CSCE 4263/5183
Advanced Data Structures

Lecture 10

k-d Trees

Fall 2025

Prof. Khoa Luu
khoaluu@uark.edu

Major Topics In This Course

1. Introduction
2. Reviews (Link List, OOP, Binary Tree, BT Search)
3. Self-balancing Binary Search Tree (AVL, Multiway Search, Red-Black)
4. Splay Tree
5. Balanced Search Tree Review
6. Heap Methods
7. Hashing Methods
8. Data Structures in Deep Learning
9. Graph and Graph CNN
10. Final Project Presentations

Major Topics In This Course

1. Introduction
2. Reviews (Link List, OOP, Binary Tree, BT Search)
3. Self-balancing Binary Search Tree (AVL, Multiway Search, Red-Black)
4. Splay Tree
5. Balanced Search Tree Review
6. Heap Methods
7. Hashing Methods
8. Data Structures in Deep Learning
9. Graph and Graph CNN
10. Final Project Presentations

Outline

In this topic, we will cover:

- The idea behind a k-d tree
- Partitioning points in a k-dimensional space
- How sub-trees restrict the region of its elements
- Counting elements, searching, insertions, and nearest neighbor
- Difficulties with removals and balancing
- Other applications

k-d trees

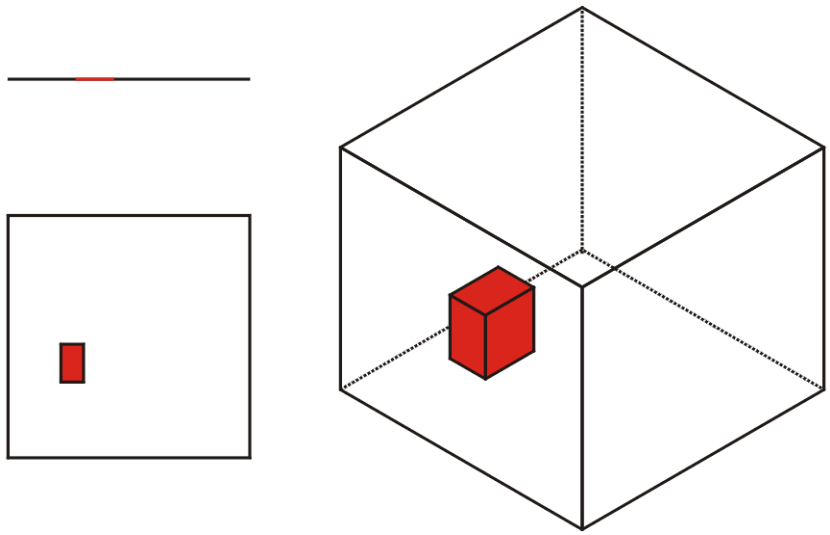
Used in image processing as a means of partitioning points in a k-dimensional space

Binary search trees are excellent at storing and searching data 1-dimensional (*i*.*e*., linear) space

k-d trees

As these examples demonstrate:

- with binary trees, it is easy to find all points which fall inside a given interval (a 1d box)
- how do we store points so that we can find all points which fall within a
  - rectangle (a 2d box), or
  - a 3d box?



k-d trees

To solve this problem, we will define a binary tree where

- depths 0, k, $2k$, etc., contain binary search nodes sorted on the 1st coordinate
- depths 1, $k+1$, $2k+1$, etc., contain binary search nodes nodes sorted on the 2nd coordinate,

and in general

- depths j, $k+j$, $2k+j$, etc., contain binary search nodes sorted on the $(k+1)$st coordinate

k-d trees

Comment on the naming convention:

- while the k refers to the dimension, it is common to refer to such a tree as, for example, a 3-dimensional k-d tree instead of a 3-d tree, as numerous data structures could be referred to as 3-d trees

