitismo*(población, nueva_población)
    población = nueva_población
    población, evaluaciones_hechas = *evaluación*(población, *X, R, k,* evaluaciones_hechas)
end while

Ordenar población de menor a mayor f
return población[0]

function *agg-sf*

Input: Tamaño de la población *n,* Probabilidad de cruce *Pc,* Probabilidad de mutar un gen *Pm,* número máximo de evaluaciones *max_evals,* Conjuto de datos *X*, conjuto de restricciones *R*, número de clusters *k*.

Cromosoma población[*n*] = *generar_población*(*n*, *X*.size, *k*)
int evaluaciones_hechas = 0
int n_cruces = *Pc* * *n* / 2, n_mutaciones = *Pm* * *n* * *X*.size

población, evaluaciones_hechas = *evaluación*(población, *X, R, k,* evaluaciones_hechas)
while evaluaciones_hechas < *max_evals* **do**:
    Cromosoma nueva_población= *seleccion*(población.size, población)
    Cromosoma hijos = *aux_cruce_por_segmento_fijo*(nueva_población, n_cruces)
    for i in {0, 1, …, n_cruces} do
        nueva_población[i] = hijos[i]
    end for
    nueva_población = *mutacion_esperanza_matematica*(nueva_población, n_mutaciones, *k*)
    nueva_población = *elitismo*(población, nueva_población)
    población = nueva_población
    población, evaluaciones_hechas = *evaluación*(población, *X, R, k,* evaluaciones_hechas)
end while

Ordenar población de menor a mayor f
return población[0]

4.2. Algoritmos genéticos estacionarios

Implementamos dos algoritmos genéticos estacionarios que se diferencian en el tipo de cruce que utilizan.

---

function *agg-un*

Input: Tamaño de la población *n*, Probabilidad de mutar un gen *Pm*, número máximo de evaluaciones *max_evals*, Conjuto de datos *X*, conjuto de restricciones *R*, número de clusters *k*.

Cromosoma población[*n*] = *generar_población*(*n*, *X*.size, *k*)
int evaluaciones_hechas = 0

población, evaluaciones_hechas = *evaluación*(población, *X*, *R*, *k*, evaluaciones_hechas)
while evaluaciones_hechas < *max_evals* **do**:
    Cromosoma nueva_población= *seleccion*(2, población)
    Cromosoma hijos = *aux_cruce_uniforme*(nueva_población, 2)
    nueva_población = *mutacion_probabilidad*(nueva_población, *Pm*, *k*)
    población = *reemplazamiento*(población, nueva_población)
    población, evaluaciones_hechas = *evaluación*(población, *X*, *R*, *k*, evaluaciones_hechas)
end while

Ordenar población de menor a mayor f
return población[0]

---

function *agg-sf*

Input: Tamaño de la población *n*, Probabilidad de mutar un gen *Pm*, número máximo de evaluaciones *max_evals*, Conjuto de datos *X*, conjuto de restricciones *R*, número de clusters *k*.

Cromosoma población[*n*] = *generar_población*(*n*, *X*.size, *k*)
int evaluaciones_hechas = 0

población, evaluaciones_hechas = *evaluación*(población, *X*, *R*, *k*, evaluaciones_hechas)
while evaluaciones_hechas < *max_evals* **do**:
    Cromosoma nueva_población= *seleccion*(2, población)
    Cromosoma hijos = *aux_cruce_por_segmento_fijo*(nueva_población, 2)
    nueva_población = *mutacion_probabilidad*(nueva_población, *Pm*, *k*)
    población = *reemplazamiento*(población, nueva_población)
    población, evaluaciones_hechas = *evaluación*(población, *X*, *R*, *k*, evaluaciones_hechas)
end while

Ordenar población de menor a mayor f
return población[0]

4.3. Algoritmos meméticos

Vamos a implementar tres variantes de algoritmos meméticos que hibridan búsqueda local suave, **BLS**, con **AGG-UN**. La hibridación consiste en aplicar la **BLS** sobre un subconjunto de cromosomas de la población cada 10 generaciones del **AGG-UN**.

