UNIVERSIDAD NACIONAL DE
SAN AGUSTÍN DE AREQUIPA

SCHOOL OF COMPUTER SCIENCE

Course : Advance Data Structures

kd - Tree

Students:

HERRERA COOPER, MIGUEL ALEXANDER

Índice

Índice **1**

Índice de figuras **1**

1. Introduction **2**

2. kd-Trees **3**
2.1. Build kd-Tree 6
2.2. Search kd-Tree 10

3. Code **13**
3.1. Proofs in 2-D 16
3.2. Proof using astroML in 2D 17
3.3. Proofs in 3D 18

4. Refferences **20**

Índice de figuras

1. Reprsentation of a point i a plane 2D. 3
2. Subdivide in P_{left} and P_{right}. 4
3. A kd-tree: on the left the way the plane is subdivided and on the right the corresponding binary tree 5
4. Correspondence between nodes in a kd-tree and regions in the plane . 8
5. A query on a kd-tree 9
6. Proof 1 16
7. Proof 2 16
8. Proof 3 - 30 point in 4 levels 17
9. Proof 4 18
10. Proof 5 19

7 de noviembre de 2019

1. Introduction

Progressive data analysis has recently gained in popularity due to its ability to deliver ongoing results before the whole computation is com- pleted. In contrast to previous computation paradigms such as online computation, progressive algorithms deliver estimates at a bounded rate: they are guaranteed to return a partial result in a speci- fied delay to comply with human attention constraints.

However, despite the advantages of progressive computation, it is not always simple or even possible to convert a sequential algorithm directly to a progressive one, and such a hurdle hinders the applicabil- ity of progressive computation to a wider range of data analyses.

In this report we present the kd-Tree algorithm to divide the space into d dimensions.

In the visualization part we present the subdivision of the space in 2 and 3 dimensions in real time.

The algorithm is implemented in Python and we use its visualization and animation libraries to demonstrate the operation of the algorithm.

2. kd-Trees

Now let's go to the 2-dimensional rectangular range searching problem. Let P be a set of n points in the plane. In the remainder of this section we assume that no two points in P have the same x-coordinate, and no two points have the same y-coordinate. This restriction is not very realistic, especially not if the points represent employees and the coordinates are things like salary or number of children.

A 2-dimensional rectangular range query on P asks for the points from P lying inside a query rectangle [x : x'] [y : y']. A point p := (p_x, p_y) lies inside this rectangle if and only if

p_x [x : x'] and p_y [y : y']

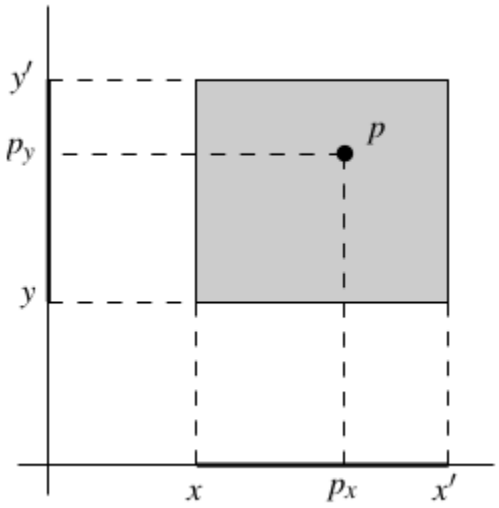


Figura 1: Reprsentation of a point i a plane 2D.

We could say that a 2-dimensional rectangular range query is composed of two 1-dimensional sub-queries, one on the x-coordinate of the points and one on the y-coordinate.

How can we generalize this structure—which was just a binary search tree—to 2-dimensional range queries? Let's consider the following recursive definition of the binary search tree: the set of (1-dimensional) points is split into two subsets of roughly equal size; one subset contains the points smaller than or equal to the splitting value, the other subset contains the points larger than the splitting value. The splitting value is stored at the root, and the two subsets are stored recursively in the two subtrees.

In the 2-dimensional case each point has two values that are important: its x- and its y-coordinate. Therefore we first split on x-coordinate, next on y-coordinate, then again on x-coordinate, and so on.

More precisely, the process is as follows. At the root we split the set P with a vertical line l into two subsets of roughly equal size. The splitting line is stored at the root. P_{left}, the subset of points to the left or on the splitting line, is stored in the left subtree, and P_{right}, the subset to the right of it, is stored in the right subtree. At the left child of the root we split P left into two subsets with a horizontal line; the points below or on it are stored in the left subtree of the left child, and the points above it are stored in the right subtree.

