UNIVERSIDAD NACIONAL DE AGUSTÍN

FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS

ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACIÓN

KD-TREE

Alumno:
CUEVA FLORES, JONATHAN BRANDON

Docente:
Machaca Arceda VICENTE

Arequipa - Perú

Índice

Introducción

Un árbol kd emplea sólo planos perpendiculares a uno de los ejes del sistema de coordenadas. Esto difiere de los árboles BSP, donde los planos pueden ser arbitrarios. Además, todos los nodos de un árbol kd, desde el nodo raíz hasta los nodos hoja, almacenan un punto. Mientras tanto, en los árboles BSP son las hojas los únicos nodos que contienen puntos (u otras primitivas geométricas). Como consecuencia, cada plano debe pasar a través de uno de los puntos del árbol kd.

1. Construcción

Dado que hay muchas maneras posibles de elegir planos alineados a los ejes, hay muchas maneras de generar árboles kd. El sistema habitual es:

- Conforme se desciende en el árbol, se emplean ciclos a través de los ejes para seleccionar los planos. (Por ejemplo, la raíz puede tener un plano alineado con el eje x, sus descendientes tendrían planos alineados con el y y los nietos de la raíz alineados con el z, y así sucesivamente)
- En cada paso, el punto seleccionado para crear el plano de corte será la mediana de los puntos puestos en el árbol kd, lo que respeta sus coordenadas en el eje que está siendo usado.

Este método lleva a un árbol kd balanceado, donde cada nodo hoja está a la misma distancia de la raíz. De todas formas, los árboles balanceados no son necesariamente óptimos para todas las aplicaciones.

Dada una lista de n puntos, el siguiente algoritmo genera un árbol kd balanceado que contiene dichos puntos.

2. DATOS DE ALTA DIMENSIÓN

Debido a la maldición de la dimensionalidad que lleva a que la mayoría de las búsquedas en espacios de alta dimensión terminen siendo búsquedas brutas innecesariamente sofisticadas, los k-d tree no son adecuados para encontrar eficientemente al vecino más cercano en espacios de alta dimensión. Como regla general, si la dimensionalidad es k, el número de puntos en los datos, N, debe ser $N >> 2^k$. De lo contrario, cuando se usan kd tree con datos de alta dimensión, se evaluarán la mayoría de los puntos del árbol y la eficiencia no será mejor que la búsqueda exhaustiva y, si es lo suficientemente rápido se requiere respuesta, en su lugar se deben usar métodos aproximados del vecino más cercano. [1]

3. Origen y Informacion

- Jon Bentley, 1975
- Árbol usado para almacenar datos espaciales.
- Búsqueda de vecinos más cercanos.
- Consultas de rango.
- Búsqueda rápida. puntos en el espacio d-dimensional que son equivalentes a los vectores en el espacio d-dimensional.[2]