k-d trees

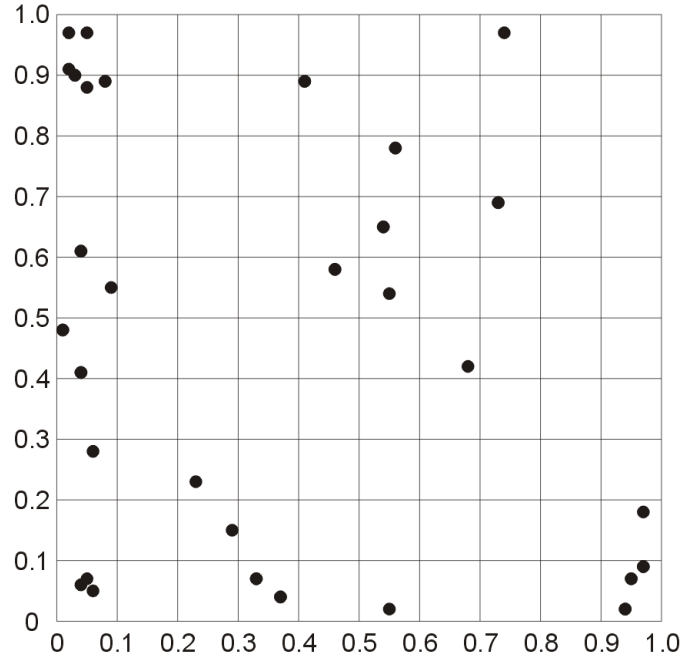
Given a set of points, we can construct a balanced k-d tree by always using the median element with respect the coordinate being searched on

We will assume no duplication, but if duplication occurs, we can choose any of the median elements

k-d trees

Suppose we wish to partition the following points in a 2-dimensional k-d tree:

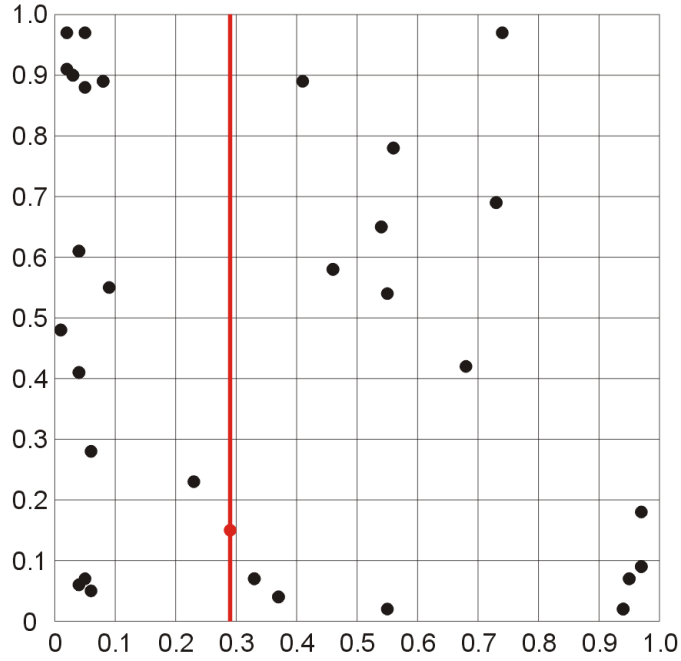
(0.03, 0.90), (0.37, 0.04), (0.56, 0.78),
(0.01, 0.48), (0.41, 0.89), (0.95, 0.07),
(0.97, 0.09), (0.54, 0.65), (0.04, 0.61),
(0.73, 0.69), (0.46, 0.58), (0.08, 0.89),
(0.04, 0.41), (0.94, 0.02), (0.33, 0.07),
(0.55, 0.54), (0.06, 0.05), (0.04, 0.06),
(0.74, 0.97), (0.29, 0.15), (0.05, 0.88),
(0.23, 0.23), (0.55, 0.02), (0.02, 0.97),
(0.05, 0.07), (0.06, 0.28), (0.09, 0.55),
(0.02, 0.91), (0.05, 0.97), (0.68, 0.42),
(0.97, 0.18)