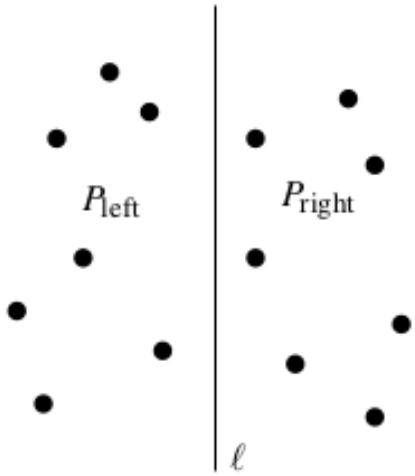


Figura 2: Subdivide in P_{left} and P_{right}.

The left child itself stores the splitting line. Similarly, the set P_{right} is split with a horizontal line into two subsets, which are stored in the left and right subtree of the right child. At the grandchildren of the root, we split again with a vertical line. In general, we split with a vertical line at nodes whose depth is even, and we split with a horizontal line at nodes whose depth is odd. A tree like this is called a kd-tree.

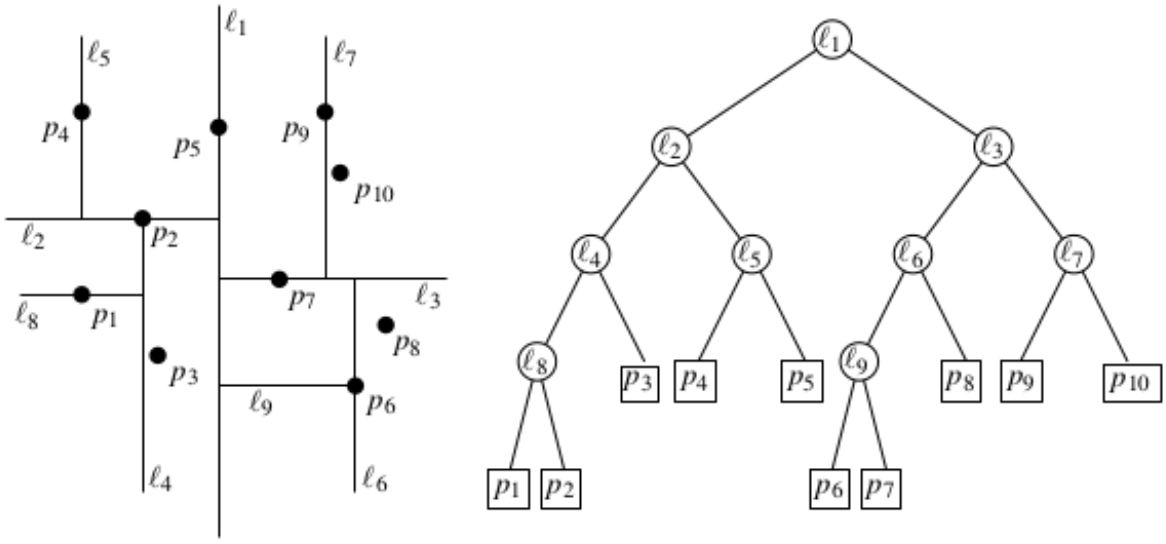


Figura 3: A kd-tree: on the left the way the plane is subdivided and on the right the corresponding binary tree

Originally, the name stood for k-dimensional tree; the tree we described above would be a 2d-tree. Nowadays, the original meaning is lost, and what used to be called a 2d-tree is now called a 2-dimensional kd-tree.

We can construct a kd-tree with the recursive procedure described below. This procedure has two parameters: a set of points and an integer. The first parameter is the set for which we want to build the kd-tree; initially this is the set P. The second parameter is depth of recursion or, in other words, the depth of the root of the subtree that the recursive call constructs. The depth parameter is zero at the first call.

The depth is important because, as explained above, it determines whether we must split with a vertical or a horizontal line. The procedure returns the root of the kd-tree.

