# B - Anexo de código



El código de este documento es una simplificación del código ejecutable escrito en lenguaje Matlab. Este documento contiene las versiones de los algoritmos desarrolladas en el capítulo 3 de la memoria.

Notas del código Matlab: " &, && " simbolizan AND, " |, || " OR, " ~ " NOT y la comilla ' la operación de transposición de una matriz. Los elementos de una fila de un array pueden estar separados indiferentemente por comas o por espacios en blanco. "a:b" al acceder a un array indican un intervalo desde el elemento en la posición *a* hasta la *b*, ambos incluidos. Si solo aparecen los dos puntos indica que se acceden a todos sus elementos.

# 1. MST

## 1.1 MST v1

```
1  function [clusters, nClusters] =
       clusterizacionMST_v1(baseDatos, f_distancia, b, nMin)
2  %clusterizacionMST Devuelve una lista con los puntos
       clusterizados según el algoritmo MST
3  %   baseDatos es una matriz de tamaño nxd, donde n es el
       número de puntos y d su dimensión
4  %   f_distancia es la función usada para calcular la distancia
       entre 2 puntos
5  %   b es la distancia máxima entre 2 puntos para que estén
       conectados
6  %   nMin es el número mínimo de puntos que tiene que tener un
       clúster para considerarse como tal
7  %   clusters es una lista en la que cada elemento asociado a
       un clúster tiene el valor Ci, si no pertenece a un clúster,
       su valor es 0
8  %   nClusters es el número total de clústeres obtenidos

10 %Se genera la matriz de incidencia triangular superior,
       calculando las distancias para cada par de puntos
11 n = size(baseDatos, 1);
12 matIncidencia = sparse([]);
13 matIncidencia(n, n) = 0;
14 for i = 1:n-1
15     %Se van añadiendo puntos por filas
16     for j = i+1:n
17         matIncidencia(i, j) = f_distancia(baseDatos(i, :),
               baseDatos(j, :));
18     end
19 end
20 matIncidencia(n, :) = zeros(1, n);

22 %GENERAR UN MST A PARTIR DEL GRAFO DE ENTRADA
23 MST = MST_gen_v1(matIncidencia);

25 %ELIMINAR LAS ARISTAS CON Wij>b
26 for i = find(MST>b)
27     MST(i) = 0;
28 end

30 %COMPROBAR LA VALIDEZ DE LOS SUBGRAFOS OBTENIDOS COMO CLÚSTERES
31 clusters = zeros(n, 1);
32 nClusters = 1;
33 for i = find(MST>0)'
```

```
34      %Hay que pasar del indexación lineal de la matriz de MST a
            la indexación del vector clusters
35      nodo = mod(i, n);
36      %Si se encuentra una arista válida no visitada, se lanza
            la búsqueda del clúster
37      %Si uno de los dos nodos está visitado, el otro también lo
            estará
38      if clusters(nodo) ==  = 0 && MST(i)>0
39          [elementos, nElementos] = buscarClustersArbol(MST,
                nodo);
40          if nElementos<nMin
41              %Si el clúster es demasiado pequeño, todas las
                    aristas visitadas se eliminan
42              for j = find(elementos)
43                  MST(j, :) = 0;
44                  MST(:, j) = 0;
45              end
46          else
47              %Si el clúster es válido, se identifican todos los
                    nodos del clúster con su identificador
48              for j = find(elementos)
49                  clusters(j) = nClusters;
50              end
51              %Se añade un clúster al conteo
52              nClusters = nClusters+1;
53          end
54      end
55  end
56  %Se había contado un clúster de más
57  nClusters = nClusters-1;
```

### 1.1.1 MSTgen

```
1   function MST = MST_gen_v1(grafoCompleto)
2   %MST_GEN Función para generar un Minimal Spanning Tree.
        Implementa el algoritmo de "On the Shortest Spanning
        Subtree of a Graph and the Traveling Salesman Problem"
3   %   grafoCompleto es la matriz de incidencia diagonal superior
        con distancias
4   %   devuelve un grafo en forma de matriz sparse de incidencia
        con distancias (contiene solo los valores de las aristas
        que formal el árbol)

6   n = size(grafoCompleto, 1);
7   MST = sparse([]);
8   MST(n, n) = 0;
9   %Se usa la representación en lista de una matriz para ordenar
```

```
    los elementos, y recuperar después su posición
[~, auxSort] = sort(grafoCompleto(:));
puntosList = zeros(n*(n-1)/2, 2);
i = 1;
for indexSort = auxSort(n*(n+1)/2:end)'
    puntosList(i, 1) = mod(indexSort, n)+1;
    puntosList(i, 2) = ceil(indexSort/n);
    i = i+1;
end

%La arista más pequeña siempre forma parte del MST
MST(puntosList(1, 1), puntosList(1, 2)) =
    grafoCompleto(puntosList(1, 1), puntosList(1, 2));
%actual es el índice del punto que se prueba a meter en el MST
actual = 2;
%El MST tendrá n-1 aristas, y la primera ya está dentro
for i = 2:n-1
    %Busca aristas por orden de longitud hasta encontrar una
        válida
    while buscarCiclos(MST, puntosList(actual, :))
        actual = actual+1;
    end
    %Si la siguiente arista no forma un ciclo, se puede añadir
        al MST
    MST(puntosList(actual, 1), puntosList(actual, 2)) =
        grafoCompleto(puntosList(actual, 1), puntosList(actual,
        2));
    actual = actual+1;
end
```

### 1.1.2 BuscarCiclos

```
function res = buscarCiclos(grafo, arista)
%buscarCiclos Comprueba si una arista formaría un ciclo al
    introducirlo en un grafo usando una estrategia de Depth
    First Search
%   grafo es una matriz de incidencia
%   arista es el candidato que se prueba
%   devuelve true si forma un ciclo, false si no

%Se guarda una lista con los nodos visitados, que en la
    primera iteración van a ser solo los nodos que une la arista
n = size(grafo);
visitadosA = zeros(n(1), 1);
visitadosA(arista(1)) = true;
visitadosB = zeros(n(1), 1);
visitadosB(arista(2)) = true;
```

```
14 %La primera iteración se hace fuera del ciclo
15 candidatosA = buscarCandidatosArbol(grafo, arista(1), 0);
16 previosA = candidatosA;
17 candidatosB = buscarCandidatosArbol(grafo, arista(2), 0);
18 previosB = candidatosB;
19 %Mientras las dos búsquedas no hayan terminado, se lanzan las
      búsquedas
20 while size(candidatosA, 2)~ = 0 && size(candidatosB, 2)~ = 0
21     %Búsqueda A
22     newCandidatosA = [];
23     newPreviosA = [];
24     %Para cada candidato, se buscan los hijos no visitados
25     for i = 1:size(candidatosA, 2)
26         auxA = buscarCandidatosArbol(grafo, candidatosA(i),
              previosA(i));
27         newCandidatosA = [newCandidatosA, auxA];
28         %A todos los hijos no visitados se les asigna el mismo
              padre
29         for j = 1:size(auxA, 2)
30             newPreviosA = [newPreviosA, candidatosA(i)];
31         end
32         %Se añade el nodo a visitados
33         visitadosA(candidatosA(i)) = true;
34     end
35     candidatosA = newCandidatosA;
36     previosA = newPreviosA;

38     %Búsqueda B
39     newCandidatosB = [];
40     newPreviosB = [];
41     %Para cada candidato, se buscan los hijos no visitados
42     for i = 1:size(candidatosB, 2)
43         auxB = buscarCandidatosArbol(grafo, candidatosB(i),
              previosB(i));
44         newCandidatosB = [newCandidatosB, auxB];
45         %A todos los hijos no visitados se les asigna el mismo
              padre
46         for j = 1:size(auxB, 2)
47             newPreviosB = [newPreviosB, candidatosB(i)];
48         end
49         %Se añade el nodo a visitados
50         visitadosB(candidatosB(i)) = true;
51     end
52     candidatosB = newCandidatosB;
53     previosB = newPreviosB;

55     %Si ambas búsquedas coinciden en 1 punto, la nueva arista
```

```
        crea un ciclo
56      if any(visitadosA & visitadosB)
57          res = true;
58          return
59      end
60  end
61  %Si se ha recorrido todo el árbol sin encontrar un ciclo, la
        arista puede ser añadida
62  res = false;
```

