Metaheurística

Práctica 4.b:

Big Bang Big Crunch 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%.

2. Explicación del algoritmo BB-BC

Como el nombre indica, el algoritmo se basa en las teorías del Big Bang y el Big Crunch. Se interpreta la disipación de energía como la aleatoriedad a la hora de buscar soluciones en el espacio de búsqueda mientras que la convergencia se puede ver como la atracción gravitatoria hacia un punto óptimo.

La idea del algoritmo es explorar el espacio de búsqueda utilizando soluciones generadas de forma aleatoria en torno a un punto llamado centro de masa, fase del Big Bang. Después se produce el Big Crunch, la fase de convergencia, a partir de las nuevas soluciones calculamos un nuevo centro de masa que se utilizará en el siguiente Big Bang.

Al principio no tenemos una población de soluciones con la que calcular el centro de masa y es por esto que el primer centro, y sólo el primero. se obtiene de forma aleatoria. El resto se calculará utilizando la siguiente ecuación:

$$x^c = \frac{\sum_{i=1}^{N} \frac{1}{f^i} x^i}{\sum_{i=1}^{N} \frac{1}{f^i}} \tag{1}$$

donde x^c representa el centro de masa, x^i un punto o solución candidata, f^i, *fitness*, el valor de la función objetivo de dicho punto y *N* es el tamaño de la población generada en la fase del Big Bang.

Una vez tenemos el centro de masa, generamos nuevas soluciones distribuyendolas alrededor de éste en todas las direcciones mediante una distribución normal. La fórmula utilizada es la siguiente:

$$x^i_{new} = x^c + \frac{lr}{k} \tag{2}$$

donde *l* es un parámetro para limitar el espacio de búsqueda, *r* es un número aleatorio generado a partir de una distribución normal y *k* es el número de iteraciones realizadas. Gracias a éste parámetro *k* el algoritmo va reduciendo el espacio de búsqueda y converge hacia una solución.

Las fases de Big Bang y Big Crunch se repiten hasta que se llega a un criterio de parada. Los pasos del algoritmo se resumen así:

1. A partir de un centro de masa aleatorio, dentro del espacio de búsqueda, generar *N* nuevas soluciones.
2. Calcular el valor de la función objetivo para cada nueva solución
3. Obtener el nuevo centro de masa utilizando la fórmula (1)
4. Generar nuevas soluciones utilizando (2)
5. Volver al paso 2 si no se ha alcanzado el criterio de parada

3. Adaptación al problema PAR

Para poder aplicar Big Bang Big Crunch a nuestro problema vamos a utilizar los centroides como representantes de una posible solución. BB-BC trabaja con puntos entonces, trabajaremos con puntos de dimensión *k* * *d*, es decir, el número de clusters multiplicado por el número de característica que tienen los puntos en nuestro problema. Tenemos un array de dimensión uno con la siguiente estructura:

c_1^1	c_2^1	...	c_d^1	...	c_1^k	c_2^k	...	c_d^k

Ésta es solo una representación para mostrar que seguimos trabajando con puntos y no desviarnos de la idea original de BB-BC. A la hora de implementar el problema podemos utilizar un vector de vectores para representar los centroides con el fin de facilitar el manejo de los datos sin que afecte al funcionamiento del algoritmo.

El parámetro *l* utilizado para generar nuevas soluciones será la diferencia del valor máximo y el mínimo que puede tomar cada característica, es decir, para $c_1^1, c_1^2, \ldots, c_1^k$ *l* valdrá lo mismo, límite superior menos límite inferior. Respecto al parámetro *r*, se genera un número aleatorio para cada atributo de cada solución.

Para calcular el *fitness* de una solución usamos la misma función objetivo que hemos utilizado en el resto de prácticas, explicada en la sección 4.2.

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

Para calcular la función objetivo de una solución es necesario que todos los puntos estén asignados a un cluster. Puesto que trabajamos con centroides, vamos a utilizar un enfoque *greedy*, explicado en la sección 4.4, para realizar dicha asignación.