2.1. Build kd-Tree

Algorithm BUILD KD-TREE (P, depth)
Input: A set of points P and the current depth depth.
Output: The root of a kd-tree storing P

Algorithm BUILDKDTREE(P,*depth*)
Input. A set of points P and the current depth *depth*.
Output. The root of a kd-tree storing P.
1. **if** P contains only one point
2. **then return** a leaf storing this point
3. **else if** *depth* is even
4. **then** Split P into two subsets with a vertical line ℓ through the median x-coordinate of the points in P. Let P_1 be the set of points to the left of ℓ or on ℓ, and let P_2 be the set of points to the right of ℓ.
5. **else** Split P into two subsets with a horizontal line ℓ through the median y-coordinate of the points in P. Let P_1 be the set of points below ℓ or on ℓ, and let P_2 be the set of points above ℓ.
6. $v_{\text{left}} \leftarrow$ BUILDKDTREE(P_1, *depth* + 1)
7. $v_{\text{right}} \leftarrow$ BUILDKDTREE(P_2, *depth* + 1)
8. Create a node v storing ℓ, make v_{left} the left child of v, and make v_{right} the right child of v.
9. **return** v

=0

The algorithm uses the convention that the point on the splitting line—the one determining the median x- or y-coordinate—belongs to the subset to the left of, or below, the splitting line. For this to work correctly, the median of a set of n numbers should be defined as the $\lceil n/2 \rceil - thsmallestnumber$.

This means that the median of two values is the smaller one, which ensures that the algorithm terminates.

Before we come to the query algorithm, let's analyze the construction time of a 2-dimensional kd-tree. The most expensive step that is performed at every recursive call is finding the splitting line. This requires determining the median x-coordinate or the median y-coordinate, depending on whether the depth is even or odd. Median finding can be done in linear time. Linear time median finding algorithms, however, are rather complicated. A better approach is to presort the set of points both on x- and on y-coordinate.

The parameter set P is now passed to the procedure in the form of two sorted lists, one on x-coordinate and one on y-coordinate. Given the two sorted lists, it is easy to find the median x-coordinate (when the depth is even) or the median y-coordinate (when the depth is odd) in linear time. It is also easy to construct the sorted lists for the two recursive calls in linear time from the given lists.

Hence, the building time T(n) satisfies the recurrence

$$T(n) = \begin{cases} O(1), & \text{if } n = 1, \\ O(n) + 2T(\lceil n/2 \rceil), & \text{if } n > 1, \end{cases}$$

which solves to O(n log n). This bound subsumes the time we spend for presorting the points on x- and y-coordinate.

To bound the amount of storage we note that each leaf in the kd-tree stores a distinct point of P. Hence, there are n leaves. Because a kd-tree is a binary tree, and every leaf and internal node uses O(1) storage, this implies that the total amount of storage is O(n). This leads to the following lemma. Lemma
A kd-tree for a set of n points uses O(n) storage and can be con- structed in O(n log n) time.

We now turn to the query algorithm. The splitting line stored at the root partitions the plane into two half-planes. The points in the left half-plane are stored in the left subtree, and the points in the right half-plane are stored in the right subtree. In a sense, the left child of the root corresponds to the left half plane and the right child corresponds to the right half-plane. (The convention used in BUILD KD TREE that the point on the splitting line belongs to the left subset implies that the left half-plane is closed to the right and the right half-plane is open to the left.)

The other nodes in a kd-tree correspond to a region of the plane as well. The left child of the left child of the root, for instance, corresponds to the region bounded to the right by the splitting line stored at the root and bounded from above by the line stored at the left child of the root.

In general, the region corresponding to a node is a rectangle, which can be unbounded on one or more sides. It is bounded by splitting lines stored at ancestors of —see Figure 4.