### 1.1.3 BuscarCandidatosArbol

```
1   function candidatos = buscarCandidatosArbol(grafo, nodo,
        previo)
2   %Devuelve una lista con todos los nodos nuevos a los que se
        puede ir desde nodo en un árbol
3   %   grafo es la matriz de incidencia
4   %   nodo es de donde se busca
5   %   previo es el padre de nodo

7   candidatos = [];
8   %Se mira primero la columna hasta la diagonal
9   for i = 1:nodo-1
10      if grafo(i, nodo)~ = 0
11          if i = = previo
12              %Si se ha encontrado el nodo del que se viene se
                    salta
13              continue
14          end
15          candidatos = [candidatos, i];
16      end
17  end
18  %Cuando se llega a la diagonal, se miran la fila hasta el borde
19  for i = nodo+1:size(grafo, 1)
20      if grafo(nodo, i)~ = 0
21          if i = = previo
22              %Si se ha encontrado el nodo del que se viene se
                    salta
23              continue
24          end
25          candidatos = [candidatos, i];
26      end
27  end
```

## 1.2 MST v2

```
function [clusters, nClusters] = clusterizacionMST_v2(baseDatos, f_distancia, b, nMin)
%clusterizacionMST Devuelve una lista con los puntos clusterizados según el algoritmo MST
%   baseDatos es una matriz de tamaño nxd, donde n es el número de puntos y d su dimensión
%   f_distancia es la función usada para calcular la distancia entre 2 puntos
%   b es la distancia máxima entre 2 puntos para que estén conectados
%   nMin es el número mínimo de puntos que tiene que tener un clúster para considerarse como tal
%   clusters es una lista en la que cada elemento asociado a un clúster tiene el valor Ci, si no pertenece a un clúster, su valor es 0
%   nClusters es el número total de clústeres obtenidos

%Se genera la matriz de incidencia triangular superior, calculando las distancias para cada par de puntos
n = size(baseDatos, 1);
matIncidencia = sparse([]);
matIncidencia(n, n) = 0;

for i = 1:n-1
    %Se van añadiendo puntos por filas
    for j = i+1:n
        aux = f_distancia(baseDatos(i, :), baseDatos(j, :));
        if aux<b
            %Solo se añaden las aristas válidas
            matIncidencia(i, j) = aux;
        end
    end
end

%Comprobar la validez de los subgrafos como clústeres
clusters = zeros(1, n);
nClusters = 1;
for i = 1:n
    %Si se encuentra un nodo no visitado y conectado conectado a otro, se lanza la búsqueda del clúster
    if clusters(i) == 0 && (any(matIncidencia(i, :)>0) || any(matIncidencia(:, i)))
        [elementos, nElementos] = buscarclústersGrafo(matIncidencia, i);
        if nElementos<nMin
            %Si el clúster es demasiado pequeño, todas las
```

```
                aristas visitadas se eliminan
35              for j = find(elementos)
36                  matIncidencia(j, :) = 0;
37                  matIncidencia(:, j) = 0;
38              end
39          else
40              %Si el clúster es válido, se identifican todos los
                    nodos del clúster con su identificador
41              for j = find(elementos)
42                  clusters(j) = nClusters;
43              end
44              %Se añade un clúster al conteo
45              nClusters = nClusters+1;
46          end
47      end
48  end
49  %Se había contado un clúster de más
50  nClusters = nClusters-1;
```

### 1.2.1 BuscarClustersGrafo

```
1   function [cluster, nCluster] = buscarClustersGrafo(grafo, nodo)
2   %buscarClusters Identifica todos los nodos conectados a un
        nodo en un grafo. Para expandir el clúster se usa una
        estrategia de Depth First Search
3   %   grafo es la matriz de incidencia
4   %   nodo es el nodo desde donde se lanza el algoritmo
5   %   devuelve una lista con los puntos del clúster, y el número
        de elementos del clúster encontrados

7   %Se guarda una lista con los nodos visitados, que en la
        primera iteración va a ser solo el nodo de inicio
8   cluster = zeros(1, size(grafo, 1));
9   cluster(nodo) = 1;

11  %La primera iteración se hace fuera del ciclo
12  candidatos = buscarCandidatosGrafo(grafo, nodo, []);
13  previos = candidatos;

15  %Mientras las dos búsquedas no hayan terminado, se lanzan las
        búsquedas
16  while size(candidatos, 2)~ = 0
17      iterCandidatos = [];
18      for i = candidatos
19          %Se añaden todos los candidatos al clúster
20          cluster(i) = 1;
21          %Se buscan los hijos no visitados respecto a la
```

```
            iteración anterior
22          aux = buscarCandidatosGrafo(grafo, i, previos);
23          for j = 1:numel(aux)
24              iterCandidatos(i, j) = aux(j);
25          end
26      end

28      %Se actualian los nodos visitados y los candidatos para la
            siguiente iteración
29      candidatos = [];
30      for j = 1:numel(iterCandidatos)
31          if iterCandidatos(j)~ = 0 && ~
                ismember(iterCandidatos(j), candidatos)
32              %Si se encuentra un nodo nuevo para visitar se
                    añade a la lista para la siguiente iteración
33              candidatos = [candidatos, iterCandidatos(j)];
34              %y a la lista de visitados
35              previos = [previos, iterCandidatos(j)];
36          end
37      end
38  end

40  %Se cuenta el número de candidatos
41  nCluster = numel(previos);
```

### 1.2.2 BuscarCandidatosGrafo

```
1   function candidatos = buscarCandidatosGrafo(grafo, nodo,
        previos)
2   %Devuelve una lista con todos los nodos nuevos a los que se
        puede ir desde nodo en un grafo
3   %   grafo es la matriz de incidencia
4   %   nodo es de donde se busca
5   %   previos es una lista de nodos visitados

7   candidatos = [];
8   %Se mira primero la columna hasta la diagonal
9   for i = 1:nodo-1
10      if grafo(i, nodo)~ = 0
11          if any(i = = previos)
12              %Si se ha encontrado un nodo visitado se salta
13              continue
14          end
15          candidatos = [candidatos, i];
16      end
17  end
18  %Cuando se llega a la diagonal, se miran la fila hasta el borde
```

```
19 for i = nodo+1:size(grafo, 1)
20     if grafo(nodo, i)~ = 0
21         if any(i = = previos)
22             %Si se ha encontrado un nodo visitado se salta
23             continue
24         end
25         candidatos = [candidatos, i];
26     end
27 end
```