La solución que devuelve el algortimo es la mejor encontrada durante toda la ejecución.

4. Herramientas necesarias para el algoritmo

Antes de poder implementar el algoritmo BB-BC tenemos que definir varias herramientas para tratar distintos puntos del problema.

4.1. Esquema de representación de soluciones

La representación de una posible solución a nuestro problema se hace mediante un ***struct*** *Solución.* Éste almacena distinta información que facilita el tratamiento del problema. Tenemos un vector en el que se almacenan los centroides, éste tendrá *k* vectores de tamaño *d.* Además, también guardamos la asignación de clusters en un vector de longitud *N*, tamaño del conjunto de datos, en el que el cluster al que pertenece el punto *i* se representa con un entero entre 0 y *k*. Por último, guardamos el valor de la función objetivo para ésta solución en la variable f.

Struct *Solución* **contains**
    vector<vector<double>> centroides
    vector<int> clusters
    double f
end

4.2. 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.

4.2.1. Cálculo de la desviación general

Para calcular la desviación general de una solución primero debemos calcular 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]
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_euclidea*(*X*[i], *S*.centroides[cluster])

---

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

4.2.2. 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[i][j] == 1 **do:**
        Si *X*[i] y *X*[j] no están en el mismo cluster => restricciones_incumplidas += 1
    elif R[i][j] == -1 **do**:
        Si *X*[i] y *X*[j] están en el mismo cluster => restricciones_incumplidas += 1
    end for
end for
return restricciones_incumplidas

4.2.3. 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_euclidea(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.

4.3. 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 clusters esté vacio.
Ésta función sólo es llamada una vez en todo el algoritmo para generar la solución inicial con la que se realiza el primer Big Bang.

function *generar_solución_aleatoria*

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

```
Bool cluster_vacio = true
Solución Solución

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)
            Solución.clusters[i] = cluster
            cuenta[cluster] += 1
        end for
```

```
        if 0 in cuenta then cluster_vacio = true
end while
Solución.centroides = calcula_centroides(S, X, k)
Solución.f = calcula_f(Solución.clusters)
return S
```

Calculamos los centroides con la siguiente función:

```
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
```

4.4. Asignación de los puntos dados los centroides

Para asignar los puntos a los distintos clusters fijados los centroides vamos a utilizar una técnica *greedy*. Recorremos los puntos siguiendo un orden aleatorio y los asignamos al cluster en el que viole menos restricciones, es decir, calculamos el incremento de *infeasibility* que produce el asignar un punto *x* a todos los clusters posibles y nos quedamos con el mejor. En el caso de que haya varios clusters con el mismo incremento entonces, asignamos el punto al cluster más cercano.

La función con la que calculamos el incremento de la *infeasibility* al asignar un punto *x* a un cluster *c* es la siguiente:

```
function local_infeasibility
Input: asignación de clusters Clusters, punto x, cluster c, conjuto de restricciones R

int infeasibility = 0

for i in {0, 1, …, Clusters.size} do:
        if R[x][i] == -1 and Clusters[i] == c then infeasibility += 1
        elif  R[x][i] == 1 and (Clusters[i] != c and Clusters[i] != -1) then infeasibility += 1
end for
return infeasibility
```

Y el pseudocódigo para asignar todos los puntos quedaría así:

function *asignar_clusters*

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

```
int clusters[X.size] = [-1, -1, …, -1] // -1 significa que no está todavía asignado a un cluster
int orden[X.size] = random_shuffle([0, 1, …, X.size])

for i in orden do:
        int minimo = ∞, int minimos = []

        // Calcular los incrementos de infeasibility al añadir el punto i al cluster c
        for c in {0, 1, …, k} do:
            infeasibility = local_infeasibility(clusters, i, c, X, R)
            if infeasibility < minimo then
                minimo = infeasibility
                minimos.clear(), minimos.push(c)
            elif infeasibility == minimo then minimos.push(c)
            end if
        end for

        // Si hay más de un mínimo asignamos al más cercano
        if minimos.size > 1 then
            double distancias[minimos.size]
            for m in minimos do:
                distancias[m] = distancia_euclidea(X[i], S.centroides[m]
            end for
            clusters[i] = minimos[min(distancias)]
        else clusters[i] = minimos[0]
end for
return clusters
```