k-d trees

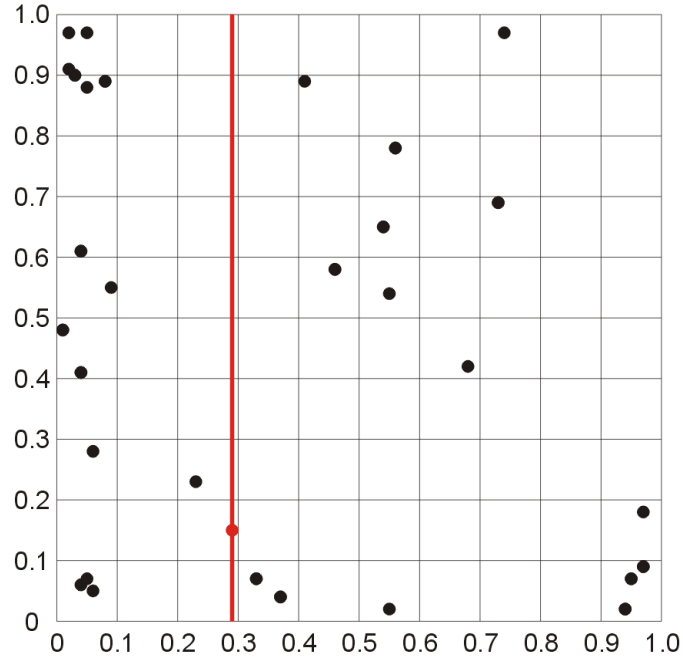
The first step is to order the points based on the 1st coordinate and find the median:

(0.01, 0.48), (0.02, 0.91), (0.02, 0.97), (0.03, 0.90),
(0.04, 0.06), (0.04, 0.41), (0.04, 0.61), (0.05, 0.07),
(0.05, 0.88), (0.05, 0.97), (0.06, 0.05), (0.06, 0.28),
(0.08, 0.89), (0.09, 0.55), (0.23, 0.23), (**0.29**, 0.15),
(0.33, 0.07), (0.37, 0.04), (0.41, 0.89), (0.46, 0.58),
(0.54, 0.65), (0.55, 0.02), (0.55, 0.54), (0.56, 0.78),
(0.68, 0.42), (0.73, 0.69), (0.74, 0.97), (0.94, 0.02),
(0.95, 0.07), (0.97, 0.09), (0.97, 0.18)



k-d trees

The median point, (0.29, 0.15), forms the root of our k-d tree



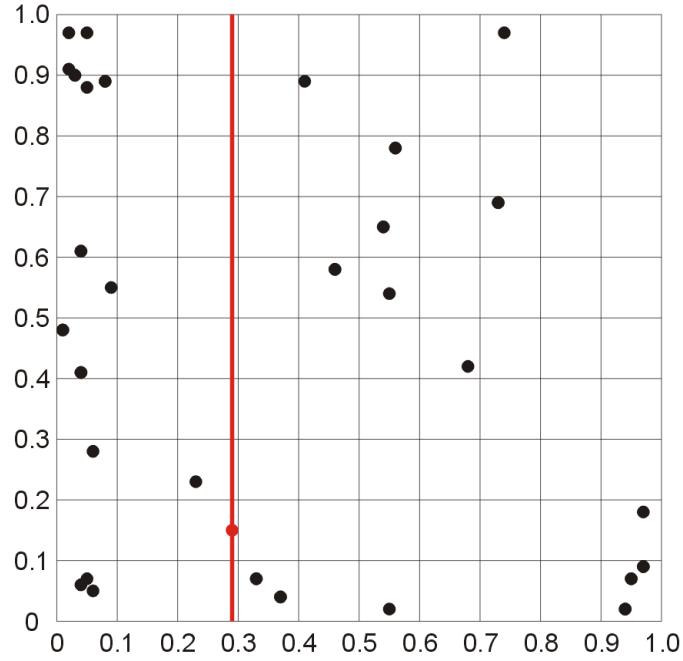
(0.29, 0.15)

k-d trees

This partitions the remaining points into two sets:

{(0.01, 0.48), (0.02, 0.91), (0.02, 0.97), (0.03, 0.90),
(0.04, 0.06), (0.04, 0.41), (0.04, 0.61), (0.05, 0.07),
(0.05, 0.88), (0.05, 0.97), (0.06, 0.05), (0.06, 0.28),
(0.08, 0.89), (0.09, 0.55), (0.23, 0.23)}

{(0.33, 0.07), (0.37, 0.04), (0.41, 0.89), (0.46, 0.58),
(0.54, 0.65), (0.55, 0.02), (0.55, 0.54), (0.56, 0.78),
(0.68, 0.42), (0.73, 0.69), (0.74, 0.97), (0.94, 0.02),
(0.95, 0.07), (0.97, 0.09), (0.97, 0.18)}