# 2. DBSCAN

```
1  function [clusters, nClusters] =
       clusterizacionDBSCAN_v0(baseDatos, b, nMin, minVecinos,
       tipoArbol, splitm)
2  %clusterizacionDBSCAN Función que implementa el algoritmo
       DBSCAN para calcular clústeres en una base de datos
       espacial. La distancia entre puntos se supone que es la
       distancia euclídea
3  %   baseDatos es una matriz de tamaño nxd, donde n es el
       número de puntos y d su dimensión
4  %   b es la distancia máxima entre 2 puntos para que estén
       conectados
5  %   nMin es el número mínimo de puntos que tiene que tener un
       clúster para considerarse como tal
6  %   minVecinos es el número de puntos a los que tiene que
       estar un punto conectado para considerarse core del clúster
7  %   tipoArbol indica el árbol que se va a usar. Puede ser
       Rtree o KDtree
8  %   splitm (m) es el parámetro de partición del R*-Tree, los
       autores recomiendan que sea 0.4
9  %   clusters es una lista en la que cada elemento asociado a
       un clúster tiene el valor Ci, si no pertenece a un clúster,
       su valor es 0
10 %   nClusters es el número total de clústeres obtenidos

12 %La estructura de R*-Tree usada en este algoritmo se ha
       modificado para encajar con el objetivo de su uso. En ese
       sentido, a parte de definir un tamaño mínimo para las
       celdas, el número de puntos para hacer una partición se
       determina a partir del número mínimo de puntos para tener
       un clúster. Se puede ajustar este parámetro para probar
       distintas configuraciones de árbol
13 factorCelda = 1.5;

15 %Se construye primero el R*-Tree a partir de la base de datos
16 switch tipoArbol
17     case "Rtree"
18         Tree = RTree_v0(baseDatos, ceil(factorCelda*nMin),
               splitm, b);
19     case "KDtree"
20         Tree = KDtree_v0(baseDatos, b, ceil(factorCelda*nMin));
21 end

23 %Tamaño de la base de datos
24 tamBaseDatos = size(baseDatos, 1);
25 tamBaseDatosPercent = ceil(tamBaseDatos/20);
```

```
27 %Se incicializa la clasificación de los puntos
28 %   0:  sin clasificar
29 %   -1: ruido
30 %   >0: corresponde al índice del clúster
31 clusters = zeros(1, tamBaseDatos);

33 %Se guarda el número de clústeres obtenidos
34 nClusters = 0;
35 %Se recorren todos los puntos de la base de datos
36 for i = 1:tamBaseDatos
37     %Si el punto está sin clasificar, se busca un clúster a
          partir del punto
38     if clusters(i) == 0
39         [clusters, nClusters] = expandirCluster_v0(baseDatos,
              Tree, clusters, b, nMin, minVecinos, i, nClusters);
40     end
41 end

43 %Se arregla clusters para que coincida con la interfaz del
      resto de programas
44 clusters(clusters<0) = 0;
```

## 2.1 ExpandirCluster

```
1  function [classPuntosNew, nCluster] =
       expandirCluster_v0(baseDatos, Tree, classPuntos, b, nMin,
       minVecinos, startPoint, nCluster, noNoise)
2  %EXPANDIRCLUSTER Aplica el algoritmo de expansión de un
       clúster a partir de un punto
3  %   baseDatos es una matriz de tamaño nxd, donde n es el
       número de puntos y d su dimensión
4  %   RTree es la estructura sobre la que se opera
5  %   classPuntos es el array con la clasificación de los puntos
6  %   b es la distancia máxima entre 2 puntos para que estén
       conectados
7  %   nMin es el número mínimo de puntos que tiene que tener un
       clúster para considerarse como tal
8  %   minVecinos es el número de puntos a los que tiene que
       estar un punto conectado para considerarse core del clúster
9  %   startPoint es el índice del punto sobre el que se expande
       el clúster
10 %   nCluster es el número de clústeres encontrados hasta ahora
11 %   noNoise es un booleano opcional para que, si es true, no
       se etiqueten los puntos visitados como ruido, solo el
       inicial
```

```
12 %   devuelve el array classPuntos y nCluster actualizados
13 %Nota: Este algoritmo funciona ligeramente distinto al
       propuesto por los autores. Si se encuentra un punto
       frontera entre 2 clústeres contiguos agrupa ambos en un
       mismo clúster. Esto se hace por el tipo de clúster que se
       quiere estudiar, de galaxias, en las que los clústeres
       tienen que estar suficientemente bien diferenciados

15 %Preparación variable de salida
16 classPuntosNew = classPuntos;

18 %Se calculan los vecinos del punto de inicio
19 %*La función buscarVecinosRTree es compatible con un KDtree
20 puntosFrontera = buscarVecinosRTree_v0(baseDatos, Tree, b,
       startPoint);
21 %Si no tiene suficientes vecinos como para ser core, el punto
       es parte de ruido
22 if length(puntosFrontera)<minVecinos
23     classPuntosNew(startPoint) = -1;
24     return
25 end
26 %Si es un punto de core, se añade al clúster
27 cluster = startPoint;

29 %Se guarda una lista de puntos visitados
30 visitados = [startPoint puntosFrontera];
31 %Se expande el clúster hasta que no se encuentran puntos nuevos
32 while ~isempty(puntosFrontera)
33     newFrontera = [];
34     %Para cada punto de la frontera se calculan sus vecinos
35     for puntoFrontera = puntosFrontera(:)'
36         vecinosPuntoFrontera =
              buscarVecinosRTree_v0(baseDatos, Tree, b,
              puntoFrontera);
37         %Si es un punto de core, se añade al clúster y su
              vecinos nuevos son la nueva frontera
38         if length(vecinosPuntoFrontera)>minVecinos
39             cluster = [cluster puntoFrontera];
40             newPoints = setdiff(vecinosPuntoFrontera,
                  visitados);
41             %Se añaden todos los puntos nuevos a sus
                  respectivas listas
42             visitados = [visitados newPoints];
43             newFrontera = union(newPoints, newFrontera);
44         %Si no es un punto de core será ruido
45         else
46             %Puede ocurrir que se visite un punto que ya forma
                  parte de otro clúster. En ese caso no se
```

```
                sobreescribirá el dato.
47              if classPuntosNew(puntoFrontera) =  = 0 && ~noNoise
48                  classPuntosNew(puntoFrontera) = -1;
49              end
50          end
51      end
52      puntosFrontera = newFrontera;
53  end

55  %Se cuenta el número de puntos de core del clúster
56  %Si el número de elementos es suficiente, se aumenta el conteo
        de clústeres y se etiquetan sus elementos
57  if length(cluster)>=nMin
58      nCluster = nCluster+1;
59      for i = cluster
60          classPuntosNew(i) = nCluster;
61      end
62  %Si no se tienen suficientes puntos, todos los puntos
        calculados se etiquetan como ruido
63  else
64      if noNoise
65          classPuntosNew(startPoint) = -1;
66      else
67          for i = cluster
68              classPuntosNew(i) = -1;
69          end
70      end
71  end
```