- AM-(10,1.0). Aplicamos **BLS** sobre todos los cromosomas de la población.
- AM-(10,0.1). Aplicamos **BLS** sobre un subconjunto seleccionado aleatorioamente con una probabilidad de 0.1 para cada cromosoma de la población.
- AM-(10,0.1mej). Aplicamos **BLS** sobre el subconjunto de los 0.1* *N* mejores cromosomas de la población.

La búsqueda local suave recive como parámetro una solución e intenta mejorarla. En esta variante de la BL recorremos el cromosoma de forma aleatoria y por cada punto comprobamos cuál es el mejor cluster que podemos asignarle. Si ninguna asignación para un punto concreto mejora la solución entonces, sumamos 1 al contador de fallos.

En vez de iterar hasta que no encontremos ninguna mejora posible, en la búsqueda local suave recorremos una solución completa sólo una vez, siempre que no se supere el número máximo de fallos. Este tope de fallos se calcula como 0.1 * *n* siendo *n* el número de genes de un cromosoma.

Hay que tener en cuenta que las evaluaciones de la función objetivo que se realizan dentro de la **BLS** cuentan para el total de evaluaciones que se pueden realizar en el algoritmo memético. El límite de fallos es de gran ayuda para evitar malgastar evaluaciones en soluciones dificilmente mejorables.

---

function *BLS*

Input: Cromosoma *C*, Conjuto de datos *X*, conjuto de restricciones *R*, número de clusters *k*, evaluaciones realizadas *evals*.

```
int fallos = 0, i = 0, mejor_cluster, mejor_f, nueva_f, max_fallos = 0.1 * C.S.size
int orden[C.S.size] = [0, 1, …, C.S.size – 1]
orden = shuffle(orden)

while indice < C.S.size and fallos < max_fallos do:
        mejor_f = ∞
        int antiguo_cluster = C.S[i], i = orden[indice]
        for j in {0, 1, …, k} do:
            if j != antiguo_cluster then
                C.S[i] = j
                // Si no deja un cluster vacío
                if antiguo_cluster not in C.S[i] then
                    nueva_f = calcula_f(C.S, X, R, k), evals += 1
                    // Guardamos el mejor cluster posible
                    if nueva_f < mejor_f then
                        mejor_f = nueva_f, mejor_cluster = i
                    end if
                end if
            end if
```

```
            if mejor_f < C.f then
                C.S[i] = mejor_cluster
                C.f = mejor_f
            else
                C.S[i] = antiguo_cluster
                fallos += 1
            end if
        end for
        indice += 1
end while
return C, evals
```

En el código implementado *max_fallos* sólo se calcula una vez.

Para implementar un algoritmo memético tan solo tenemos que generar nuevas poblaciones con **AGG-UN** y cada 10 generaciones llamar a la búsqueda local suave descrita arriba.

function *AM-(10,1.0)*

***Input**: Tamaño de la población n*, Probabilidad de cruce *Pc*, Probabilidad de mutar un gen *Pm*, número máximo de evaluaciones *max_evals*, Conjuto de datos *X*, conjuto de restricciones *R*, número de clusters *k*.

```
Cromosoma población[n] = generar_población(n, X.size, k)
int evaluaciones_hechas = 0
int generacion = 0

población, evaluaciones_hechas = evaluación(población, X, R, k, evaluaciones_hechas)
while evaluaciones_hechas < max_evals do:
        if (generacion % 10) == 0 then
            for cromosoma in población do:
                cromosoma, evaluaciones_hechas = BLS(cromosoma, X, R, k, evaluaciones_hechas)
            end for
        end if
        Generar nueva población con AGG-UN, guardarla en población y actualizar evaluaciones
        generacion += 1
end while

Ordenar población de menor a mayor f
return población[0]
```

function *AM-(10,0.1)*

***Input**: Tamaño de la población n*, Probabilidad de cruce *Pc*, Probabilidad de mutar un gen *Pm*, número máximo de evaluaciones *max_evals*, Conjuto de datos *X*, conjuto de restricciones *R*, número de clusters *k*.