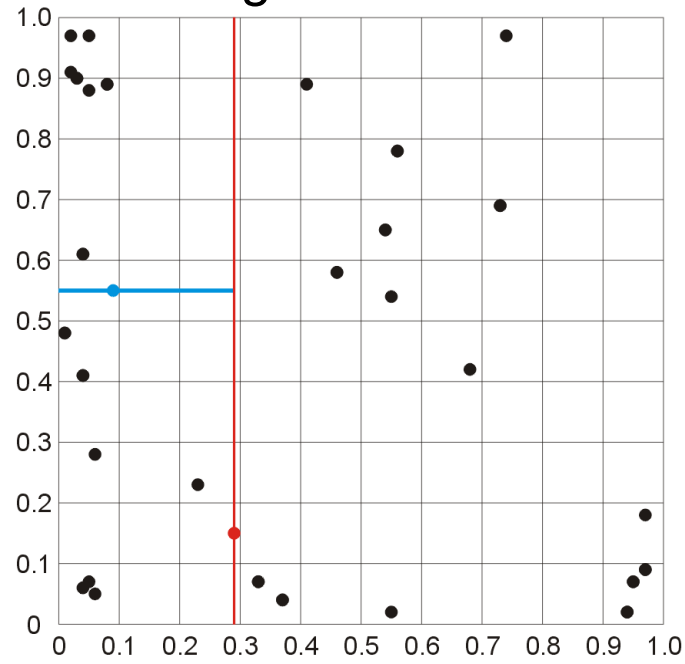


(0.29, 0.15)

k-d trees

Starting with the first partition, we order these according to the 2nd coordinate:

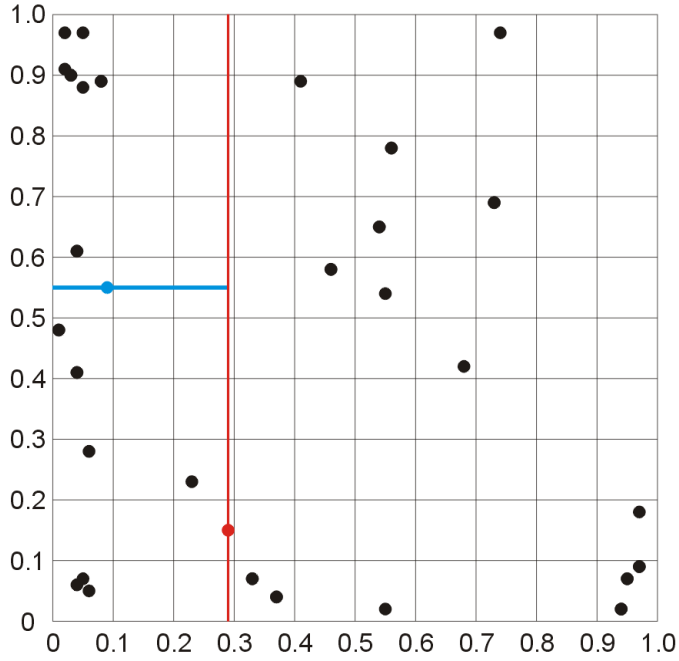
(0.06, 0.05), (0.04, 0.06), (0.05, 0.07), (0.23, 0.23),
(0.06, 0.28), (0.04, 0.41), (0.01, 0.48), (0.09, **0.55**),
(0.04, 0.61), (0.03, 0.90), (0.02, 0.91), (0.02, 0.97),
(0.05, 0.88), (0.08, 0.89), (0.05, 0.97)



(0.29, 0.15)