# 3. DTFoF

```
1  function [clusters, nClusters] =
       clusterizacionDualTreeFoF_v11(baseDatos, b, nMin, tipoArbol)
2  %clusterizacionDUALTREEFOF Función que implementa el algoritmo
       de clusterización FoF en un árbol dual con splay. La
       distancia entre puntos se supone que es la distancia
       euclídea
3  %   baseDatos es una matriz de tamaño nxd, donde n es el
       número de puntos y d su dimensión
4  %   b es la distancia máxima entre 2 puntos para que estén
       conectados
5  %   nMin es el número mínimo de puntos que tiene que tener un
       clúster para considerarse como tal
6  %   tipoArbol es una string que indica el tipo de árbol que se
       va a usar. Puede ser "KDtree" o "Rtree". Por defecto se
       hace el KDtree. En caso de hacer un Rtree, el parámetro de
       partición se toma por defecto en 0.4
7  %   clusters es una lista en la que cada elemento asociado a
       un clúster tiene el valor Ci. Si no pertenece a un clúster,
       su valor es 0
8  %   nClusters es el número total de clústeres obtenidos

10 %La estructura de R*-Tree usada en este algoritmo se ha
       modificado para encajar con el objetivo de su uso.
11 %En ese sentido, a parte de definir un tamaño mínimo para las
       celdas, el número de puntos para hacer una partición se
       determina a partir del número mínimo de puntos para tener
       un clúster. Se puede ajustar este parámetro para probar
       distintas configuraciones de árbol.
12 factorCelda = 1.5;

14 %Se genera primero el árbol a partir de los puntos según su
       tipo
15 switch tipoArbol
16     case "Rtree"
17         Tree = RTree_v0(baseDatos, ceil(factorCelda*nMin),
               splitm, b);
18     case "KDtree"
19         Tree = KDtree_v0(baseDatos, b, ceil(factorCelda*nMin));
20 end

22 %Se inicializa la lista de incidencia y el número de clústeres
23 clusters = 1:length(baseDatos);
24 nClusters = 0;

26 %Se aplica la función ENUM para generar una lista con todas
```

```
    las uniones
27 uniones = enum_v1(baseDatos, Tree, 1, 1, b);

29 %Se eliminan las uniones repetidas y se reorganizan para que
    aparezcan primero los índices menores
30 for i = 1:length(uniones)
31     if uniones(i, 1)>uniones(i, 2)
32         aux = uniones(i, 2);
33         uniones(i, 2) = uniones(i, 1);
34         uniones(i, 1) = aux;
35     end
36 end
37 uniones = unique(uniones, "rows");

39 %Se reordenan las uniones para que aparezcan primero en la
    lista las uniones a los puntos con índices menores
40 [~, ordenUniones] = sort(uniones(:, 1));
41 auxUniones = uniones;
42 for i = 1:length(uniones)
43     uniones(i) = auxUniones(ordenUniones(i));
44 end

46 %Se recorren todas las parejas para hacer la operación merge
47 for pareja = 1:length(uniones)
48     [clusters, r] = splay_v1(clusters, uniones(pareja, 1));
49     [clusters, s] = splay_v1(clusters, uniones(pareja, 2));
50     clusters(r) = s;
51 end

53 %Se reordenan todos los clústeres
54 for i = 1:length(clusters)
55     clusters = splay_v1(clusters, i);
56 end

58 %Se recorre clusters para que la salida sea acorde con el
    resto de algoritmos de clusterización
59 %Se guarda un histograma con los clústeres y un lista de sus
    miembros
60 histograma = zeros(1, length(clusters));
61 miembros(length(clusters), length(clusters)) = sparse(0);
    %matriz sparse para ahorrar memoria
62 %Se recorre toda la lista de clusters para contar los puntos
    de cada clúster
63 for i = 1:length(clusters)
64     histograma(clusters(i)) = histograma(clusters(i))+1;
65     miembros(clusters(i), histograma(clusters(i))) = i;
66 end
```

```
68 %Si se tiene un clúster con suficientes miembros, se asignan
       correspondientemente, sino se les asigna 0
69 for i = 1:length(clusters)
70     %No tiene elementos
71     if histograma(i) = = 0
72         continue
73     %Es un cluster
74     elseif histograma(i)> = nMin
75         nClusters = nClusters+1;
76         for j = 1:histograma(i)
77             clusters(miembros(i, j)) = nClusters;
78         end
79     %No es un clúster, pero tiene elementos
80     elseif histograma(i)>0
81         for j = 1:histograma(i)
82             clusters(miembros(i, j)) = 0;
83         end
84     end
85 end
```

## 3.1 Enum

```
1  function pairs = enum_v1(dataBase, Tree, nodoA, nodoB, b)
2  %ENUM Función recursiva que devuelve una lista de todos los
       puntos conectados a una distancia <b y que conectan los
       nodos A y B. La función es una implementación del algoritmo
       1 del paper "A fast algorithm for identifying
       friends-of-friends halos"
3  %   dataBase es una matriz en la que cada fila contiene las
       coordenadas del punto correspondiente
4  %   Tree estructura de árbol donde se realiza la operación
5  %   nodoA y nodoB son los índices de los nodos que se quieren
       conectar
6  %   b es la distancía máxima de enlace
7  %   devuelve una matriz en la que cada fila contiene la pareja
       de puntos a una distancia <b

9  %Si se están buscando parejas dentro de una misma hoja, se
       buscan los puntos conectados
10 if nodoA = = nodoB && Tree{nodoA, 3}(1)<0
11     nPuntos = length(Tree{nodoA, 3});
12     %Si la celda no tiene al menos 2 puntos no hay conexiones
13     if nPuntos = = 1 || nPuntos = = 2
14         pairs = [];
15         return
16     end
```

```
17      pairs = [];
18      nParejas = 1;
19      %Si es una celda pequeña, todos los puntos están
           conectados a su raíz, que es el punto con el índice más
           pequeño
20      if Tree{nodoA, 3}(1) =  = -2
21          raiz = min(Tree{nodoA, 3}(2:end));
22          for i = 2:nPuntos
23              if Tree{nodoA, 3}(i)~ = raiz
24                  pairs(nParejas, :) = [raiz Tree{nodoA, 3}(i)];
25                  nParejas = nParejas+1;
26              end
27          end
28      %Si no es pequeña, hay que comprobar la distancia de cada
           conexión
29      else
30          for i = 2:nPuntos-1
31              for j = i+1:nPuntos
32                  if dist(dataBase, Tree{nodoA, 3}(i),
                       Tree{nodoA, 3}(j))<b
33                      pairs(nParejas, :) = [Tree{nodoA, 3}(i)
                           Tree{nodoA, 3}(j)];
34                      nParejas = nParejas+1;
35                  end
36              end
37          end
38      end
39      return
40  end

42  %Si los dos nodos son hojas con todos sus puntos conectados,
       si se encuentra una conexión se dejan de buscar más parejas
43  %Se conectan las raices de los nodos entre sí
44  if Tree{nodoA, 3}(1) =  = -2 && Tree{nodoB, 3}(1) =  = -2
45      for i = 2:length(Tree{nodoA, 3})
46          for j = 2:length(Tree{nodoB, 3})
47              if dist(dataBase, Tree{nodoA, 3}(i), Tree{nodoB,
                   3}(j))<b
48                  pairs = [min(Tree{nodoA, 3}(2:end))
                       min(Tree{nodoB, 3}(2:end))];
49                  return
50              end
51          end
52      end
53  end

55  %Inicialización de valores
56  pairs = [];
```

```
57 nParejas = 1;

59 %Intercambio del orden si nodo A es una hoja
60 if Tree{nodoA, 3}(1)<0
61     aux = nodoA;
62     nodoA = nodoB;
63     nodoB = aux;
64 end

66 %Si los dos nodos son hojas se calculan las distancias punto a
       punto
67 if Tree{nodoA, 3}(1)<0
68     %Se calculan distancias solo si van a estar
           suficientemente cerca
69     for i = 2:length(Tree{nodoA, 3})
70         for j = 2:length(Tree{nodoB, 3})
71             if dist(dataBase, Tree{nodoA, 3}(i), Tree{nodoB,
                   3}(j))<b
72                 pairs(nParejas, :) = [Tree{nodoA, 3}(i)
                       Tree{nodoB, 3}(j)];
73                 nParejas = nParejas+1;
74             end
75         end
76     end
77 %Se buscan las parejas de nodos en sus hijos
78 %En el nodoA si nodoA no es una hoja
79 %En el nodoB si nodoA es una hoja y nodoB no lo es
80 else
81     if minDistManhattan_v0(Tree, Tree{nodoA, 3}(1), nodoB)<b
82         pairs = [pairs; enum_v0(dataBase, Tree, Tree{nodoA,
               3}(1), nodoB, b)];
83     end
84     if minDistManhattan_v0(Tree, Tree{nodoA, 3}(2), nodoB)<b
85         pairs = [pairs; enum_v0(dataBase, Tree, Tree{nodoA,
               3}(2), nodoB, b)];
86     end
87 end
```