We denote the region corresponding to a node

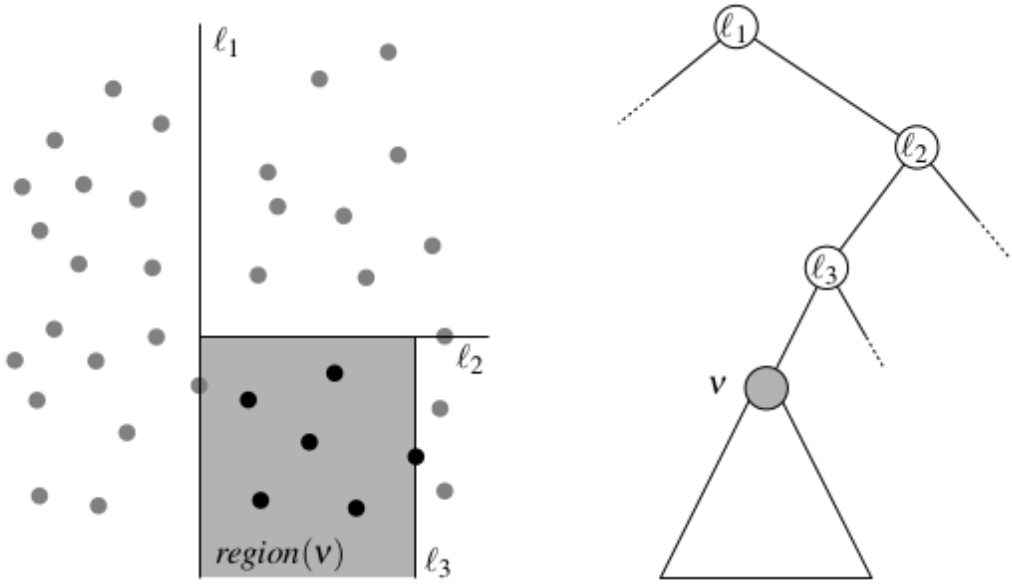


Figura 4: Correspondence between nodes in a kd-tree and regions in the plane

by region(). The region of the root of a kd-tree is simply the whole plane.

Observe that a point is stored in the subtree rooted at a node if and only if it lies in region(). For instance, the subtree of the node in Figure 4 stores the points indicated as black dots. Therefore we have to search the subtree rooted at only if the query rectangle intersects region().

This observation leads to the following query algorithm: we traverse the kd-tree, but visit only nodes whose region is intersected by the query rectangle. When a region is fully contained in the query rectangle, we can report all the points stored in its subtree. When the traversal reaches a leaf, we have to check whether the point stored at the leaf is contained in the query region and, if so, report it.

Figure 5 illustrates the query algorithm. (Note that the kd-tree of Figure 5 could not have been constructed by Algorithm BUILD KD TREE ; the median wasn't always chosen as the split value.) The grey nodes are visited when we query with the grey rectangle. The node marked with a star corresponds to a region that is completely contained in the query rectangle; in the figure this rectangular region is shown darker.

Hence, the dark grey subtree rooted at this node is traversed and all points stored in it are reported. The other leaves that are visited correspond to regions that are only partially inside the query rectangle.

Hence, the points stored in them must be tested for inclusion in the query range; this results in points p_6 and p_{11} being reported, and points p_3, p_{12}, and p_{13} not being reported.The query algorithm is described by the following recursive procedure,

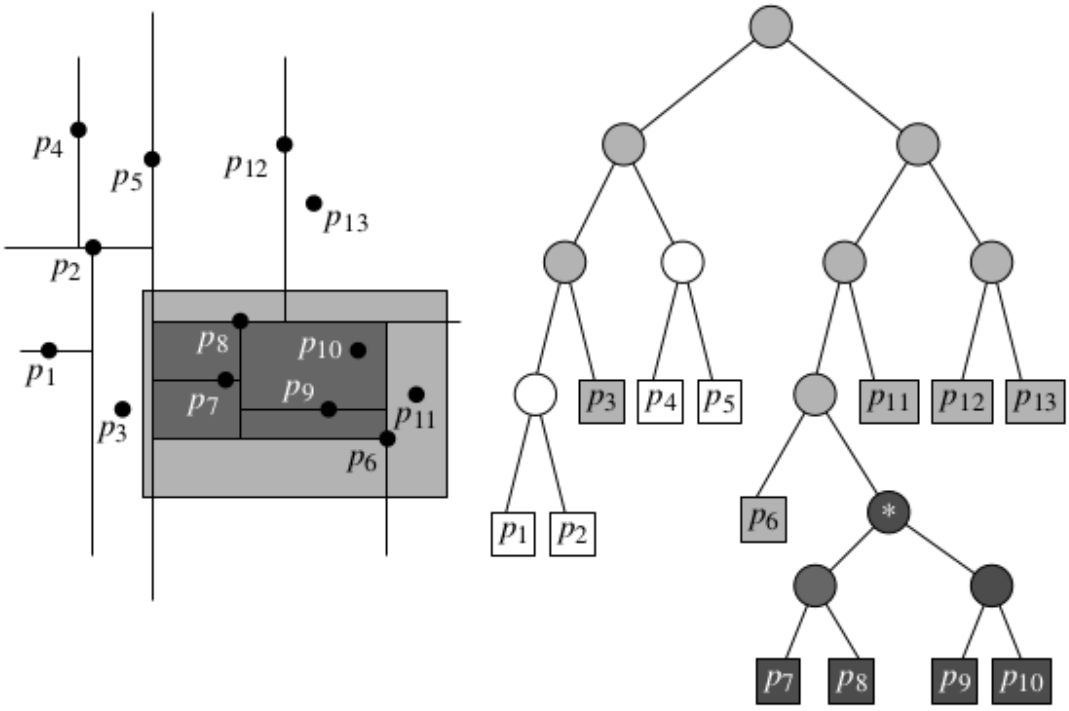


Figura 5: A query on a kd-tree

which takes as arguments the root of a kd-tree and the query range R.