Cromosoma población[*n*] = *generar_población*(*n*, *X*.size, *k*)
int evaluaciones_hechas = 0
int generacion = 0

población, evaluaciones_hechas = *evaluación*(población, *X, R, k,* evaluaciones_hechas)
while evaluaciones_hechas < *max_evals* **do**:
    if (generacion % 10) == 0 **then**
        for cromosoma **in** población **do**:
            if *random_float*(0, 1) <= 0.1 **then**
                cromosoma, evaluaciones_hechas = *BLS(*cromosoma, *X, R, k,*
                evaluaciones_hechas)
            end if
        end for
    end if
    Generar nueva población con **AGG-UN**, guardarla en población y actualizar evaluaciones
    generacion += 1
end while

Ordenar población de menor a mayor f
return población[0]

function *AM-(10,0.1mej)*

Input*: Tamaño de la población n,* Probabilidad de cruce *Pc,* Probabilidad de mutar un gen *Pm,* número máximo de evaluaciones *max_evals,* Conjuto de datos *X,* conjuto de restricciones *R,* número de clusters *k*.

Cromosoma población[*n*] = *generar_población*(*n*, *X*.size, *k*)
int evaluaciones_hechas = 0
int generacion = 0

población, evaluaciones_hechas = *evaluación*(población, *X, R, k,* evaluaciones_hechas)
while evaluaciones_hechas < *max_evals* **do**:
    if (generacion % 10) == 0 **then**
        Ordenar población de menor a mayor f
        for i **in** {0, 1, …, 0.1**n*} **do**:
            población[i], evaluaciones_hechas = *BLS(*población[i], *X, R, k,* evaluaciones_hechas)
        end for
    end if
    Generar nueva población con **AGG-UN**, guardarla en población y actualizar evaluaciones
    generacion += 1
end while

Ordenar población de menor a mayor f
return población[0]

5. Procedimiento considerado para desarrollar la práctica

Para implementar estos algoritmos se ha utilizado C++ tomando como ejemplo el algoritmo genético que se nos proporcionaba. Para realizar una prueba de los algoritmos es necesario compilar el código fuente mediante el uso del makefile. Para ejecutar el programa hay que seguir la siguiente sintaxis:

./test dataset porcentaje-de-restricciones numero-de-clusters semilla algoritmo

un ejemplo de ejecución sería "./test ecoli 10 8 11264 aggun". Los valores que puede tomar *algoritmo* son: aggun, aggsf, ageun, aggsf, am_all, am_random y am_best.

6. Experimentos y análisis de resultados

Se han realizado 5 pruebas por algoritmo utilizando siempre las mismas 5 semillas: 11264, 16438, 75645, 79856 y 96867.

6.1. Resultados de los algoritmos genéticos generacionales

Para la ejecuciones de los **AGG** los parámetros utilizados han sido: tamaño de la población 50, probabilidad de cruce 0.7, probabilidad de mutación 0.001 y un máximo de 100000 evaluaciones de la función objetivo.

AGG-UN 10% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Seed	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
11264	0.669	0.669	0	0.716	0.716	0	14.054	13.835	6	24.348	21.182	118
16438	0.669	0.669	0	0.716	0.716	0	14.054	13.835	6	24.198	22.588	60
75645	0.669	0.669	0	0.716	0.716	0	14.350	10.802	97	24.186	21.208	111
79856	0.669	0.669	0	0.716	0.716	0	14.427	10.880	97	23.833	22.196	61
96867	0.669	0.669	0	0.716	0.716	0	14.054	13.835	6	24.233	21.577	99
Media	**0.669**	**0.669**	**0**	**0.716**	**0.716**	**0**	**14.188**	**12.637**	**42.4**	**24.159**	**21.750**	**89.8**