## 3.2 Splay

```
1 function [clusters, raiz] = splay_v1(clusters, punto)
2 %SPLAY Reordena el árbol de tal manera que un punto sea hijo
      de la raíz.
3 %La función es una implementación del algoritmo 2 del paper "A
      fast algorithm for identifying friends-of-friends halos"
4 %   clusters es la estructura de incidencia donde se
```

```
    encuentran los clústeres
%   punto es el punto a reordenar
%   Devuelve el array de incidencia reordenado
%   raiz es el índice de la raíz del punto

raiz = punto;
visitados = [];
%Se escala el árbol hasta la raíz del clúster
while clusters(raiz)~ = raiz
    visitados = [visitados raiz];
    raiz = clusters(raiz);
end
for visitado = visitados
    clusters(visitado) = raiz;
end
```

# 4. ÁRBOLES

## 4.1 KD-tree

```
1  function KDtree = KDtree_v0(dataBase, b, nMax)
2  %KDTREE Función para generar un KD-tree a partir de una base
       de datos
3  %   b es la distancia máxima de enlace entre elementos
4  %   nMin es el número máximo de puntos en una celda
5  %   Devuelve un R*-tree como una matriz de celdas. Cada fila
       corresponde a un nodo del árbol, organizado de la siguiente
       manera:
6  %       KDtree{i, 1} = padre del nodo
7  %       KDtree{i, 2} = límites del nodo. Se dan las
       coordenadas por parejas de min-max en cada dimensión
8  %       KDtree{i, 3} = hijos del nodo. Si el primer elmento es
       negativo, se trata de una hoja, y los índices siguientes
       hacen referencia a los puntos de la base de datos. Si
       además ese número es -2, todos los puntos del nodo están
       conectados entre sí
9  %Nota: Al ser una estructura de árbol idéntica en su
       representación a la del R*-tree, las funciones buscarHoja y
       buscarVecinosRTree funcionan con un KDtree

11 %INICIALIZACIÓN
12 %Dimension de los puntos
13 dim = size(dataBase, 2);
14 %Incialización del árbol
15 KDtree{1, 1} = 1;
16 KDtree{1, 2} = limitesNodo_v0(dataBase, 1:length(dataBase));
17 KDtree{1, 3} = 1:length(dataBase);

19 %Lista de nodos pendientes para dividir
20 pendientes = 1;

22 %Se añade una pequeña mejora para que las celdas sean lo más
       cuadradas posibles
23 %Se ordenan los ejes de mayor a menor respecto del espacio
       total que ocupan los puntos
24 auxTam = zeros(1, dim);
25 for i = 1:dim
26     auxTam(i) = abs(KDtree{1, 2}(2*i)-KDtree{1, 2}(2*i-1));
27 end
28 [~, ordenParticiones] = sort(auxTam);
29 particion = 0;

31 %Ciclo principal
```

```
32 while ~isempty(pendientes)
33     %Se fija el eje de partición del paso
34     particion = mod(particion, dim)+1;
35     ejeParticion = ordenParticiones(particion);

37     %Nodos que dividir para la siguiente iteración
38     newPendientes = [];

40     %Se dividen todas las celdas nuevas del paso anterior que
           no son hojas
41     for nodo = pendientes
42         [KDtree, auxPendientes] = dividirCelda_v0(dataBase,
               KDtree, b, nMax, ejeParticion, nodo);
43         newPendientes = [newPendientes auxPendientes];
44     end

46     %Se actualizan los nodos pendientes de ser divididos
47     pendientes = newPendientes;
48 end
```