It uses a subroutine REPORT SUBTREE (v), which traverses the subtree rooted at a node and reports all the points stored at its leaves.

Recall that lc(v) and rc(v) denote the left and right child of a node v, respectively.

2.2. Search kd-Tree

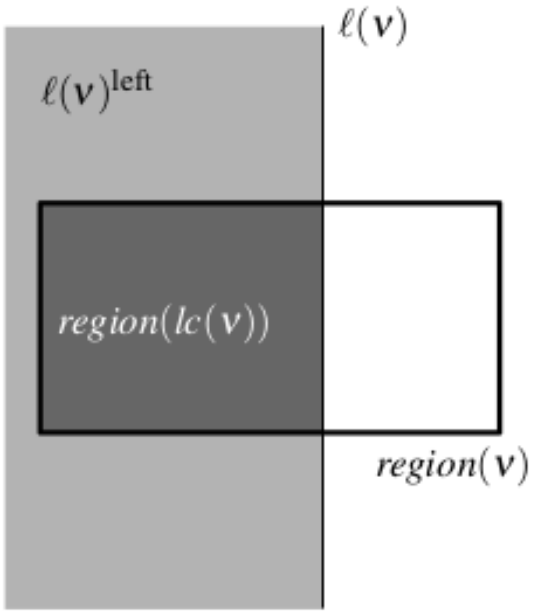
Algorithm SEARCHKDTREE(v, R)
Input. The root of (a subtree of) a kd-tree, and a range R.
Output. All points at leaves below v that lie in the range.

```
1.  if v is a leaf
2.    then Report the point stored at v if it lies in R.
3.    else if region(lc(v)) is fully contained in R
4.           then REPORTSUBTREE(lc(v))
5.           else if region(lc(v)) intersects R
6.                  then SEARCHKDTREE(lc(v),R)
7.         if region(rc(v)) is fully contained in R
8.           then REPORTSUBTREE(rc(v))
9.           else if region(rc(v)) intersects R
10.                 then SEARCHKDTREE(rc(v),R)
```

The main test the query algorithm performs is whether the query range R intersects the region corresponding to some node v. To be able to do this test we can compute region(v) for all nodes during the preprocessing phase and store it, but this is not necessary: one can maintain the current region through the recursive calls using the lines stored in the internal nodes.

For instance, the region corresponding to the left child of a node v at even depth can be computed from region(v) as follows:

$$region(lc(v)) = region(v) \cap \ell(v)^{\text{left}},$$



where l(v) is the splitting line stored at l, and $l(v)^{left}$ is the half-plane to the left of and including l(v).

Observe that the query algorithm above never assumes that the query range R is a rectangle. Indeed, it works for any other query range as well.

The analysis of the query time that we gave above is rather pessimistic: we bounded the number of regions intersecting an edge of the query rectangle by the number of regions intersecting the line through it. In many practical situations the range will be small. As a result, the edges are short and will intersect much fewer regions. For example, when we search with a range [x : x][y : y]—this query effectively asks whether the point (x, y) is in the set—the query time is bounded by O(log n). The following theorem summarizes the performance of kd-trees.

Theorem
A kd-tree for a set P of n points in the plane uses O(n) storage and can be built in O(n log n) time. A rectangular range query on the kd-tree takes O($\sqrt{n}$ + k) time, where k is the number of reported points.

Kd-trees can also be used for point sets in 3- or higher-dimensional space. The construction algorithm is very similar to the planar case: At the root, we split the set of points into two subsets of roughly the same size by a hyperplane perpendicular to the x 1 -axis. In other words, at the root the point set is partitioned based on the first coordinate of the points. At the children of the root the partition is based on the second coordinate, at nodes at depth two on the third coordinate, and so on, until at depth d 1 we partition on the last coordinate.

At depth d we start all over again, partitioning on first coordinate. The recursion stops when there is only one point left, which is then stored at a leaf. Because a d-dimensional kd-tree for a set of n points is a binary tree with n leaves, it uses O(n) storage. The construction time is O(n log n). (As usual, we assume d to be a constant.)

Nodes in a d-dimensional kd-tree correspond to regions, as in the plane. The query algorithm visits those nodes whose regions are properly intersected by the query range, and traverses subtrees (to report the points stored in the leaves) that are rooted at nodes whose region is fully contained in the query range.

It can be shown that the query time is bounded by $O(n^{1-1/d} + k)$.