5. Implementación Big Bang Big Crunch

Una vez definidas todas las herramientas necesarias para la ejecución del algoritmo podemos implementarlo siguiendo el siguiente pseudocódigo:

function *BB-BC*

Input: Tamaño de la población *N*, conjuto de datos *X*, conjuto de restricciones *R*, número de clusters *k*, máximo de evaluaciones *max_evals*, límites de las características *límites*

```
int evaluaciones = 0, iter = 1
Solución punto_inicial = generar_solución_aleatoria(X.size, X, k)
Solución mejor_solución = punto_inicial, población = []
```

```
double centro_de_masa = punto_inicial.centroides

while evaluaciones < max_evals do:
        // Big Bang
        población.clear()
        while población.size != N do:
            Solución nueva_solución
            nueva_solución.centroides = centro_de_masa
            for centroide in nueva_solución.centroides do:
                for i in {0, 1, …, centroide.size} do:
                    centroide[i] += distribución_normal() * (límites[i].max -  límites[i].min) / iter
                end for
            end for
            nueva_solución = mantener_límites(nueva_solución, límites)
            nueva_solución.clusters = asignar_clusters(nueva_solución, X, R, k)
            nueva_solución.f = calcula_f(nueva_solución, X, R, k)
            // Si nueva_solución.f == NaN entonces hay clusters vacíos
            if nueva_solución.f != NaN then población.push(nueva_solución)
        end while

        // Big Crunch
        double nuevo_centro_de_masa[k][X.size] = [[0, 0, …, 0], …, [0, 0, …, 0]]
        double suma = 0
        for s in población do:
            suma += 1 / s.f
            for i in {0, 1, …, k} do:
                nuevo_centro_de_masa[i] = s.centroides[i] / s.f
            end for
        end for
        centro_de_masa = nuevo_centro_de_masa / suma

        población = sort(población)
        if población[0].f < mejor_solución.f then mejor_solución = población[0]
        iter += 1
end while
return mejor_solución
```

6. Mejoras del algoritmo

Big Bang Big Crunch favorece la exploración a la explotación al generar nuevas soluciones de forma aleatoria. Aunque reducir el desplazamiento de las nuevas soluciones al dividir por el número de iteraciones realizadas ayuda a la convergencia, ésto no es suficiente. En espacios de búsqueda grandes, como es el caso del dataset *ecoli*, BB-BC no converge a una buena solución lo suficientemente rápido y es ésta división la que evita que pueda alcanzar buenos resultados ya que limita el espacio de búsqueda.

Para evitar que ésto suceda, o al menos reducir el efecto, modificamos el cálculo del centro de masa. A partir de ahora sólo utilizaremos un porcentaje de las mejores soluciones en cada iteración para calcularlo, acelerando así la convergencia a buenas soluciones. Para ello sustituimos el bucle:

```
for s in población do:
        suma += 1 / s.f
        for i in {0, 1, …, k} do:
            nuevo_centro_de_masa[i] = s.centroides[i] / s.f
        end for
end for
```

por uno que sólo utilice una parte de la población

```
int n_mejores = N * crunch
población = sort(población)
for i in {0, 1, …, n_mejores} do:
        suma += 1 / población[i].f
        for j in {0, 1, …, k} do:
            nuevo_centro_de_masa[j] = población[i].centroides[j] / población[i].f
        end for
end for
```

Añadimos un nuevo parámetro al algoritmo, *crunch*, para indicar el porcentaje de las mejores soluciones de la población que es utilizado durante la fase Big Crunch.