AGG-SF 10% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Seed	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
11264	0.669	0.669	0	0.716	0.716	0	14.427	10.880	97	24.384	22.157	83
16438	0.669	0.669	0	0.716	0.716	0	14.054	13.835	6	24.095	21.439	99
75645	0.669	0.669	0	0.716	0.716	0	14.054	13.835	6	24.283	21.493	104
79856	0.669	0.669	0	0.716	0.716	0	14.054	13.835	6	23.763	22.019	65
96867	0.669	0.669	0	0.716	0.716	0	14.134	10.624	96	24.139	22.046	78
Media	**0.669**	**0.669**	**0**	**0.716**	**0.716**	**0**	**14.145**	**12.602**	**42.2**	**24.133**	**21.831**	**85.8**

AGG-UN 20% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Seed	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
11264	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	24.425	21.916	187
16438	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	24.119	21.878	167
75645	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.795	21.890	142
79856	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	24.251	21.876	177
96867	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	24.525	22.298	166
Media	**0.669**	**0.669**	**0**	**0.716**	**0.716**	**0**	**14.287**	**14.287**	**0**	**24.223**	**21.972**	**167.8**

AGG-SF 20% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Seed	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
11264	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.723	21.778	145
16438	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	24.069	21.668	179
75645	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.653	21.722	144
79856	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	24.128	21.928	164
96867	0.669	0.669	0	0.716	0.716	0	14.935	10.766	228	24.215	21.894	173
Media	**0.669**	**0.669**	**0**	**0.716**	**0.716**	**0**	**14.417**	**13.583**	**45.6**	**23.958**	**21.798**	**161.0**

Tiempos AGG-UN (segundos)								
	Iris		Rand		Newthyroid		Ecoli	
Seed	10%	20%	10%	20%	10%	20%	10%	20%
11264	2.79	3.65	2.83	3.63	5.29	7.29	12.70	18.17
16438	2.94	3.66	2.85	3.64	5.28	7.29	12.18	17.15
75645	2.79	3.65	2.83	3.64	5.29	7.29	13.06	18.67
79856	2.94	3.78	2.90	3.78	5.63	7.80	12.13	17.05
96867	2.78	3.64	2.82	3.61	5.25	7.24	12.15	17.09
Media	**2.85**	**3.67**	**2.85**	**3.66**	**5.35**	**7.38**	**12.44**	**17.63**

Tiempos AGG-SF (segundos)								
	Iris		Rand		Newthyroid		Ecoli	
Seed	10%	20%	10%	20%	10%	20%	10%	20%
11264	1.86	2.69	1.93	2.68	4.09	6.10	10.31	15.27
16438	1.86	2.70	1.93	2.69	4.10	6.10	10.34	15.29
75645	1.86	2.70	1.92	2.68	4.09	6.09	10.29	15.27
79856	1.86	2.69	1.92	2.68	4.08	6.07	10.28	15.23
96867	1.86	2.70	1.93	2.68	4.10	6.09	10.32	15.26
Media	**1.86**	**2.70**	**1.93**	**2.68**	**4.09**	**6.09**	**10.31**	**15.26**

Como podemos observar, los resultados son muy similares aunque, de media, **AGG-SF** obtiene resultados un poco mejores. Ésto se puede deber a las semillas, ya que, dependiendo de cual elijamos, se favorece más la explotación o la exploración en el cruce por segmento fijo. Además, cabe destacar que **AGG-SF** es más rápido que **AGG-UN**.

6.2. Resultados de los algoritmos genéticos estacionarios

Para la ejecuciones de los **AGE** los parámetros utilizados han sido: tamaño de la población 50, probabilidad de cruce 1.0, probabilidad de mutación 0.001 y un máximo de 100000 evaluaciones de la función objetivo.