3. Code

```
import numpy as np
from matplotlib import pyplot as plt

if "setup_text_plots" not in globals():
    from astroML.plotting import setup_text_plots
setup_text_plots(fontsize=8, usetex=False)


#Clase KDTree la cual dividira el espacio recursivamente
class KDTree:
    def __init__(self, data, mins, maxs):
        self.data = np.asarray(data)


        assert self.data.shape[1] == 2

        if mins is None:
            mins = data.min(0)
        if maxs is None:
            maxs = data.max(0)

        self.mins = np.asarray(mins)
        self.maxs = np.asarray(maxs)
        self.sizes = self.maxs - self.mins

        self.child1 = None
        self.child2 = None

        if len(data) > 1:
            # ordenar en la dimension con mayor difusion
            largest_dim = np.argmax(self.sizes)
            i_sort = np.argsort(self.data[:, largest_dim])
            self.data[:] = self.data[i_sort, :]

            # Encontrar punto de division
            N = self.data.shape[0]
            half_N = int(N / 2)
            split_point = 0.5 * (self.data[half_N, largest_dim]
                                 + self.data[half_N - 1, largest_dim])

            # Creo subnodos
            mins1 = self.mins.copy()
```

```
            mins1[largest_dim] = split_point
            maxs2 = self.maxs.copy()
            maxs2[largest_dim] = split_point

            # Recursivamente construyo un KD-Tree en cada subnodo
            self.child1 = KDTree(self.data[half_N:], mins1, self.maxs)
            self.child2 = KDTree(self.data[:half_N], self.mins, maxs2)

    def draw_rectangle(self, ax, depth=None):
        #REcusrivamente ploteamos una visualizacion de la regi n del KD-
        if depth == 0:
            rect = plt.Rectangle(self.mins, *self.sizes, ec='k', fc='none
            ax.add_patch(rect)

        if self.child1 is not None:
            if depth is None:
                self.child1.draw_rectangle(ax)
                self.child2.draw_rectangle(ax)
            elif depth > 0:
                self.child1.draw_rectangle(ax, depth - 1)
                self.child2.draw_rectangle(ax, depth - 1)


#Creamos un conjunto de puntos aleatoriamente structurados  en 2 dimensio
np.random.seed(0)

X = np.random.random((70, 2)) * 2 - 1
X[:, 1] *= 0.1
X[:, 1] += X[:, 0] ** 2


#Usamos nuestra clase KD-Tree para dividir recursivametne el espacio
KDT = KDTree(X, [-1.1, -0.1], [1.1, 1.1])


#ploteamos 7 niveles diferentes de el KD-Tree
#Por ejemplo si deseas 4 niveles solo pones figsize=(5,5)
fig = plt.figure(figsize=(5, 5))# Aqui puedes cabiar los niveles
fig.subplots_adjust(wspace=0.1, hspace=0.15,
                    left=0.1, right=0.9,
                    bottom=0.05, top=0.9)
```

```
for level in range(1, 5):#GEneras los niveles
    #Si deseas 4 niveles  solo pones ...subplot(2,2,level,...)
    ax = fig.add_subplot(4, 4, level, xticks=[], yticks=[])#Aqui
    ax.scatter(X[:, 0], X[:, 1], s=9)
    KDT.draw_rectangle(ax, depth=level - 1)

    ax.set_xlim(-1.2, 1.2)
    ax.set_ylim(-0.15, 1.15)
    ax.set_title('level %i' % level)


fig.suptitle('$k$d-Tree')
plt.show()
```