4. CODIGO PYTHON

```
from random import*
import vtk
from collections import defaultdict
from collections import deque
import networkx as nx
from networkx.drawing.nx_agraph import graphviz_layout
import matplotlib.pyplot as plt
import time


#####################       variables globales
    ###################################


levels = defaultdict(list)
Grafo = nx.DiGraph()



#####################       construccion grafo tree
    ###################################

def hierarchy_pos(G, root=None, width=1., vert_gap = 0.2,
    vert_loc = 0, xcenter = 0.5):
    if not nx.is_tree(G):
        raise TypeError('cannot use hierarchy_pos on a graph
    that is not a tree')

    if root is None:
        if isinstance(G, nx.DiGraph):
            root = next(iter(nx.topological_sort(G)))  #allows
    back compatibility with nx version 1.11
        else:
            root = random.choice(list(G.nodes))

    def _hierarchy_pos(G, root, width=1., vert_gap = 0.2,
    vert_loc = 0, xcenter = 0.5, pos = None, parent = None):

        if pos is None:
            pos = {root:(xcenter,vert_loc)}
        else:
            pos[root] = (xcenter, vert_loc)
        children = list(G.neighbors(root))
        if not isinstance(G, nx.DiGraph) and parent is not None:
            children.remove(parent)
        if len(children)!=0:
```

```
41              dx = width/len(children)
42              nextx = xcenter - width/2 - dx/2
43              for child in children:
44                  nextx += dx
45                  pos = _hierarchy_pos(G,child, width = dx,
    vert_gap = vert_gap,
46                                       vert_loc = vert_loc-vert_gap
    , xcenter=nextx,
47                                       pos=pos, parent = root)
48          return pos
49
50
51      return _hierarchy_pos(G, root, width, vert_gap, vert_loc,
    xcenter)
52
53
54 ################     VTK     ######################
55
56 dots=[]
57 actors=[]
58 lineasf = []
59 lista = []
60
61 def cubo(x,y,z,w,h,p,axis):
62     cube=vtk.vtkCubeSource()
63     cube.SetXLength(w)
64     cube.SetYLength(h)
65     cube.SetZLength(p)
66     cube.SetCenter(x,y,z)
67     cubeMapper=vtk.vtkPolyDataMapper()
68     cubeMapper.SetInputConnection(cube.GetOutputPort())
69     cubeActor=vtk.vtkActor()
70     if(axis==2):
71         cubeActor.GetProperty().SetColor(0,0,255)
72     if(axis==1):
73         cubeActor.GetProperty().SetColor(0,255,0)
74     if(axis==0):
75         cubeActor.GetProperty().SetColor(255,0,0)
76     cubeActor.GetProperty().SetOpacity(0.8)
77     cubeActor.SetMapper(cubeMapper)
78     lineasf.append(cubeActor)
79
80 def rect(x,y,z,w,h,p):
81     cube=vtk.vtkCubeSource()
82     cube.SetXLength(w)
83     cube.SetYLength(h)
84     cube.SetZLength(p)
85     cube.SetCenter(x,y,z)
86     cubeMapper=vtk.vtkPolyDataMapper()
```

```
87      cubeMapper.SetInputConnection(cube.GetOutputPort())
88      cubeActor=vtk.vtkActor()
89      #cubeActor.GetProperty().SetColor(243,195,2)
90      cubeActor.GetProperty().SetOpacity(0.2)
91      cubeActor.SetMapper(cubeMapper)
92      actors.append(cubeActor)
93
94  def dot(point):
95      sphere=vtk.vtkSphereSource()
96      sphere.SetRadius(3.5)
97      if len(point) == 2:
98          sphere.SetCenter(point[0],point[1],0)
99      else:
100         sphere.SetCenter(point[0],point[1],point[2])
101     sphereMapper=vtk.vtkPolyDataMapper()
102     sphereMapper.SetInputConnection(sphere.GetOutputPort())
103     sphereActor=vtk.vtkActor()
104     sphereActor.GetProperty().SetColor(255,255,0)
105     sphereActor.SetMapper(sphereMapper)
106     dots.append(sphereActor)
107
108 def line(x,y,altura=400,region='X'):
109     cube=vtk.vtkLineSource()
110     cubeActor=vtk.vtkActor()
111     if(region=='Y'):
112         cube.SetPoint1(x,y,0)
113         cube.SetPoint2(altura,y,0)
114         cubeMapper=vtk.vtkPolyDataMapper()
115         cubeMapper.SetInputConnection(cube.GetOutputPort())
116         cubeActor.GetProperty().SetColor(0,0,255)
117         cubeActor.SetMapper(cubeMapper)
118     else:
119         cube.SetPoint1(x,y,0)
120         cube.SetPoint2(x,altura,0)
121         cubeMapper=vtk.vtkPolyDataMapper()
122         cubeMapper.SetInputConnection(cube.GetOutputPort())
123         cubeActor.GetProperty().SetColor(255,0,0)
124         cubeActor.SetMapper(cubeMapper)
125     lineasf.append(cubeActor)