AGE-UN 10% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Seed	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
11264	0.669	0.669	0	0.716	0.716	0	14.516	10.895	99	24.017	21.763	84
16438	0.669	0.669	0	0.716	0.716	0	14.470	10.886	98	23.804	21.684	79
75645	0.669	0.669	0	0.716	0.716	0	14.459	10.875	98	24.147	21.974	81
79856	0.669	0.669	0	0.716	0.716	0	14.384	10.874	96	24.014	19.479	169
96867	0.669	0.669	0	0.716	0.716	0	14.427	10.880	97	24.196	22.237	73
Media	**0.669**	**0.669**	**0**	**0.716**	**0.716**	**0**	**14.451**	**10.882**	**97.6**	**24.035**	**21.427**	**97.2**

AGE-SF 10% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Seed	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
11264	0.669	0.669	0	0.716	0.716	0	14.940	10.807	113	24.167	22.449	64
16438	0.669	0.669	0	0.716	0.716	0	14.054	13.835	6	24.085	22.234	69
75645	0.669	0.669	0	0.716	0.716	0	14.427	10.880	97	23.769	22.186	59
79856	0.669	0.669	0	0.716	0.716	0	14.054	13.835	6	24.095	22.056	76
96867	0.669	0.669	0	0.716	0.716	0	14.054	13.835	6	24.193	22.154	76
Media	**0.669**	**0.669**	**0**	**0.716**	**0.716**	**0**	**14.306**	**12.638**	**45.6**	**24.062**	**22.216**	**68.8**

AGE-UN 20% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Seed	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
11264	0.669	0.669	0	0.716	0.716	0	15.558	10.805	260	24.210	21.969	167
16438	0.669	0.669	0	0.716	0.716	0	15.172	10.894	234	23.877	21.851	151
75645	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	24.367	21.912	183
79856	0.669	0.669	0	0.716	0.716	0	15.094	10.871	231	23.793	21.888	142
96867	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	24.158	21.878	170
Media	**0.669**	**0.669**	**0**	**0.716**	**0.716**	**0**	**14.880**	**12.229**	**145.0**	**24.081**	**21.900**	**162.6**

AGE-SF 20% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Seed	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
11264	0.669	0.669	0	0.716	0.716	0	15.080	10.857	231	23.715	21.810	142
16438	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	25.053	21.512	264
75645	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	24.076	21.970	157
79856	0.669	0.669	0	0.716	0.716	0	15.476	10.815	255	24.539	21.722	210
96867	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	24.015	21.708	172
Media	**0.669**	**0.669**	**0**	**0.716**	**0.716**	**0**	**14.684**	**12.907**	**97.2**	**24.280**	**21.744**	**189.0**

Tiempos AGE-UN (segundos)								
	Iris		Rand		Newthyroid		Ecoli	
Seed	10%	20%	10%	20%	10%	20%	10%	20%
11264	3.40	4.31	3.43	4.28	6.10	8.16	14.29	19.24
16438	3.40	4.31	3.45	4.29	6.13	8.14	14.25	19.35
75645	3.40	4.46	3.69	4.76	6.57	8.43	14.78	20.10
79856	3.41	4.31	3.45	4.29	6.13	8.13	14.35	19.25
96867	3.40	4.30	3.43	4.29	6.12	8.11	14.33	19.40
Media	**3.40**	**4.34**	**3.49**	**4.38**	**6.21**	**8.19**	**14.40**	**19.47**

Tiempos AGE-SF (segundos)								
	Iris		Rand		Newthyroid		Ecoli	
Seed	10%	20%	10%	20%	10%	20%	10%	20%
11264	2.25	3.12	2.32	3.11	4.57	6.59	11.59	16.57
16438	2.25	3.14	2.32	3.09	4.58	6.55	11.55	16.50
75645	2.27	3.12	2.32	3.10	4.57	6.58	11.54	16.56
79856	2.26	3.12	2.32	3.10	4.58	6.57	11.55	16.56
96867	2.25	3.12	2.32	3.10	4.59	6.57	11.52	16.45
Media	**2.25**	**3.12**	**2.32**	**3.10**	**4.58**	**6.57**	**11.55**	**16.53**