7. Hibridación con búsqueda local

Para mejorar aún más la convergencia del algoritmo vamos a hibridarlo con búsqueda local. Se ha probado a utilizar tanto búsqueda local suave como normal y es la segunda la que mejores resultados ha obtenido. Por tanto, el pseudocódigo de la búsqueda local utilizada es el siguiente:

function *BL*

Input: Solución *Sol*, Conjuto de datos *X*, conjuto de restricciones *R*, número de clusters *k*, máximo de evaluaciones *max_evals*.

```
int evaluaciones = 0
bool mejora = True
while evaluaciones < max_evals and mejora == True do:
        mejora = False

        vector <Int, Int> parejas
        for i in {0, 1, …, X.size} do:
            for j in {0, 1, …, k} do:
                if j != Sol.clusters[i] then parejas.push(<i, j>)
            end for
```

```
        end for
        parejas = shuffle(parejas)

        int indice = 0
        while indice < parejas.size and mejora == False do:
            int antiguo_cluster = Sol.clusters[parejas[indice].i]
            Sol.clusters[parejas[indice].i] = parejas[indice].j
            if antiguo_cluster in Sol.clusters then
                double nueva_f = calcula_f(Sol.clustesr, X, R,  k)
                if nueva_f < Sol.f then
                    Sol.f = nueva_f
                    mejora = True
                else
                    Sol.clusters[parejas[indice].i] = antiguo_cluster
                end if
            end if
            indice++
        end for
end while
return Sol
```

Volvemos a modificar el bucle anterior para ahora incluir búsqueda local. Aplicamos cada 10 iteraciones búsqueda local sobre las soluciones que se van a utilizar en el cálculo del nuevo centro de masa, es decir, intentamos mejorar las soluciones que en un principio ya son buenas. Éste cambio ayuda a la velocidad de convergencia y se puede observar claramente en los resultados.

```
int n_mejores = N * crunch
población = sort(población)
for i in {0, 1, …, n_mejores} do:
        if iter % 10 == 0 then población[i] = BL(población[i], X, R, k, 10000)
        suma += 1 / población[i].f
        for j in {0, 1, …, k} do:
            nuevo_centro_de_masa[j] = población[i].centroides[j] / población[i].f
        end for
end for
```

8. Análisis de los experimentos realizados

Se han realizado 5 pruebas por variante de BB-BC utilizando siempre las mismas 5 semillas. En todas se ha utilizado un tamaño de población de 30 soluciones y un máximo de 100000 de evaluaciones de la función objetivo. En el caso de las variantes con la mejora de convergencia, se usa un 20% de las mejores soluciones para calcular el nuevo centro de masa. La búsqueda local ha sido limitada a 10000 evaluaciones de la función objetivo.

Se han realizado pruebas con otras configuraciones de parámetros, distintos tamaños de población y otros porcentajes de soluciones seleccionadas, siendo la configuración ya mencionada la que mejores

resultados ha obtenido. El límite de evaluaciones es el mismo utilizado en prácticas anteriores para poder hacer una comparación con los algoritmos de éstas.

8.1. Resultados BB-BC

BB-BC 10% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Seed	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
11264	1.095	1.095	0	1.184	1.184	0	22.804	21.817	27	51.437	49.103	87
16438	1.299	1.249	8	0.831	0.831	0	25.800	23.569	61	52.847	48.474	163
75645	1.227	1.227	0	1.300	1.300	0	21.532	20.289	34	50.200	48.349	69
79856	0.960	0.960	0	0.838	0.838	0	24.763	24.763	0	55.452	50.703	177
96867	1.138	1.075	10	1.204	1.204	0	22.788	22.679	3	50.190	46.595	134
Media	**1.144**	**1.121**	**3.6**	**1.071**	**1.071**	**0**	**23.537**	**22.623**	**25**	**52.025**	**48.645**	**126**