k-d trees

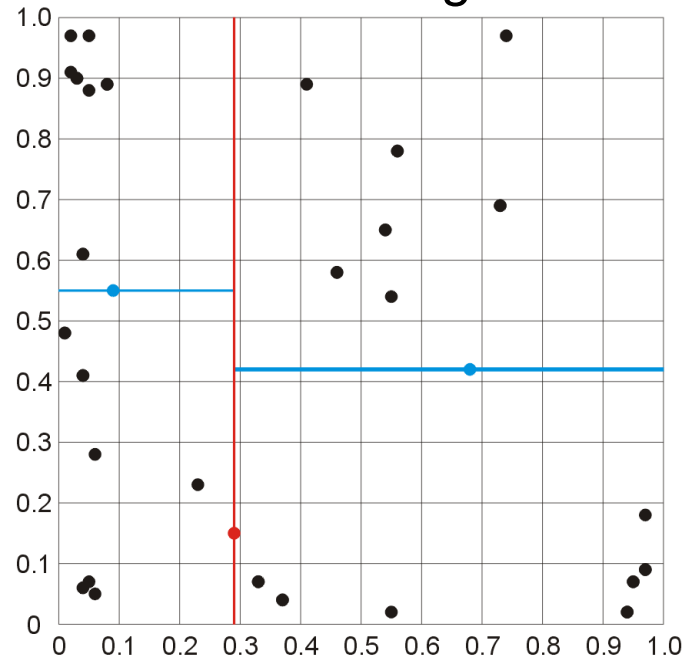
This point creates the left child of the root



k-d trees

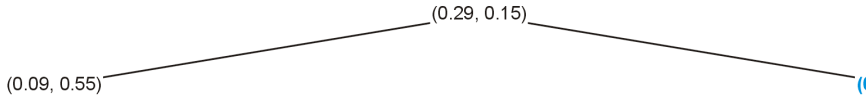
Starting with the second partition, we also order these according to the 2nd coordinate:

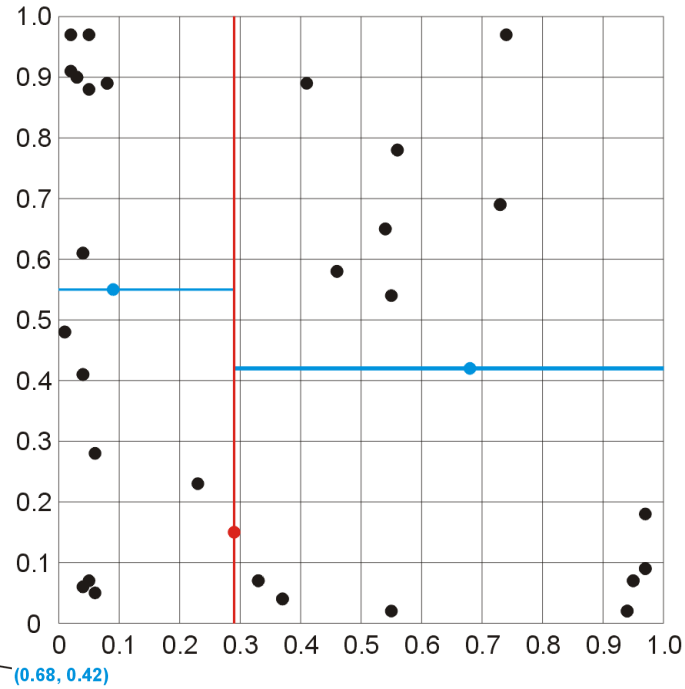
(0.55, 0.02), (0.94, 0.02), (0.37, 0.04), (0.33, 0.07),
(0.95, 0.07), (0.97, 0.09), (0.97, 0.18), (0.68, **0.42**),
(0.55, 0.54), (0.46, 0.58), (0.54, 0.65), (0.73, 0.69),
(0.56, 0.78), (0.41, 0.89), (0.74, 0.97)