Al igual que ocurría en los **AGG**, la versión que utiliza el cruce por segmento fijo es más rápida que la de cruce uniforme. En este caso tenemos que **AGE-UN** obtiene peores resultados que **AGE-SF** para el conjunto de datos *newthyroid* mientras que los de *ecoli* son un poco mejores. Como se ha comentado en el caso de los generacionales, ésto se puede deber a las semillas utilizadas. Sería necesario realizar más pruebas para saber que algoritmo es mejor, incluso modificar el cruce por segmento fijo para que siempre favorezca la explotación o la exploración, independientemente de la semilla utilizada.

6.3. Resultados de los algoritmos meméticos

Para la ejecuciones de los **AM** los parámetros utilizados han sido: tamaño de la población 10, probabilidad de cruce 0.7, probabilidad de mutación 0.001 y un máximo de 100000 evaluaciones de la función objetivo.

AM-(10,1.0) 10% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Seed	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
11264	0.669	0.669	0	0.716	0.716	0	14.368	10.894	95	23.922	21.346	96
16438	0.669	0.669	0	0.716	0.716	0	14.186	10.712	95	23.614	21.977	61
75645	0.669	0.669	0	0.716	0.716	0	14.054	13.835	6	23.745	22.055	63
79856	0.669	0.669	0	0.716	0.716	0	14.336	10.716	99	24.024	21.878	80
96867	0.669	0.669	0	0.716	0.716	0	14.054	13.835	6	23.755	22.172	59
Media	**0.669**	**0.669**	**0**	**0.716**	**0.716**	**0**	**14.2**	**11.998**	**60.2**	**23.812**	**21.885**	**71.8**

AM-(10,0.1) 10% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Seed	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
11264	0.669	0.669	0	0.716	0.716	0	14.054	13.835	6	23.231	21.192	76
16438	0.669	0.669	0	0.716	0.716	0	14.054	13.835	6	23.777	22.140	61
75645	0.669	0.669	0	0.716	0.716	0	14.054	13.835	6	23.845	22.128	64
79856	0.669	0.669	0	0.716	0.716	0	14.373	10.789	98	23.628	22.313	49
96867	0.669	0.669	0	0.716	0.716	0	14.427	10.880	97	23.643	21.551	78
Media	**0.669**	**0.669**	**0**	**0.716**	**0.716**	**0**	**14.193**	**12.635**	**42.6**	**23.625**	**21.865**	**65.6**

AM-(10,0.1mej) 10% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Seed	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
11264	0.669	0.669	0	0.716	0.716	0	14.456	10.872	98	23.287	21.677	60
16438	0.669	0.669	0	0.716	0.716	0	14.054	13.835	6	24.754	21.240	131
75645	0.669	0.669	0	0.716	0.716	0	14.054	13.835	6	23.598	22.230	51
79856	0.669	0.669	0	0.716	0.716	0	14.373	10.789	98	23.675	22.226	54
96867	0.669	0.669	0	0.716	0.716	0	14.944	10.812	113	24.187	22.202	74
Media	**0.669**	**0.669**	**0**	**0.716**	**0.716**	**0**	**14.376**	**12.028**	**64.2**	**23.9**	**21.915**	**74**

AM-(10,1.0) 20% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Seed	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
11264	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.571	21.867	127
16438	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.996	21.877	158
75645	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.965	21.845	158
79856	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.294	21.805	111
96867	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	24.078	21.904	162
Media	**0.669**	**0.669**	**0**	**0.716**	**0.716**	**0**	**14.287**	**14.287**	**0**	**23.781**	**21.860**	**143.2**

AM-(10,0.1) 20% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Seed	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
11264	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.855	21.923	144
16438	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.218	21.608	120
75645	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.744	21.880	139
79856	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.812	21.867	145
96867	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.646	21.687	146
Media	**0.669**	**0.669**	**0**	**0.716**	**0.716**	**0**	**14.287**	**14.287**	**0.0**	**23.655**	**21.793**	**138.8**