### 4.1.1 DividirCelda

```
1  function [KDtree, noHojas] = dividirCelda_v0(dataBase, KDtree,
       b, nMax, eje, padre)
2  %SPLITTREE divide un nodo dado por los puntos en 2 nodos
3  %   dataBase es una matriz en la que cada fila contiene las
       coordenadas del punto correspondiente
4  %   KDtree es la estructura sobre la que se opera
5  %   b es la distancía máxima de enlace
6  %   nMax es el número máximo de puntos en una celda
7  %   eje es el eje sobre el que se va a dividir la celda
8  %   padre es el índice del nodo que se va a dividir
9  %
10 %   devuelve el árbol dividido
11 %   noHojas devuelve un array con el índice de los nodos que
       no son hojas

13 %Se generan los dos nodos hijos
14 indexA = size(KDtree, 1)+1;
15 indexB = size(KDtree, 1)+2;
16 KDtree{indexA, 1} = padre; %Celda con el límite superior en la
       partición
17 KDtree{indexB, 1} = padre; %Celda con el límite inferior en la
       partición

19 %Se divide la celda en 2
20 KDtree{indexA, 2} = KDtree{padre, 2};
```

```
21 KDtree{indexB, 2} = KDtree{padre, 2};
22 auxParticion = (KDtree{padre, 2}(2*eje)+KDtree{padre,
      2}(2*eje-1))/2; %punto medio
23 KDtree{indexA, 2}(2*eje) = auxParticion;
24 KDtree{indexB, 2}(2*eje-1) = auxParticion;

26 %Se inicializan los hijos de las celdas nuevas
27 KDtree{indexA, 3} = [];
28 KDtree{indexB, 3} = [];

30 %Se ordenan los puntos según el eje usando sus referencias
      directamente con sort (ver documentación de Matlab)
31 [~, I] = sort(dataBase(KDtree{padre, 3}, eje));
32 sortedPoints = KDtree{padre, 3}(I);

34 %Se reparten los puntos del nodo padre entre los hijos usando
      la lista ordenada. Se usa una búsqueda dicotómica
35 index = ceil(length(sortedPoints)/2);
36 auxTam = ceil(index/2);
37 while true
38     %Si el índice es 1, todos los puntos pertenecen a la
          partición B
39     if index == 1
40         KDtree{indexB, 3} = [KDtree{indexB, 3} sortedPoints];
41         break
42     %Si el índice es el último, todos los puntos pertenecen a
          la partición A
43     elseif index == length(sortedPoints)
44         KDtree{indexA, 3} = [KDtree{indexA, 3} sortedPoints];
45         break
46     %Se comprueba si se ha caído en el punto de corte
47     elseif dataBase(sortedPoints(index), eje)>= auxParticion
          && dataBase(sortedPoints(index-1), eje)<= auxParticion
48         KDtree{indexA, 3} = [KDtree{indexA, 3}
              sortedPoints(1:index-1)];
49         KDtree{indexB, 3} = [KDtree{indexB, 3}
              sortedPoints(index:end)];
50         break
51     end

53     %Si no se está en el punto de corte, se busca la división
54     if dataBase(sortedPoints(index), eje)>auxParticion
55         index = index-auxTam;
56     else
57         index = index+auxTam;
58     end

60     %Actualización de tal manera que auxTam no sea nunca 0
```

```
61      if auxTam~ = 1
62          auxTam = floor(auxTam/2);
63      end
64  end

66  %Dimensión del problema
67  dim = length(KDtree{padre, 2})/2;

69  %Se calcula el tamaño de las celdas nuevas
70  auxCeldaA = zeros(1, dim);
71  auxCeldaB = zeros(1, dim);
72  for d = 1:dim
73      auxCeldaA(d) = KDtree{indexA, 2}(2*d)-KDtree{indexA,
            2}(2*d-1);
74      auxCeldaB(d) = KDtree{indexB, 2}(2*d)-KDtree{indexB,
            2}(2*d-1);
75  end

77  %Se comprueba el tamaño de la celda y el número de puntos de
        cada celda nueva para ver si son hojas
78  %Si un nodo no es una hoja, se actualiza el valor de salida
        noHojas
79  noHojas = [];
80  if sqrt(auxCeldaA*auxCeldaA')<b
81      KDtree{indexA, 3} = [-2 KDtree{indexA, 3}];
82  elseif index<=nMax
83      KDtree{indexA, 3} = [-1 KDtree{indexA, 3}];
84  else
85      noHojas = indexA;
86  end
87  if sqrt(auxCeldaB*auxCeldaB')<b
88      KDtree{indexB, 3} = [-2 KDtree{indexB, 3}];
89  elseif length(sortedPoints(index:end))<=nMax
90      KDtree{indexB, 3} = [-1 KDtree{indexB, 3}];
91  else
92      noHojas = [noHojas indexB];
93  end

95  %Se actualizan los hijos del nodo actual
96  KDtree{padre, 3} = [indexA indexB];
```

## 4.2 R*-tree

```
1  function RTree = RTree_v0(dataBase, MaxPoints, splitm, b)
2  %RTREE Función que devuelve un R*-tree dado una colección de
       puntos k-dimensional
3  %   dataBase es una matriz en la que cada fila contiene las
       coordenadas del punto correspondiente
4  %   MaxPoints (M) es el número máximo de puntos que puede
       tener una hoja
5  %   splitm (m) es el parámetro de partición, los autores
       recomiendan que sea 0.4
6  %   b es la distancia de enlace máxima
7  %   Devuelve un R*-tree como una matriz de celdas. Cada fila
       corresponde a un nodo del árbol, organizado de la siguiente
       manera:
8  %       RTree{i, 1} = padre del nodo
9  %       RTree{i, 2} = límites del nodo. Se dan las coordenadas
       por parejas de min-max en cada dimensión.
10 %       RTree{i, 3} = hijos del nodo. Si el primer elmento es
       negativo, se trata de una hoja, y los índices siguientes
       hacen referencia a los puntos de la base de datos. Si
       además ese número es -2, todos los puntos del nodo están
       conectados entre sí
11 %Nota: Una ligera modificación respecto el R*-Tree Clásico, es
       que se tiene una condición para dejar de buscar divisiones.
       Cuando la diagonal del rectángulo es menor que la distancia
       de enlace máxima (b), ese rectángulo es una hoja. Todos los
       puntos de esa hoja están conectados entre sí

13 %Inicialización del árbol (matlab asigna cada celda donde
       cabe, darle un tamaño al inicializar no mejora el programa)
14 RTree{1, 1} = 1;
15 RTree{1, 2} = limitesNodo_v0(dataBase, 1:size(dataBase, 1));
16 RTree{1, 3} = 1:size(dataBase, 1);

18 %Se guarda un conteo de los nodos que todavía no son hojas
19 nodosNoSeparados = 1;
20 while ~isempty(nodosNoSeparados)
21     currentNodosNoSeparados = [];
22     %Se llama a splitTree para separar todos los nodos que no
           son hojas en el nivel actual
23     for nodoSeparar = nodosNoSeparados
24         [RTree, aux] = splitTree_v0(dataBase, RTree,
               MaxPoints, splitm, b, nodoSeparar);
25         currentNodosNoSeparados = [currentNodosNoSeparados
               aux];
26     end
27     nodosNoSeparados = currentNodosNoSeparados;
```

```
28 end
```

### 4.2.1 LimitesNodo

```
1  function lim = limitesNodo_v0(dataBase, puntos)
2  %LIMITESNODO Devuelve las dimensiones del rectángulo dado por
       los puntos
3  %   dataBase es una matriz en la que cada fila contiene las
       coordenadas del punto correspondiente
4  %   puntos es un array con los índices de los puntos del nodo
5  %   devuelve un array que contiene los límites mínimo y máximo
       del rectángulo en las sucesivas dimensiones

7  %Se inicializa el array de las dimensiones
8  lim = zeros(1, size(dataBase, 2)*2);
9  for d = 1:size(dataBase, 2)
10     lim(d*2-1) = inf;
11     lim(d*2) = -inf;
12 end

14 %Se buscan los máximos y los mínimos en cada dimensión
15 for punto = puntos
16     for d = 1:size(dataBase, 2)
17         if dataBase(punto, d)<lim(d*2-1)
18             lim(d*2-1) = dataBase(punto, d);
19         end
20         if dataBase(punto, d)>lim(d*2)
21             lim(d*2) = dataBase(punto, d);
22         end
23     end
24 end
```