126
127
128
129
130 def left_min(node, tmp, i):
131     if(node!=None):
132         if(node.point[i]<=tmp.point[i] and node.axis == i):
133             return node
134         return left_min(node.parent,tmp,i)
135     return None
```

```
136 def right_min(node, tmp, i):
137     if(node!=None):
138         if(node.point[i]>=tmp.point[i] and node.axis == i):
139             return node
140         return right_min(node.parent,tmp,i)
141         #return node
142     return None
143 
144 def construccion_lines():
145     for i in levels:
146         for j in levels[i]:
147             if (j.axis == 0):
148                 if(j.parent!=None):
149                     if(j.parent.point[1]>=j.point[1]):
150                         line(j.point[0],0,j.parent.point[1],'X')
151                     else:
152                         line(j.point[0],400,j.parent.point[1],'X
    ')
153                 else:
154                     line(j.point[0],0)
155             else:
156                 if(j.parent!=None):
157                     if(j.parent.point[0]>=j.point[0]):
158                         node_aux = left_min(j.parent,j,0)
159                         if(node_aux==None):
160                             line(0,j.point[1],j.parent.point[0],
    'Y')
161                         else:
162                             print(j.point,"\t",node_aux.point)
163                             line(node_aux.point[0],j.point[1],j.
    parent.point[0],'Y')
164                     else:
165                         node_aux = right_min(j.parent,j,0)
166                         if(node_aux==None):
167                             line(400,j.point[1],j.parent.point
    [0],'Y')
168                         else:
169                             print(j.point,"\t",node_aux.point)
170                             line(node_aux.point[0],j.point[1],j.
    parent.point[0],'Y')
171                 else:
172                     line(0,j.point[1],j.parent.point[0],'Y')
173 
174 def construccion_separator():
175     for i in levels:
176         for j in levels[i]:
177             if(i%2!=0):
178                 if(j.parent.point[0]>=j.point[0]):
179                     node = left_min(j.parent,j,0)
```

```
180                         if (node != None and node.point[0]<=j.point
    [0]):
181                             line(node.point[0],j.point[1],j.parent.
    point[0],'Y')
182                         else:
183                             line(0,j.point[1],j.parent.point[0],'Y')
184                     else:
185                         node = right_min(j.parent,j,0)
186                         if (node != None and node.point[0]<=j.point
    [0]):
187                             print(j.parent.point[0],"\t",j.point,"\t
    ",node.point[0])
188                             line(j.parent.point[0],j.point[1],node.
    point[0],'Y')
189                         else:
190                             line(400,j.point[1],j.parent.point[0],'Y
    ')
191                 else:
192                     if(j.parent!=None):
193                         if(j.parent.point[1]>=j.point[1]):
194                             line(j.point[0],0,j.parent.point[1],'X')
195                         else:
196                             line(j.point[0],400,j.parent.point[1],'X
    ')
197                     else:
198                         line(j.point[0],0)
199
200 def construccion_planes(node,x,w,y,h,z,p,nivel=0):
201     if(node==None):
202         return None
203     else:
204         if(nivel==0):
205             lista.append([(node.point[0],(y+h)/2,(z+p)/2),(0,abs
    (h-y),abs(p-z)),nivel])
206             construccion_planes(node.left,x,node.point[0],y,h,z,
    p,1)
207             construccion_planes(node.right,node.point[0],w,y,h,z
    ,p,1)
208         if(nivel==1):
209             lista.append([((x+w)/2,node.point[1],(z+p)/2),(abs(w
    -x),0,abs(p-z)),nivel])
210             construccion_planes(node.left,x,w,y,node.point[1],z,
    p,2)
211             construccion_planes(node.right,x,w,node.point[1],h,z
    ,p,2)
212         if(nivel==2):
213             lista.append([((x+w)/2,(y+h)/2,node.point[2]),(abs(w
    -x),abs(h-y),0),nivel])
```

```
214             construccion_planes(node.left ,x,w,y,h,z,node.point
    [2],0)
215             construccion_planes(node.right ,x,w,y,h,node.point
    [2],p,0)
216
217