AM-(10,0.1mej) 20% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Seed	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
11264	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.830	21.831	149
16438	0.669	0.669	0	0.716	0.716	0	15.066	10.880	229	26.682	24.684	149
75645	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.069	19.608	258
79856	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.860	21.888	147
96867	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.841	21.937	142
Media	**0.669**	**0.669**	**0**	**0.716**	**0.716**	**0**	**14.443**	**13.606**	**45.8**	**24.257**	**21.989**	**169**

Tiempos AM-(10,1.0) (segundos)								
	Iris		Rand		Newthyroid		Ecoli	
Seed	10%	20%	10%	20%	10%	20%	10%	20%
11264	1.77	2.57	1.83	2.57	3.99	5.96	9.96	15.09
16438	1.76	2.57	1.83	2.57	3.98	5.97	9.92	15.02
75645	1.76	2.58	1.82	2.58	3.99	5.93	10.22	14.99
79856	1.76	2.58	1.82	2.57	3.96	5.94	9.91	15.07
96867	1.76	2.57	1.83	2.56	3.98	5.98	10.11	15.15
Media	**1.76**	**2.57**	**1.83**	**2.57**	**3.98**	**5.96**	**10.02**	**15.06**

Tiempos AM-(10,0.1) (segundos)								
	Iris		Rand		Newthyroid		Ecoli	
Seed	10%	20%	10%	20%	10%	20%	10%	20%
11264	2.39	3.23	2.43	3.23	4.64	6.68	10.29	15.22
16438	2.38	3.24	2.44	3.24	4.66	6.66	10.29	15.23
75645	2.38	3.23	2.43	3.24	4.65	6.67	10.30	15.15
79856	2.37	3.22	2.43	3.24	4.66	6.67	10.30	15.29
96867	2.39	3.25	2.44	3.22	4.67	6.68	10.28	15.19
Media	**2.38**	**3.23**	**2.43**	**3.23**	**4.65**	**6.67**	**10.29**	**15.22**

Tiempos AM-(10,0.1mej) (segundos)								
	Iris		Rand		Newthyroid		Ecoli	
Seed	10%	20%	10%	20%	10%	20%	10%	20%
11264	2.38	3.24	2.44	3.23	4.67	6.67	10.31	15.19
16438	2.39	3.24	2.45	3.22	4.66	6.67	10.26	15.22
75645	2.39	3.26	2.44	3.24	4.68	6.63	10.27	15.14
79856	2.38	3.25	2.44	3.23	4.67	6.63	10.29	15.22
96867	2.38	3.25	2.44	3.23	4.67	6.65	10.29	15.17
Media	**2.39**	**3.25**	**2.44**	**3.23**	**4.67**	**6.65**	**10.28**	**15.19**

Como era de esperar AM-(10,0.1mej) es el peor de las tres variantes. Al aplicar **BLS** sobre una buena solución lo que conseguimos es “malgastar” evaluaciones de la función objetivo ya que es más complicado mejorarla que si lo hiciesemos con una mala solución.

Por otro lado, AM-(10,0.1) obtiene mejores resultados que AM-(10,1.0). Esto se puede deber al número máximo de evaluaciones, es decir, AM-(10,1.0) realiza muchas evaluaciones en las primeras generaciones ya que, al comenzar con soluciones aleatorias, lo más probable es que éstas no sean buenas y la **BLS** acabe recorriendo los cromosomas enteros o casi al completo y como consecuencia, cuando obtiene buenas soluciones se queda sin evaluaciones para poder mejorarlas. Ésto se puede ver reflejado en los tiempos de ejecución, siendo AM-(10,1.0) la variante más rápida de las tres debido a la cantidad de evaluaciones que realiza en las primeras generaciones.