Proofs

3.1. Proofs in 2-D

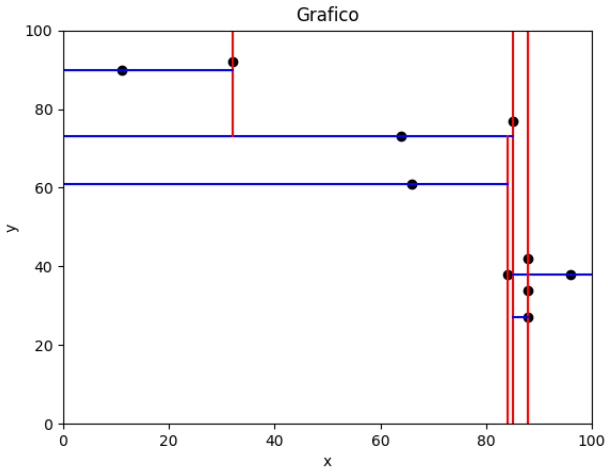


Figura 6: Proof 1

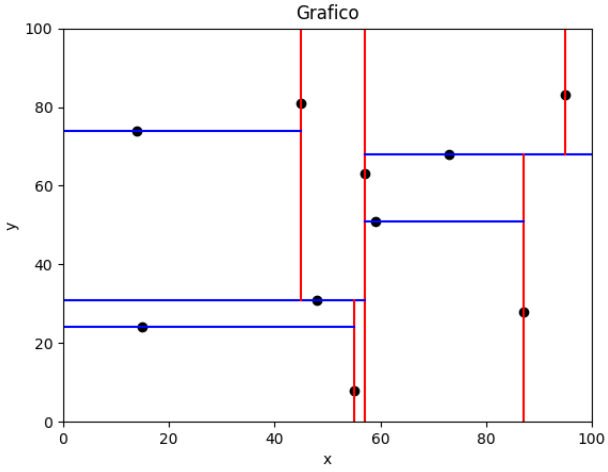


Figura 7: Proof 2

3.2. Proof using astroML in 2D

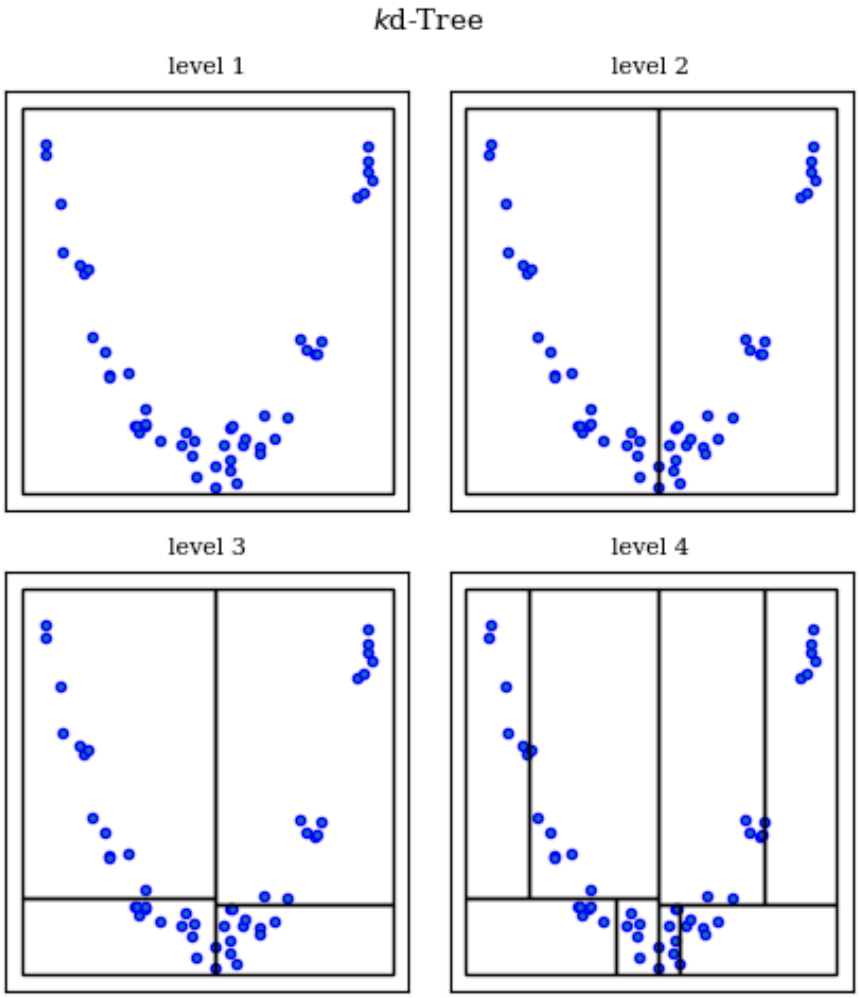


Figura 8: Proof 3 - 30 point in 4 levels

3.3. Proofs in 3D

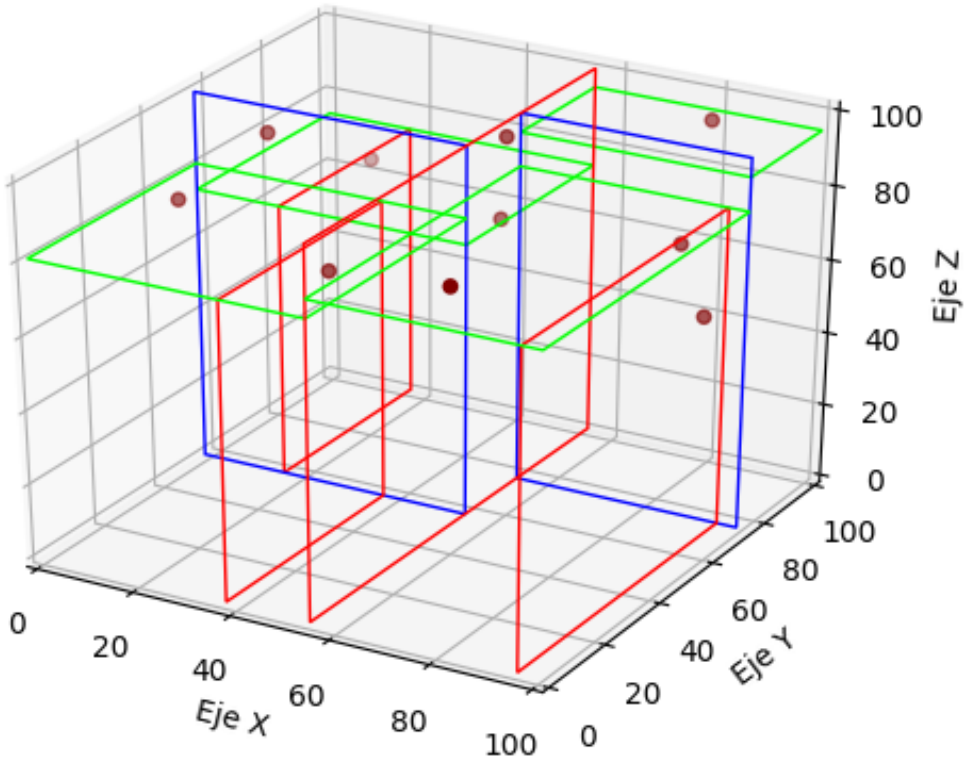