218 def Screen():
219     ren = vtk.vtkRenderer()
220     ren.AddActor(actors[0])
221     renWin = vtk.vtkRenderWindow()
222     renWin.AddRenderer(ren)
223     renWin.SetSize(600, 600)
224     iren = vtk.vtkRenderWindowInteractor()
225     iren.SetRenderWindow(renWin)
226     ren.SetBackground(0,0,0)
227
228     for act in dots:
229         ren.AddActor(act)
230         renWin.AddRenderer(ren)
231         iren.SetRenderWindow(renWin)
232         renWin.Render()
233         #time.sleep(0.3)
234     for act in lineasf:
235         ren.AddActor(act)
236         renWin.AddRenderer(ren)
237         iren.SetRenderWindow(renWin)
238         renWin.Render()
239         time.sleep(1)
240
241     iren.Start()
242
243
244
245
246 ####################          k-d tree
    ####################################
247
248 class Node:
249     def __init__(self,point,axis):
250         self.point = point
251         self.left = None
252         self.right = None
253         self.axis = axis
254         self.parent = None
255     def set_parent(self,node):
256         self.parent = node
257
258 def build_kdtree(points, depth = 0, nodes=None):
259     if not points:
```

```
260         return
261
262     k = len(points[0])
263     axis = depth %k
264
265     points.sort(key=lambda x:x[axis])
266     median = len(points)//2
267
268     node = Node(points[median],axis)
269     node.set_parent(nodes)
270     node.left = build_kdtree(points[0:median], depth+1,node)
271     node.right = build_kdtree(points[median+1:], depth+1,node)
272     return node
273
274
275 def generate_graph(node):
276     if node.left!=None:
277         str1 = ','.join(str(e) for e in node.point)
278         str2 = ','.join(str(e) for e in node.left.point)
279         Grafo.add_edge(str1,str2)
280         generate_graph(node.left)
281     if node.right!=None:
282         str1 = ','.join(str(e) for e in node.point)
283         str2 = ','.join(str(e) for e in node.right.point)
284         Grafo.add_edge(str1,str2)
285         generate_graph(node.right)
286
287
288 def arbol_levels(node):
289     if node == None:
290         return None
291     cola = []
292     cola.append(node)
293     nvl = 0
294     while cola:
295         nodeCount = len(cola)
296         while nodeCount > 0:
297             tmp = cola[0]
298             dot(tmp.point)
299             temp = cola.pop(0)
300             levels[nvl].append(tmp)
301             print(tmp.point,"\t",tmp.axis,end="\t\t")
302             if(tmp.left!=None):
303                 cola.append(tmp.left)
304             if(tmp.right!=None):
305                 cola.append(tmp.right)
306             nodeCount-=1
307         print("\n")
308         nvl+=1
```

```
309
310
311
312
313
314
315 ################          MAIN                ################
316
317 def main():
318     #pointList = [(30,60),(20,70),(170,150),(60,120),(130,150)
    ,(90,10),(100,190),(135,80)]
319     #pointList = [(279,121),(233,203),(367,272),(173,16)
    ,(152,326),(370,51),(360,377),(102,73),(250,102),(72,268)
    ,(361,57),(382,98),(331,288)]
320     pointList=[]
321     dimensions = 2
322     cantidad = 10
323     if(dimensions == 3):
324         rect (200,200,200,400,400,400)
325         for i in range(0,cantidad):
326             val1 = randrange(10,390)
327             val2 = randrange(10,390)
328             val3 = randrange(10,390)
329             pointList.append((val1,val2,val3))
330     else:
331         rect (200,200,0,400,400,0)
332         for i in range(0,cantidad):
333             val1 = randrange(10,390)
334             val2 = randrange(10,390)
335             pointList.append((val1,val2))
336
337     tree = build_kdtree(pointList)
338     generate_graph(tree)
339     arbol_levels(tree)
340     str1 = ','.join(str(e) for e in tree.point)
341     pos = hierarchy_pos(Grafo,str1)
342     nx.draw(Grafo,pos=pos,with_labels=True,font_size = 15,
343                     width = 2)
344
345     plt.show()
346     #construccion_separator()
347     if(dimensions == 2):
348         construccion_lines()
349     else:
350         construccion_planes(tree,0,400,0,400,0,400)
351         print(lista)
352         for i in lista:
353             cubo(i[0][0],i[0][1],i[0][2],i[1][0],i[1][1],i
    [1][2],i[2])
```

```
354         #print(tree.left.point,"\t",tree.right.point)
355 
356 main()
357 Screen()
```