BB-BC 20% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Seed	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
11264	1.076	1.076	0	1.170	1.170	0	20.438	20.438	0	51.732	50.699	77
16438	1.330	1.330	0	0.883	0.883	0	28.831	28.831	0	51.521	49.804	128
75645	1.153	1.153	0	1.153	1.153	0	20.377	20.286	5	48.192	47.105	81
79856	0.981	0.981	0	0.806	0.806	0	23.683	23.683	0	52.274	50.409	139
96867	1.169	1.112	18	1.162	1.162	0	21.649	21.557	5	48.123	47.144	73
Media	**1.142**	**1.130**	**3.6**	**1.035**	**1.035**	**0**	**22.995**	**22.959**	**2**	**50.368**	**49.032**	**99.6**

Los resultados del algoritmo sin ninguna modificación no son especialmente buenos. Para los dataset más sencillos, *iris* y *rand*, consigue minizar bien la *infeasibility* en casi todos los casos, aun así, son problemas fáciles que no resuelve a la perfección el 100% de los casos como otros algoritmos vistos en prácticas anteriores. Además, no obtiene los centroides óptimos, es decir, aunque la *infeasibility* es 0 la tasa C no es mínima, los centroides están desplazados. Respecto a *newthyroid* y *ecoli*, los resultados dejan mucho que desear.

Tiempos BB-BC (segundos)								
	Iris		Rand		Newthyroid		Ecoli	
Seed	10%	20%	10%	20%	10%	20%	10%	20%
11264	14.36	19.35	14.33	19.34	27.86	38.81	112.30	147.60
16438	14.38	19.62	14.38	19.35	28.07	38.69	112.80	147.70
75645	14.44	19.39	14.35	19.34	28.10	38.85	114.10	150.60
79856	14.35	19.35	14.34	19.34	28.14	39.00	113.00	147.30
96867	14.38	19.40	14.26	19.22	27.79	38.69	112.90	147.00
Media	**14.38**	**19.42**	**14.33**	**19.32**	**27.99**	**38.81**	**113.02**	**148.04**

8.2. Resultados BB-BC mejorado

BB-BC mejorado 10% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Seed	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
11264	0.668	0.668	0	0.714	0.714	0	14.451	14.231	6	40.067	39.102	36
16438	0.668	0.668	0	0.714	0.714	0	14.346	14.127	6	39.354	38.280	40
75645	0.668	0.668	0	0.714	0.714	0	14.768	14.695	2	40.132	39.166	36
79856	0.668	0.668	0	0.714	0.714	0	14.358	14.139	6	40.515	40.220	11
96867	0.668	0.668	0	0.714	0.714	0	14.368	14.148	6	38.939	37.866	40
Media	**0.668**	**0.668**	**0**	**0.714**	**0.714**	**0**	**14.458**	**14.268**	**5.2**	**39.801**	**38.927**	**32.6**

BB-BC mejorado 20% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Seed	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
11264	0.668	0.668	0	0.714	0.714	0	14.211	14.211	0	38.186	37.489	52
16438	0.668	0.668	0	0.714	0.714	0	14.212	14.212	0	40.237	40.062	13
75645	0.668	0.668	0	0.714	0.714	0	14.258	14.258	0	39.520	38.622	67
79856	0.668	0.668	0	0.714	0.714	0	14.211	14.211	0	38.506	37.339	87
96867	0.668	0.668	0	0.714	0.714	0	14.211	14.211	0	36.625	36.021	45
Media	**0.668**	**0.668**	**0**	**0.714**	**0.714**	**0**	**14.220**	**14.220**	**0**	**38.615**	**37.907**	**52.8**

Podemos ver una clara mejora en todos los datasets especialmente en *newthyroid* que el algoritmo ha pasado de tener resultados muy lejos de ser buenos a resultados casi óptimos. Ahora si es capaz de alcanzar el óptimo en todos los casos para los conjuntos de datos sencillos. Aunque lo resultados han mejorado respecto al algoritmo original, seguimos teniendo malos resultados para problemas más complejos como *ecoli*.