### 4.2.2 SplitTree

```
1  function [RTree, noHojas] = splitTree_v0(dataBase, RTree,
       MaxPoints, splitm, b, nodo)
2  %SPLITTREE divide un nodo dado por los puntos en 2 nodos
3  %   dataBase es una matriz en la que cada fila contiene las
       coordenadas del punto correspondiente
4  %   RTree es la estructura sobre la que se opera
5  %   MaxPoints es el número máximo de puntos que puede tener
       una hoja
6  %   splitm es el parámetro de partición, los autores
       recomiendan que sea 0.4
7  %   b es la distancia de enlace máxima
```

```
8  %   nodo es el índice del nodo que se va a dividir
9  %   RTree devuelve el árbol dividido
10 %   noHojas devuelve un array con el índice de los nodos que
       no son hojas

12 %Se calcula cual va a ser el eje sobre el que se haga la
       división
13 eje = splitElejirEje_v0(dataBase, RTree, nodo, splitm);
14 %Si el nodo no se puede dividir, se sale de la función
       asignando el nuevo
15 if eje =  = -1
16     auxLim = limitesNodo_v0(dataBase, RTree{nodo, 3});
17     if sqrt(auxLik*auxLim')<b
18         RTree{nodo, 3} = [-2 RTree{nodo, 3}];
19     else
20         RTree{nodo, 3} = [-1 RTree{nodo, 3}];
21     end
22     noHojas = [];
23     return
24 end

26 %Se ordenan los puntos según el eje usando sus referencias
       directamente con sort (ver documentación de Matlab)
27 [~, I] = sort(dataBase(RTree{nodo, 3}, eje));
28 sortedPoints = RTree{nodo, 3}(I);

30 %Número de puntos en el nodo
31 totPoints = length(RTree{nodo, 3});
32 %Inicialización de los vectores de las aristas
33 aristasA = zeros(1, size(dataBase, 2));
34 aristasB = aristasA;

36 %Se calcula la partición óptima
37 %Ya que las particiones óptimas no solapan, se elige la que
       minimiza el volumen
38 minVol = inf;
39 for k = 1:totPoints-2*ceil(totPoints*splitm)+2
40     %Separación de 1:M*m-1+k
41     limitesA = limitesNodo_v0(dataBase,
           sortedPoints(1:ceil(totPoints*splitm)-1+k));
42     %Separación de M*m+k:-1
43     limitesB = limitesNodo_v0(dataBase,
           sortedPoints(ceil(totPoints*splitm)+k:end));
44     %Se calcula la logitud de las aristas de los rectángulos
45     for d = 1:size(dataBase, 2)
46         aristasA(d) = abs(limitesA(2*d-1)-limitesA(2*d));
47         aristasB(d) = abs(limitesB(2*d-1)-limitesB(2*d));
48     end
```

```
49      %Se calcula el volumen
50      if prod(aristasA)+prod(aristasB)<minVol
51          minVol = prod(aristasA)+prod(aristasB);
52          %Se guarda el índice de la partición óptima
53          minVolIndex = k;
54          %Y los límites de las particiones
55          limitesAMin = limitesA;
56          limitesBMin = limitesB;
57      end
58  end

60  %Se generan los dos nodos hijos
61  indexA = size(RTree, 1)+1;
62  indexB = size(RTree, 1)+2;
63  RTree{indexA, 1} = nodo;
64  RTree{indexB, 1} = nodo;
65  RTree{indexA, 2} = limitesAMin;
66  RTree{indexB, 2} = limitesBMin;
67  %Inicialización de noHojas
68  noHojas = [];
69  %Se comprueba a ver si es una hoja por el tamaño de la celda,
        o por el número de puntos
70  if sqrt(limitesAMin*limitesAMin')<b
71      RTree{indexA, 3} = [-2
            sortedPoints(1:ceil(totPoints*splitm)-1+minVolIndex)];
72  elseif
        length(sortedPoints(1:ceil(totPoints*splitm)-1+minVolIndex))<MaxPoints
73      RTree{indexA, 3} = [-1
            sortedPoints(1:ceil(totPoints*splitm)-1+minVolIndex)];
74  else
75      RTree{indexA, 3} =
            sortedPoints(1:ceil(totPoints*splitm)-1+minVolIndex);
76      noHojas = indexA;
77  end
78  if sqrt(limitesBMin*limitesBMin')<b
79      RTree{indexB, 3} = [-2
            sortedPoints(ceil(totPoints*splitm)+minVolIndex:end)];
80  elseif
        length(sortedPoints(ceil(totPoints*splitm)+minVolIndex:end))<MaxPoints
81      RTree{indexB, 3} = [-1
            sortedPoints(ceil(totPoints*splitm)+minVolIndex:end)];
82  else
83      RTree{indexB, 3} =
            sortedPoints(ceil(totPoints*splitm)+minVolIndex:end);
84      noHojas = [noHojas indexB];
85  end
86  %Se actualizan los hijos del nodo actual
87  RTree{nodo, 3} = [indexA indexB];
```

### 4.2.3 SplitElegirEje

```
function splitEje = splitElejirEje_v0(dataBase, RTree, nodo, splitm)
%SPLITELEJIREJE Da el eje sobre el que es más óptimo dividir un nodo. Frente al algoritmo del paper, esta implementación divide un conjunto arbitrariamente grande
%   dataBase es una matriz en la que cada fila contiene las coordenadas del punto correspondiente
%   RTree es la estructura sobre la que se opera
%   nodo es el índice del nodo que se va a dividir
%   splitm es el parámetro de partición, los autores recomiendan que sea 0.4
%   devuelve el eje óptimo. -1 indica que ha habido un error, y el nodo no se puede dividir

%Si ha habido algún error, o el número de puntos es 1, se devuelve -1
splitEje = -1;

%Se incializa el valor del valor mínimo de S
minS = inf;
%Como el conjunto de puntos a dividir puede ser arbitrariamente grande, se escoje adecuadamanete el tamaño de las particiones
totPoints = length(RTree{nodo, 3});
for eje = 1:size(dataBase, 2)
    %Se ordenan los puntos según el eje usando sus referencias directamente con sort (ver documentación de Matlab)
    [~, I] = sort(dataBase(RTree{nodo, 3}, eje));
    sortedPoints = RTree{nodo, 3}(I);

    %Se calcula la suma de todos los márgenes para el eje
    S = 0;
    %Para 2 y 3 dimensiones se usa el criterio del paper de calcular el margen
    %Para dimensiones superiores, por simplicidad, se usa el volumen
    %Esto se hace para todas las M-2M*m+2 divisiones posibles
    for k = 1:totPoints-2*ceil(totPoints*splitm)+2
        %Separación de 1:M*m-1+k
        try
            limitesA = limitesNodo_v0(dataBase, sortedPoints(1:ceil(totPoints*splitm)-1+k));
        catch
            %Si se tiene 1 solo elemento, los límites son los del punto
            limitesA = repelem(dataBase(sortedPoints(1)), 2);
```

```
33          end
34          %Separación de M*m+k:-1
35          try
36              limitesB = limitesNodo_v0(dataBase,
                    sortedPoints(ceil(totPoints*splitm)+k:end));
37          catch
38              %Si se tiene 1 solo elemento, los límites son los
                    del punto
39              limitesB =
                    repelem(dataBase(sortedPoints(length(sortedPoints))),
                    2);
40          end
41          %Se calcula la logitud de las aristas de los
                rectángulos
42          for i = 1:size(dataBase, 2)
43              aristasA(i) = abs(limitesA(2*i-1)-limitesA(2*i));
44              aristasB(i) = abs(limitesB(2*i-1)-limitesB(2*i));
45          end
46          %Se suma a S el resultado de la iteración actual
47          if size(dataBase, 2) = = 2
48              %Perímetro de un rectángulo
49              S = S+2*(sum(aristasA)+sum(aristasB));
50          elseif size(dataBase, 2) = = 3
51              %Área de un prisma
52              S =
                    S+2*(aristasA*circshift(aristasA)'+aristasB*circshift(arist
53          else
54              %Hipervolumen
55              S = S+prod(aristasA)+prod(aristasB);
56          end
57      end
58      %Se actualiza el valor mínimo
59      if S<minS
60          splitEje = eje;
61          minS = S;
62      end
63  end
```