Figura 9: Proof 4

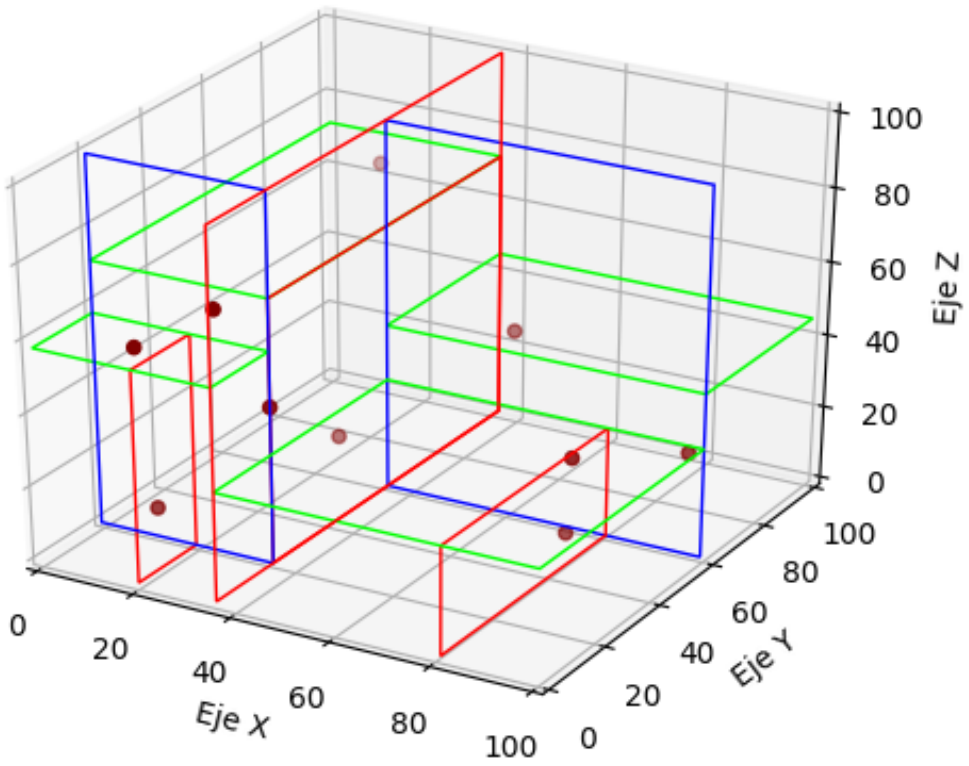


Figura 10: Proof 5

4. Refferences

- A Progressive k-d tree for Approximate k-Nearest Neighbors
  Jaemin Jo, Jinwook Seo, and Jean-Daniel Fekete Senior Member, IEEE
- Computational Geometry - Algorithms and Applications
  Third Edition - Springer