te.py

5. Imagenes de Prueba

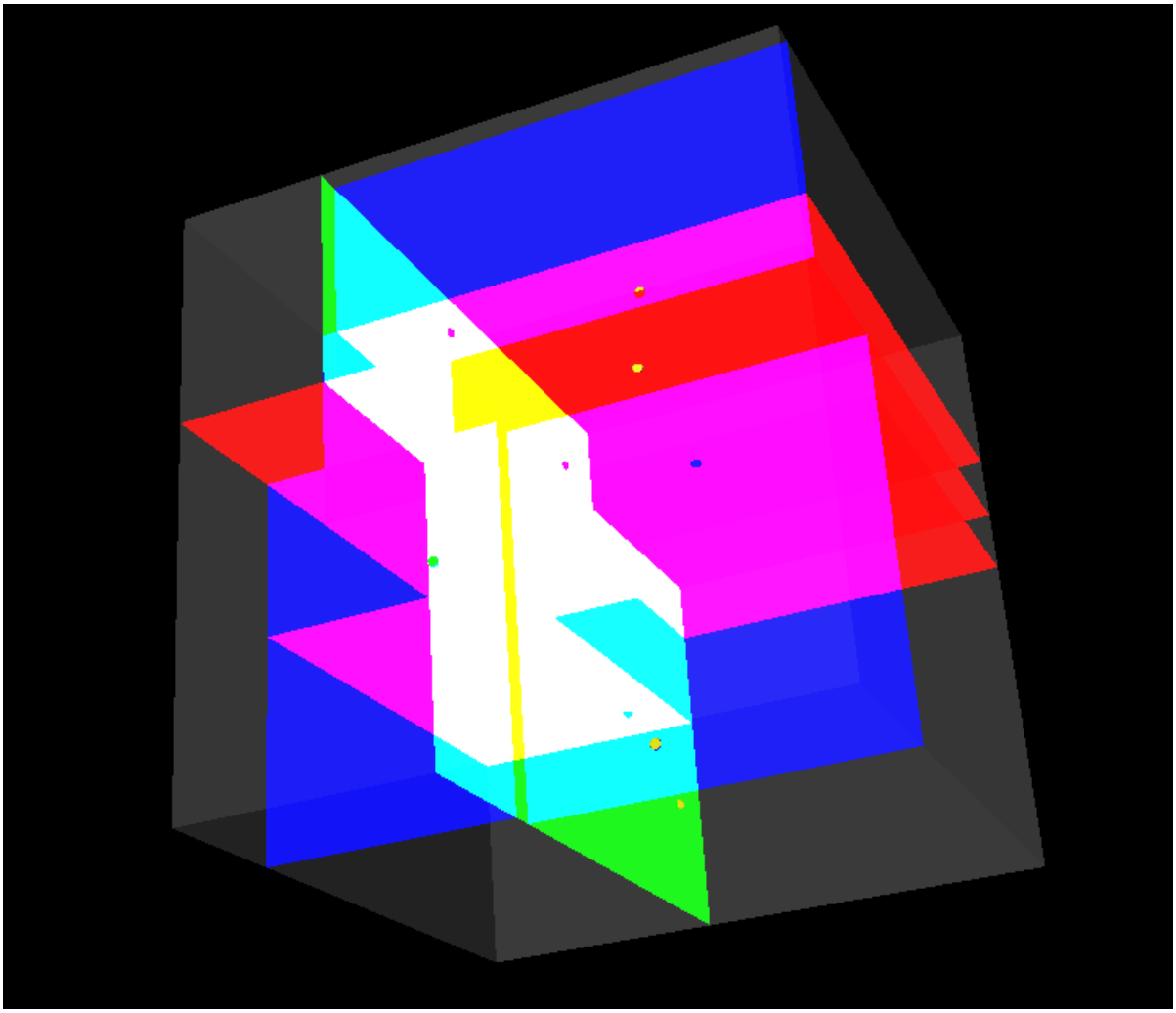

Figura 1: Test de 15 datos en 3 dimensiones

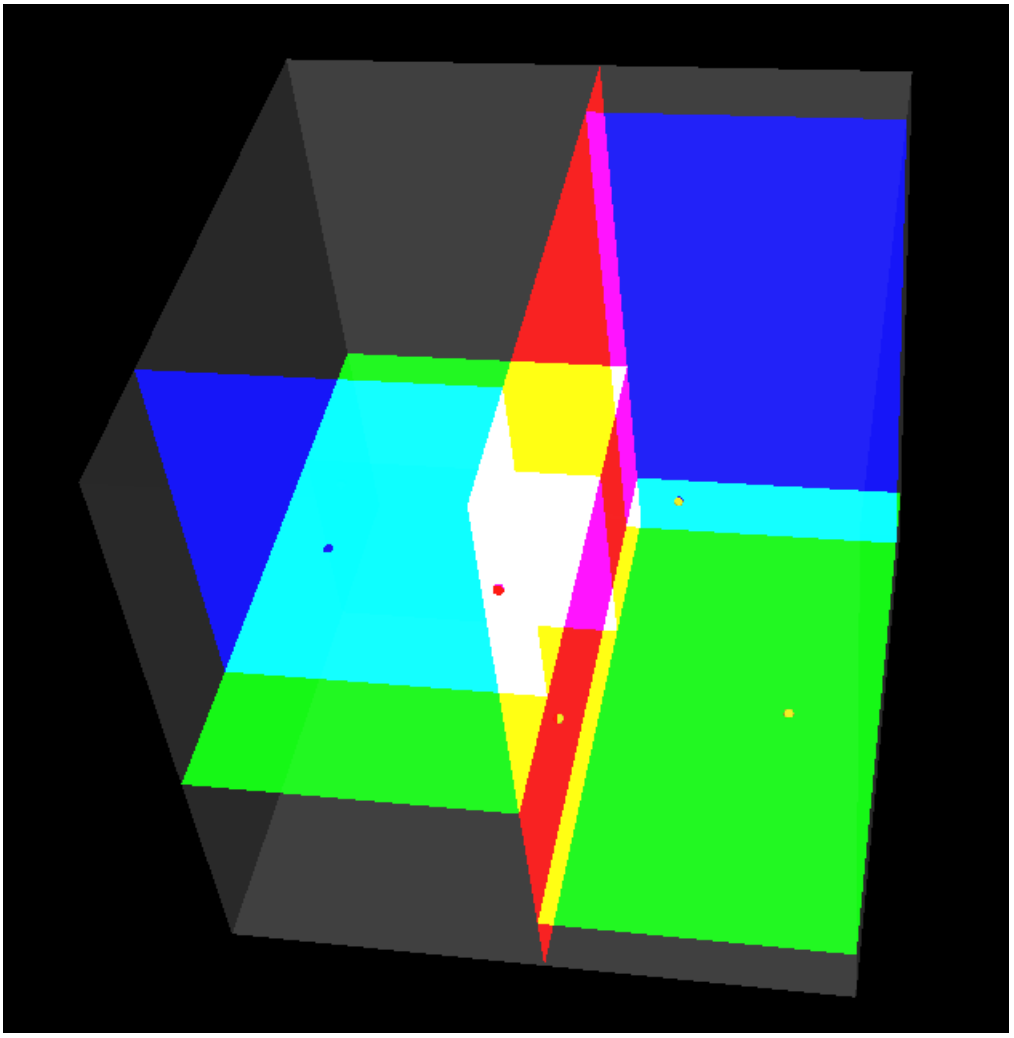

Figura 2: Test de 7 datos en 3 dimensiones

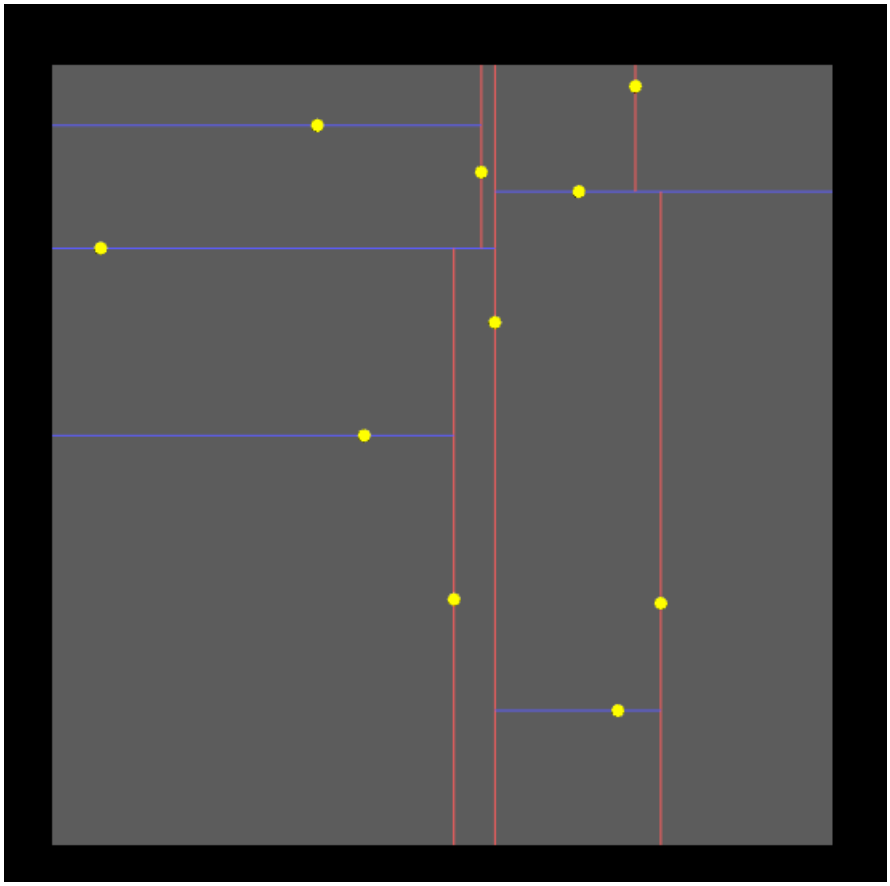

Figura 3: Test de 10 datos en 2 dimensiones