Tiempos BB-BC mejorado (segundos)								
	Iris		Rand		Newthyroid		Ecoli	
Seed	10%	20%	10%	20%	10%	20%	10%	20%
11264	14.36	19.24	14.27	19.34	28.17	38.98	113.10	148.50
16438	14.65	19.51	14.34	19.34	28.16	38.65	113.20	147.90
75645	14.33	19.37	14.34	19.32	28.00	38.93	115.00	146.40
79856	14.35	19.35	14.34	19.37	28.14	38.98	114.00	147.50
96867	14.35	19.38	14.35	19.35	28.16	38.50	117.10	146.70
Media	**14.41**	**19.37**	**14.33**	**19.34**	**28.13**	**38.81**	**114.48**	**147.40**

8.3. Resultados hibridación con búsqueda local

Se han realizado pruebas de la hibridación con y sin la mejora del algoritmo original.

BB-BC + BL 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	23.526	21.917	60
16438	0.669	0.669	0	0.715	0.715	0	14.054	13.835	6	23.575	21.348	83
75645	0.669	0.669	0	0.716	0.716	0	14.054	13.835	6	23.168	21.719	54
79856	0.669	0.669	0	0.716	0.716	0	14.054	13.835	6	23.456	21.765	63
96867	0.669	0.669	0	0.715	0.715	0	14.054	13.835	6	23.597	22.256	50
Media	**0.669**	**0.669**	**0**	**0.716**	**0.716**	**0**	**14.054**	**13.835**	**6**	**23.464**	**21.801**	**62.0**

BB-BC + BL 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	23.137	21.635	112
16438	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.133	21.671	109
75645	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.265	21.696	117
79856	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.251	21.923	99
96867	0.669	0.669	0	0.715	0.715	0	14.287	14.287	0	23.151	20.992	161
Media	**0.669**	**0.669**	**0**	**0.716**	**0.716**	**0**	**14.287**	**14.287**	**0**	**23.188**	**21.583**	**119.6**

BB-BC mejorado + BL 10% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Seed	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
11264	0.668	0.668	0	0.714	0.714	0	14.054	13.835	6	23.043	21.326	64
16438	0.668	0.668	0	0.714	0.714	0	14.054	13.835	6	23.173	21.536	61
75645	0.668	0.668	0	0.714	0.714	0	14.054	13.835	6	22.161	19.559	97
79856	0.668	0.668	0	0.714	0.714	0	14.054	13.835	6	23.389	21.940	54
96867	0.668	0.668	0	0.714	0.714	0	14.054	13.835	6	23.393	21.327	77
Media	**0.668**	**0.668**	**0**	**0.714**	**0.714**	**0**	**14.054**	**13.835**	**6**	**23.032**	**21.138**	**70.6**

BB-BC mejorado + BL 20% restricciones												
	Iris			Rand			Newthyroid			Ecoli		
Seed	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea	f	tasa_C	infea
11264	0.668	0.668	0	0.714	0.714	0	14.226	14.226	0	23.256	21.807	108
16438	0.668	0.668	0	0.714	0.714	0	14.234	14.234	0	23.209	21.774	107
75645	0.668	0.668	0	0.714	0.714	0	14.228	14.228	0	23.211	21.936	95
79856	0.668	0.668	0	0.714	0.714	0	14.223	14.223	0	22.888	21.587	97
96867	0.668	0.668	0	0.714	0.714	0	14.226	14.226	0	22.978	21.677	97
Media	**0.668**	**0.668**	**0**	**0.714**	**0.714**	**0**	**14.227**	**14.227**	**0**	**23.108**	**21.756**	**100.8**

Como podemos observar, el uso de búsqueda local ayuda tanto a la convergencia a buenas soluciones que casi no hay diferencia entre usar, o no, la selección de las mejores soluciones para calcular el centro

de masa. No obstante, se obtienen mejores resultados si hacemos uso de ésta mejora. Los resultados obtenidos son muy buenos para todos los datasets en comparación con los que teniamos antes.