6.4. Resultados generales

Resultados globales en el PAR con 10% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Algoritmo	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
Greedy	1.476	0.857	97.6	2.451	1.485	134	X	X	X	51.905	39.724	454
BL	0.669	0.669	0	0.716	0.716	0	X	X	X	23.929	21.836	78
AGG-UN	0.669	0.669	0	0.716	0.716	0	14.188	12.637	42.4	24.159	21.750	89.8
AGG-SF	0.669	0.669	0	0.716	0.716	0	14.145	12.602	42.2	24.133	21.831	85.8
AGE-UN	0.669	0.669	0	0.716	0.716	0	14.451	10.882	97.6	24.035	21.427	97.2
AGE-SF	0.669	0.669	0	0.716	0.716	0	14.306	12.638	45.6	24.062	22.216	68.8
AM-(10,1.0)	0.669	0.669	0	0.716	0.716	0	14.2	11.998	60.2	23.812	21.885	71.8
AM-(10,0.1)	0.669	0.669	0	0.716	0.716	0	14.193	12.635	42.6	23.625	21.865	65.6
AM-(10,0.1mej)	0.669	0.669	0	0.716	0.716	0	14.376	12.028	64.2	23.9	21.915	74

Resultados globales en el PAR con 20% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Algoritmo	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
Greedy	0.842	0.710	41.4	1.163	0.864	82.8	X	X	X	45.667	38.305	548.8
BL	0.669	0.669	0	0.716	0.716	0	X	X	X	24.291	22.000	170.8
AGG-UN	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	24.223	21.972	167.8
AGG-SF	0.669	0.669	0	0.716	0.716	0	14.145	12.602	42.2	24.133	21.831	85.8
AGE-UN	0.669	0.669	0	0.716	0.716	0	14.880	12.229	145	24.081	21.900	162.6
AGE-SF	0.669	0.669	0	0.716	0.716	0	14.684	12.907	97.2	24.280	21.744	189.0
AM-(10,1.0)	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.781	21.86	143.2
AM-(10,0.1)	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.655	21.793	138.8
AM-(10,0.1mej)	0.669	0.669	0	0.716	0.716	0	14.443	13.606	45.8	24.257	21.989	169

Al comparar todos los algoritmos utilizados en ésta práctica vemos que los resultados de los conjutos *iris* y *rand* son irrelevantes ya que todos obtienen las mismas soluciones. Solo podemos usar *newthyroid* y *ecoli* para compararlos.

Como era de esperar, los algoritmos meméticos obtienen los mejores resultados de todos. Por otro lado, los **AGG** obtienen mejores resultados que los **AGE**, algo esperable también, ya que éstos últimos exploran menos el espacio de búsqueda al generar sólo dos hijos por generación.

Tiempos globales en el PAR (segundos)								
	Iris		Rand		Newthyroid		Ecoli	
Algoritmo	10%	20%	10%	20%	10%	20%	10%	20%
Greedy	0.12	0.15	0.13	0.15	X	X	1.08	1.67
BL	3.81	3.52	3.54	3.73	X	X	164.24	150.95
AGG-UN	2.85	3.67	2.85	3.66	5.35	7.38	12.44	17.63
AGG-SF	1.85	2.70	1.93	2.68	4.09	6.09	10.31	15.26
AGE-UN	3.40	4.34	3.49	4.38	6.21	8.19	14.40	19.47
AGE-SF	2.25	3.12	2.32	3.1	4.58	6.57	11.55	16.53
AM-(10,1.0)	1.76	2.57	1.83	2.57	3.98	5.96	10.02	15.06
AM-(10,0.1)	2.38	3.23	2.43	3.23	4.65	6.67	10.29	15.22
AM-(10,0.1mej)	2.39	3.25	2.44	3.23	4.67	6.65	10.28	15.19

Respecto a los tiempos, las variantes uniformes de los algoritmos genéticos son más lentas que las variantes de segmento fijo, aunque todos los genéticos son más lentos que todas las variantes de los meméticos.

AM-(10,0.1) es el algoritmo que mejores resultados obtiene, mientras que AM-(10,1.0) es el más rápido de todos.