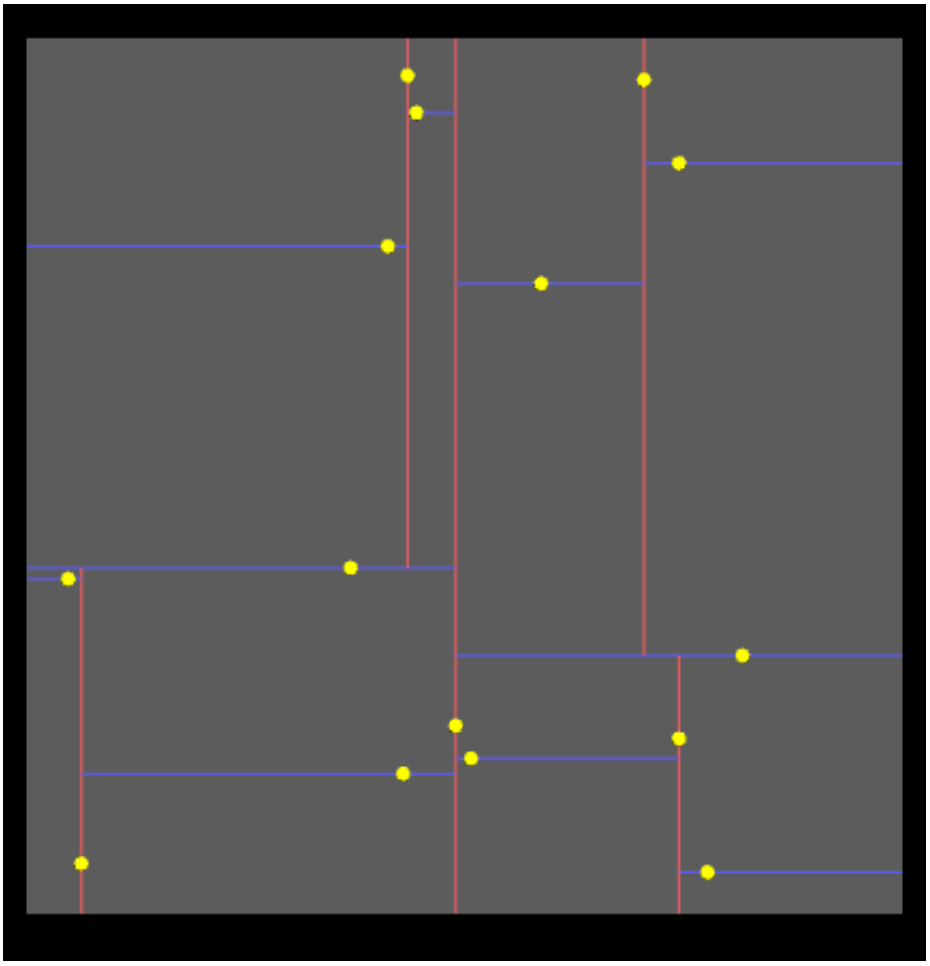

Figura 4: Test de 15 datos en 2 dimensiones

6. Grafos

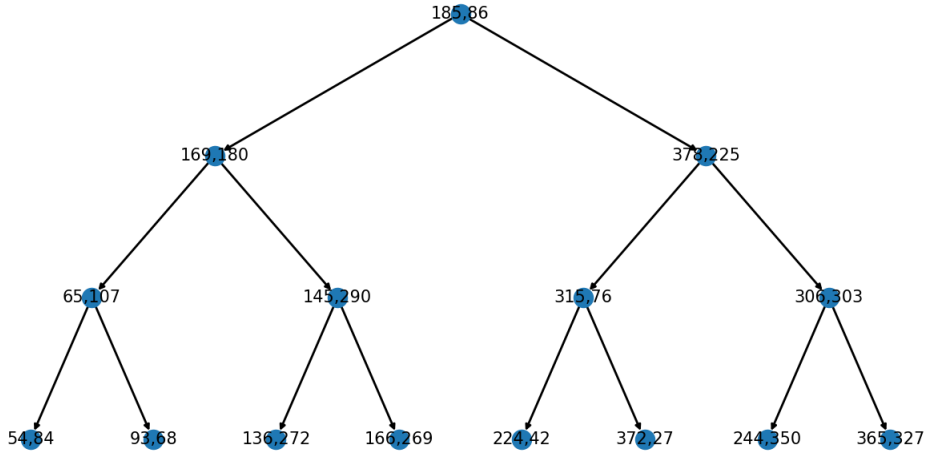

Figura 5: Grafo para dos dimensiones

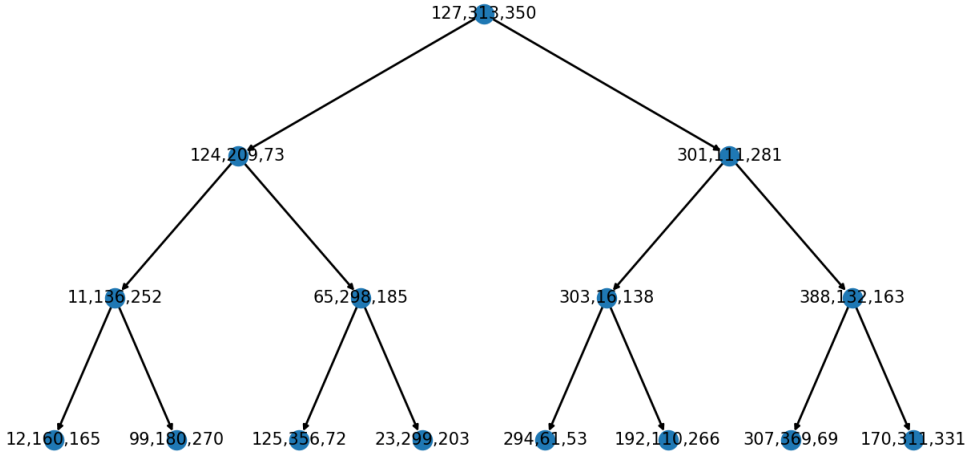

Figura 6: Grafo para tres dimensiones

7. Conclusiones

- Búsqueda del punto mas cercano de manera eficiente.
- Permite diferentes dimensiones de un punto.
- Para balancear el árbol toma el tiempo del sort implementado.
- Usa una base BST para generar un arbol de facil acceso.

Referencias

[1] J. L. Bentley. "multidimensional binary search trees used for associative searching". *Communications of the ACM.*

[2] Daniel Chacón Moreno. *Estudio y análisis de la teoría de la multirresolución en el modelado de sólidos.* Number 26-34. Tesis Maestría. Ciencias con Especialidad en Ingeniería en Sistemas Computacionales, 2000.