Tiempos BB-BC + BL (segundos)								
	Iris		Rand		Newthyroid		Ecoli	
Seed	10%	20%	10%	20%	10%	20%	10%	20%
11264	2.39	2.72	2.25	2.88	4.59	7.33	36.40	51.30
16438	2.50	2.96	2.41	2.95	4.67	7.39	37.91	54.32
75645	2.26	2.86	2.38	2.75	5.14	8.55	35.68	51.16
79856	2.48	2.69	2.46	2.74	4.61	8.09	35.64	50.44
96867	2.43	2.73	2.27	2.78	5.84	9.07	35.73	51.82
Media	**2.41**	**2.79**	**2.35**	**2.82**	**4.97**	**8.09**	**36.27**	**51.81**

Tiempos BB-BC mejorado + BL (segundos)								
	Iris		Rand		Newthyroid		Ecoli	
Seed	10%	20%	10%	20%	10%	20%	10%	20%
11264	3.665	4.912	3.666	4.642	5.768	8.31	15.14	22.42
16438	3.607	4.699	3.714	4.648	5.832	8.443	15.9	22.01
75645	3.6	4.626	3.672	4.653	5.728	8.418	15.15	20.48
79856	3.6	4.651	3.744	4.969	5.587	8.53	14.9	20.26
96867	3.64	4.652	3.684	4.663	5.716	8.244	14.8	20.34
Media	**3.62**	**4.71**	**3.70**	**4.72**	**5.73**	**8.39**	**15.18**	**21.10**

Gracias a la hibridación ahora obtenemos muy buenos resultados en todos los datasets y además conseguimos reducir el tiempo de ejecución del algoritmo.

8.4. Comparativa de las distintas variantes de BB-BC

Las siguientes tablas recogen los resultados de todas las varientes probadas del algoritmo Big Bang Big Crunch. En éstas se pueden ver claramente como mejoran los resultados lo cambios propuestos.

Resultados variantes BB-BC 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
BB-BC	1.144	1.121	3.6	1.071	1.071	0	23.537	22.623	25	52.025	48.645	126.0
BB-BC Mej	0.668	0.668	0	0.714	0.714	0	14.458	14.268	5.2	39.801	38.927	32.6
BB-BC + BL	0.669	0.669	0	0.715	0.715	0	14.054	13.835	6	23.464	21.801	62
BB-BC Mej + BL	0.668	0.668	0	0.714	0.714	0	14.054	13.835	6	23.032	21.138	70.6

Resultados variantes BB-BC 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
BB-BC	1.142	1.130	3.6	1.035	1.035	0	22.995	22.959	2	50.368	49.032	99.6
BB-BC Mej	0.668	0.668	0	0.714	0.714	0	14.220	14.220	0	38.614	37.907	52.8
BB-BC + BL	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.188	21.583	119.6
BB-BC Mej + BL	0.668	0.668	0	0.714	0.714	0	14.227	14.227	0	23.108	21.756	100.8

Tiempos variantes BB-BC en el PAR (segundos)								
	Iris		Rand		Newthyroid		Ecoli	
Algoritmo	10%	20%	10%	20%	10%	20%	10%	20%
BB-BC	14.38	19.42	14.33	19.32	27.99	38.81	113.02	148.04
BB-BC Mej	14.41	19.37	14.33	19.34	28.13	38.81	114.48	147.40
BB-BC + BL	2.41	2.79	2.35	2.82	4.97	8.09	36.27	51.81
BB-BC Mej + BL	3.62	4.71	3.70	4.72	5.73	8.39	15.18	21.10