## 4.3 BuscarVecinos

```
1  function vecinos = buscarVecinosRTree_v0(baseDatos, RTree, b,
       punto)
2  %BUSCARVECINOSRTREE Busca todos los puntos a una distancia <b
       del punto
3  %   baseDatos es una matriz de tamaño nxd, donde n es el
       número de puntos y d su dimensión.
```

```
4 %   RTree es la estructura sobre la que se opera
5 %   b es la distancia máxima entre 2 puntos para que estén
    conectados.
6 %   punto es el índice del punto sobre el que se buscan los
    vecinos
7 %   devuelve un array de indices de los vecinos del punto

9 %Se busca primero la hoja del árbol al que pertenece
10 hojaIndex = buscarHoja_v0(RTree, baseDatos(punto, :));

12 %Primero, se buscan vecinos dentro del propio nodo
13 %Si el nodo está totalmente conectado, todos los puntos del
    nodo son sus vecinos
14 if RTree{hojaIndex, 3}(1) = = -2
15     vecinos = RTree{hojaIndex, 3}(2:end);
16 %Si no, se lanza la búsqueda usual de vecinos dentro del nodo
17 else
18     vecinos = buscarVecinosClasico(baseDatos, punto,
         RTree{hojaIndex, 3}(2:end), b);
19 end

21 %Se guardan las coordenadas del punto
22 coordPunto = baseDatos(punto, :);
23 %Si el punto está suficientemente cerca del borde de su nodo,
    se buscan posibles vecinos en nodos cercanos
24 if checkDistFront(RTree{hojaIndex, 2}, coordPunto, b)
25     %Se identifica el padre que contiene todo los posibles
         vecinos del punto
26     padreActual = RTree{hojaIndex, 1};
27     while checkDistFront(RTree{padreActual, 2}, coordPunto, b)
         && padreActual~ = 1
28         padreActual = RTree{padreActual, 1};
29     end

31     %Se hace una búsqueda en profundidad de las hojas cercanas
         al punto
32     candidatos = RTree{padreActual, 3};
33     %Se guarda una lista con todos los puntos para comprobar
34     while ~isempty(candidatos)
35         newCandidatos = [];
36         for candidato = candidatos
37             if candidato = = hojaIndex
38                 continue
39             end
40             %Si el nodo candidato es una hoja, se comprueban
                 los puntos
41             if RTree{candidato, 3}(1)<0
42                 vecinos = [vecinos
```

```
                    buscarVecinosClasico(baseDatos, punto,
                    RTree{candidato, 3}(2:end), b)];
43              %Si el nodo candidato está cerca, se añaden sus
                    hijos a la lista de candidatos
44              elseif checkDistFront(RTree{candidato, 2},
                    coordPunto, b)
45                  newCandidatos = [newCandidatos
                    RTree{candidato, 3}];
46              end
47          end
48          candidatos = newCandidatos;
49      end
50  end
51  end

53  function vecinos = buscarVecinosClasico(baseDatos, punto,
        candidatos, b)
54  %Busca vecinos de un punto calculando la distancia euclídea
        punto a punto
55  vecinos = [];
56  for candidato = candidatos
57      if sqrt((baseDatos(punto, :)-baseDatos(candidato,
            :))*(baseDatos(punto, :)-baseDatos(candidato, :))')<b
           && punto~ = candidato
58          vecinos = [vecinos candidato];
59      end
60  end
61  end

63  function res = checkDistFront(coordNodo, coordPunto, b)
64  %Determina si un punto está cerca de la frontera de un nodo
        usando la distancia Manhattan
65  res = 0;
66  for d = 1:length(coordPunto)
67      %Distancia al borde
68      if abs(coordNodo(d*2)-coordPunto(d))<b ||
           abs(coordNodo(d*2-1)-coordPunto(d))<b
69          res = 1;
70          return
71      end
72  end
```

### 4.3.1 BuscarHoja

```
function hoja = buscarHoja_v0(RTree, coordPunto)
%BUSCARHOJA Busca la hoja a la que pertenece un punto
%   RTree es la estructura sobre la que se opera
%   coordPunto es un array con las coordenadas del punto
%   devuelve el índice de la hoja a la que pertenece

%Indice del nodo en el que está buscando, se empieza siempre en la raíz del árbol
nodoActual = 1;
while true
    %Si el nodo actual es una hoja, se devuelve el nodo en el que se está
    if RTree{nodoActual, 3}(1)<0
        hoja = nodoActual;
        return
    end
    %Si no, se decide en cual de los dos hijos está
    hijo = RTree{nodoActual, 3}(1);
    if puntoInNodo(RTree{hijo, 2}, coordPunto)
        nodoActual = hijo;
        continue
    end
    nodoActual = RTree{nodoActual, 3}(2);
end
```

### 4.3.2 PuntoInNodo

```
function res = puntoInNodo(coordNodo, coordPunto)
%Devuelve un booleano si el punto está en el espacio definido por el hijo
res = 1;
for d = 1:length(coordPunto)
    if coordPunto(d)<coordNodo(2*d-1) || coordNodo(2*d)<coordPunto(d)
        res = 0;
        return
    end
end
```

# 5. FuncionesComunes

## 5.1 Dist

```
function res = dist(dataBase, puntoA, puntoB)
%Distancia euclídea
aux = dataBase(puntoA, :)-dataBase(puntoB, :);
res = sqrt(aux*aux');
```

## 5.2 MinDistManhattan

```
function res = minDistManhattan_v0(Tree, nodoA, nodoB)
%MINDISTMANHATTAN Calcula la distancia Manhattan mínima entre
    dos celdas de un árbol. Se calcula con el mínimo de
    distancia de entre todas las direcciones posibles. Si una
    celda está dentro de otra, la distancia calculada es 0
%   Tree el árbol en el que se hace el cálculo
%   nodoA y nodoB son los índices de las celdas sobre las que
    se hace el cálculo
%   devuelve el límite superior de la distancia manhattan
    mínima

res = 0;
if nodoA =  = 5 && nodoB =  = 4
for d = 1:length(Tree{nodoA, 2})/2
    %Si una partición está "dentro" de otra, se salta el
       cálculo en esa dimensión
    % minB<=minA<maxA<=maxB o minA<=minB<maxB<=maxA
     if (Tree{nodoB, 2}(2*d-1)<=Tree{nodoA, 2}(2*d-1) &&
        Tree{nodoA, 2}(2*d)<=Tree{nodoB, 2}(2*d)) ||
        (Tree{nodoA, 2}(2*d-1)<=Tree{nodoB, 2}(2*d-1) &&
        Tree{nodoB, 2}(2*d)<=Tree{nodoA, 2}(2*d))
         continue
     end
    %Se calcula la distancia mínima entre max-max, max-min,
       min-max, min-min, y se compara con la distancia actual,
       guardándose la máxima
     res = max([res min([abs(Tree{nodoA, 2}(2*d)-Tree{nodoB,
        2}(2*d)) abs(Tree{nodoA, 2}(2*d)-Tree{nodoB, 2}(2*d-1))
        abs(Tree{nodoA, 2}(2*d-1)-Tree{nodoB, 2}(2*d))
        abs(Tree{nodoA, 2}(2*d-1)-Tree{nodoB, 2}(2*d-1))])]);
end
```