8.5. Comparativa con los algoritmos vistos en otras prácticas

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
ES	0.669	0.669	0	0.716	0.716	0	14.251	13.227	28	23.968	21.360	97.2
BMB	0.669	0.669	0	0.716	0.716	0	14.054	13.835	6	26.373	22.649	138.8
ILS-BL	0.669	0.669	0	0.716	0.716	0	14.258	12.063	60	23.566	21.57	74.4
ILS-ES Cauchy	0.669	0.669	0	0.746	0.722	3.4	14.256	13.371	24.2	61.269	35.475	961.4
ILS-ES Prop	0.669	0.669	0	0.755	0.725	4.2	14.101	13.874	6.2	24.815	22.282	94.4
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
BB-BC	1.144	1.121	3.6	1.071	1.071	0	23.537	22.623	25	52.025	48.645	126.0
BB-BC Mej	0.668	0.668	0	0.714	0.714	0	14.458	14.268	5.2	39.801	38.927	32.6
BB-BC + BL	0.669	0.669	0	0.715	0.715	0	14.054	13.835	6	23.464	21.801	62
BB-BC Mej + BL	0.668	0.668	0	0.714	0.714	0	14.054	13.835	6	23.032	21.138	70.6

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	549
BL	0.669	0.669	0	0.716	0.716	0	X	X	X	24.291	22.000	171
ES	0.669	0.669	0	0.716	0.716	0	14.829	12.251	141	24.132	21.851	170
BMB	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	25.390	22.447	219.4
ILS-BL	0.669	0.669	0	0.716	0.716	0	14.443	13.606	45.8	23.638	21.849	133.4
ILS-ES Cauchy	0.704	0.676	9	0.780	0.725	6.2	14.717	12.914	98.6	60.073	35.038	1866.2
ILS-ES Prop	0.724	0.682	13.2	0.730	0.721	2.6	14.289	14.106	10	24.844	22.118	203.2
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
BB-BC	1.142	1.130	3.6	1.035	1.035	0	22.995	22.959	2	50.368	49.032	99.6
BB-BC Mej	0.668	0.668	0	0.714	0.714	0	14.220	14.220	0	38.614	37.907	52.8
BB-BC + BL	0.669	0.669	0	0.716	0.716	0	14.287	14.287	0	23.188	21.583	119.6
BB-BC Mej + BL	0.668	0.668	0	0.714	0.714	0	14.227	14.227	0	23.108	21.756	100.8

Ignorando los resultados de BB-BC original, si comparamos la versión mejorada podemos ver que obtenemos resultados mejores que el resto de algoritmos que no usan ningún tipo de hibridación para los datasets *iris, rand* y *newthyroid,* el único algoritmo que obtiene un resultado mejor es AGG-SF para *newthyroid*. Por otro lado, para el dataset *ecoli* obtenemos el peor de todos los resultados.

En cualquier caso, si comparamos los algoritmos con hibridación, nuestra versión de BB-BC obtiene los mejores resultados, consiguiendo el valor mínimo de la función objetivo para todos los datasets.

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
ES	0.15	0.20	0.13	0.15	0.28	1.01	2.78	5.53
BMB	0.29	0.40	0.30	0.39	1.16	1.73	10.79	15.75
ILS-BL	0.17	0.25	0.18	0.25	0.71	1.15	10.83	15.89
ILS-ES Cauchy	0.1	0.11	0.09	0.11	0.36	0.56	0.45	0.6
ILS-ES Prop	0.18	0.24	0.17	0.24	0.84	1.19	9.26	13.02
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
BB-BC	14.38	19.42	14.33	19.32	27.99	38.81	113.02	148.04
BB-BC Mej	14.41	19.37	14.33	19.34	28.13	38.81	114.48	147.40
BB-BC + BL	2.41	2.79	2.35	2.82	4.97	8.09	36.27	51.81
BB-BC Mej + BL	3.62	4.71	3.70	4.72	5.73	8.39	15.18	21.10