k-d trees

This point creates the right child of the root



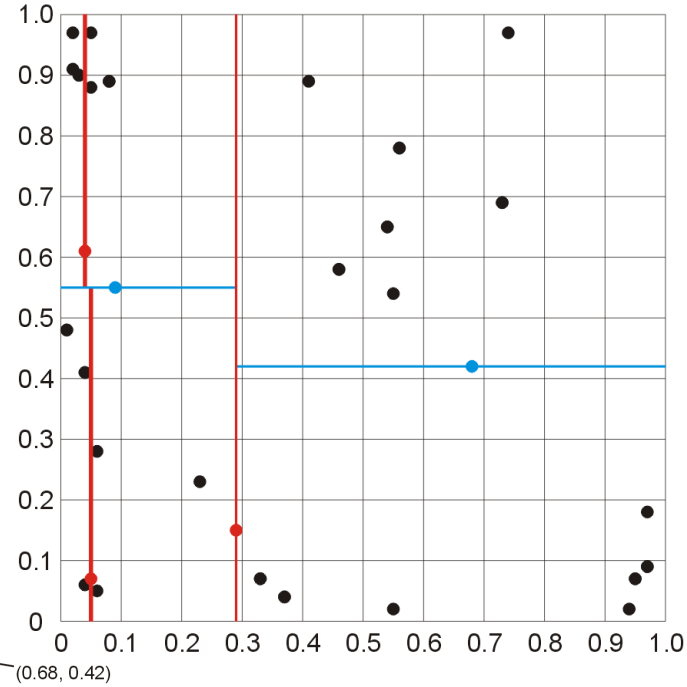


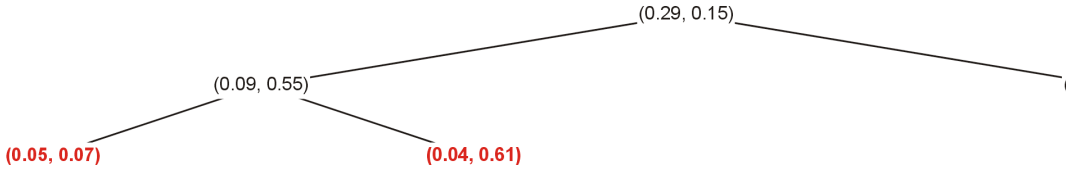
k-d trees

Next, ordering the partitioned elements by the 1st coordinate, we choose the medians to find the children of the left child (0.09, 0.55):

(0.01, 0.48), (0.04, 0.06), (0.04, 0.41), (**0.05**, 0.07), (0.06, 0.05), (0.06, 0.28), (0.23, 0.23),

(0.02, 0.91), (0.02, 0.97), (0.03, 0.90), (**0.04**, 0.61), (0.05, 0.88), (0.05, 0.97), (0.08, 0.89)



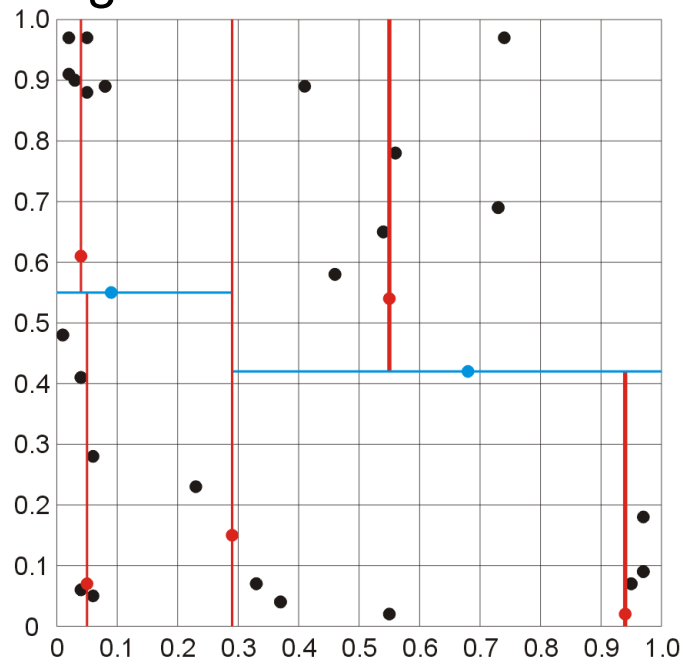


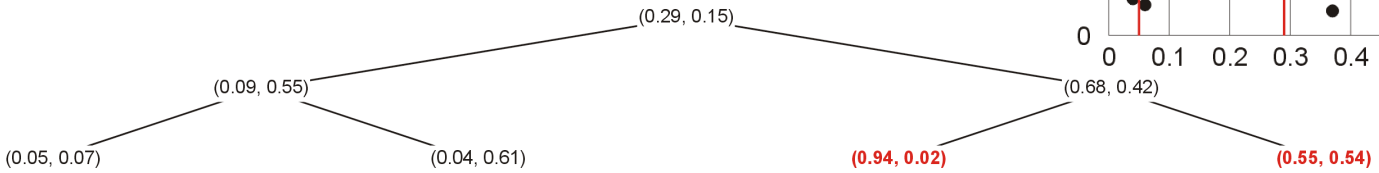
k-d trees

Doing the same with the two right partitions, we get the children of the right child of the root:

(0.33, 0.07), (0.37, 0.04), (0.55, 0.02), (**0.94,** 0.02),
(0.95, 0.07), (0.97, 0.09), (0.97, 0.18)

(0.41, 0.89), (0.46, 0.58), (0.54, 0.65), (**0.55**, 0.54),
(0.56, 0.78), (0.73, 0.69), (0.74, 0.97)

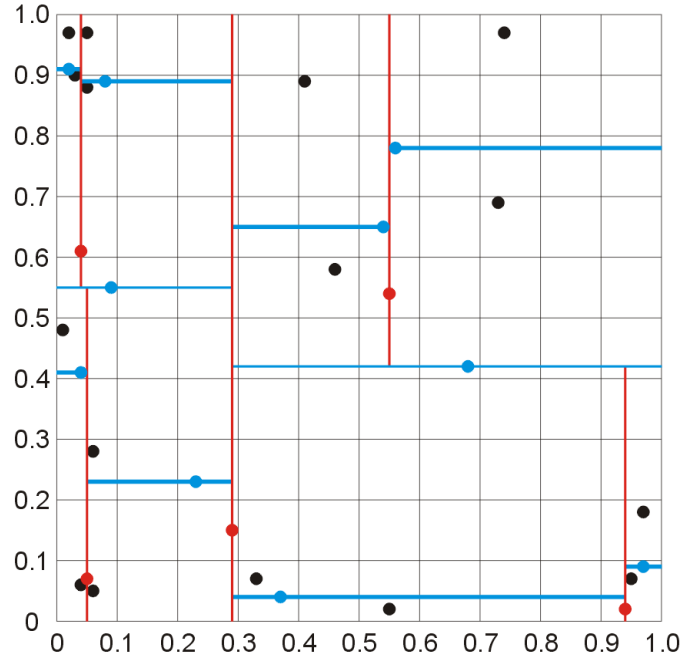


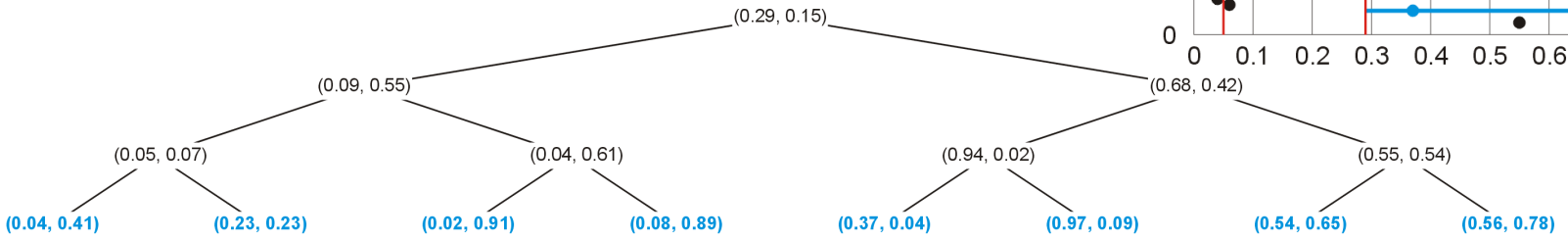


k-d trees

At the next level, we order the points again based on the 2nd coordinate and choose the medians:

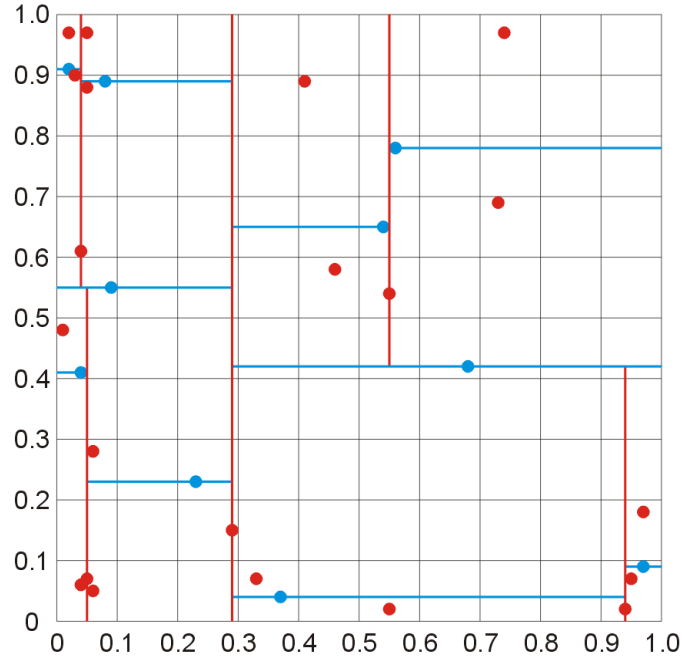
(0.04, 0.06), (0.04, **0.41**), (0.01, 0.48)
(0.06, 0.05), (0.23, **0.23**), (0.06, 0.28)
(0.03, 0.90), (0.02, **0.91**), (0.02, 0.97)
(0.05, 0.88), (0.08, **0.89**), (0.05, 0.97)
(0.55, 0.02), (0.37, **0.04**), (0.33, 0.07)
(0.95, 0.07), (0.97, **0.09**), (0.97, 0.18)
(0.46, 0.58), (0.54, **0.65**), (0.41, 0.89)
(0.73, 0.69), (0.56, **0.78**), (0.74, 0.97)

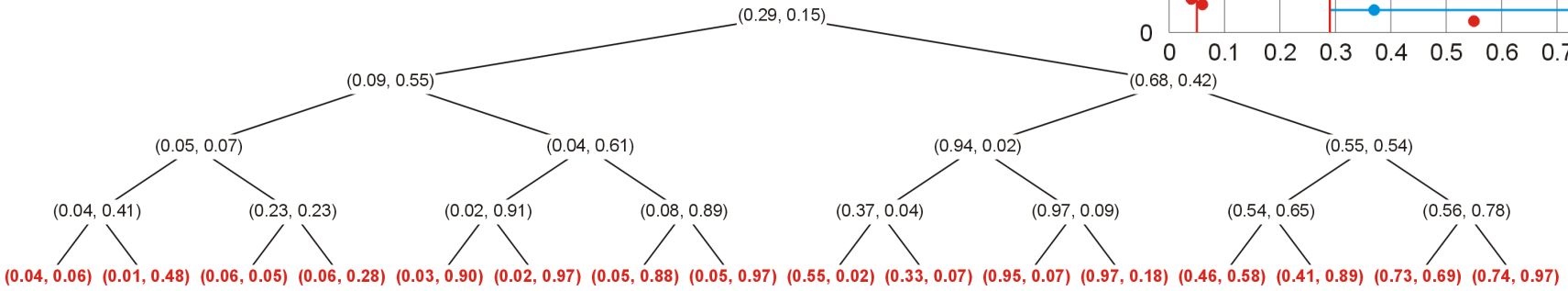




k-d trees

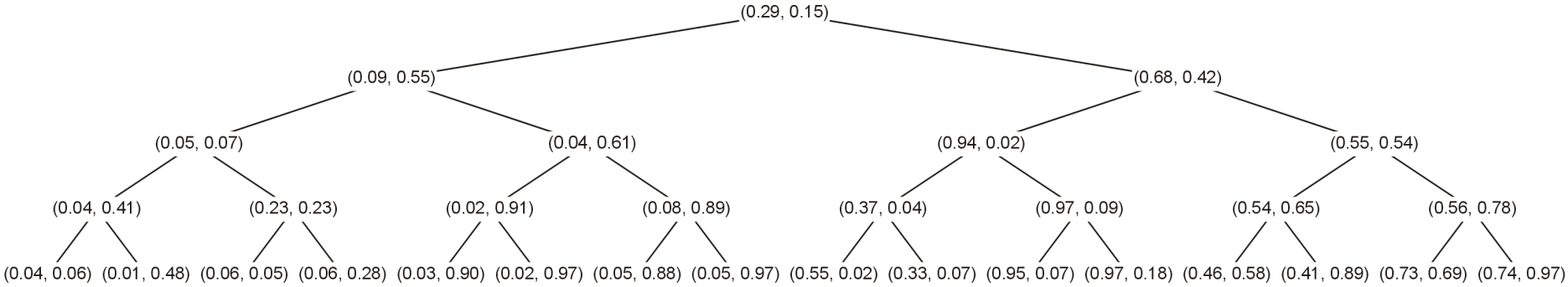
The remaining points fit in their appropriate locations





k-d trees

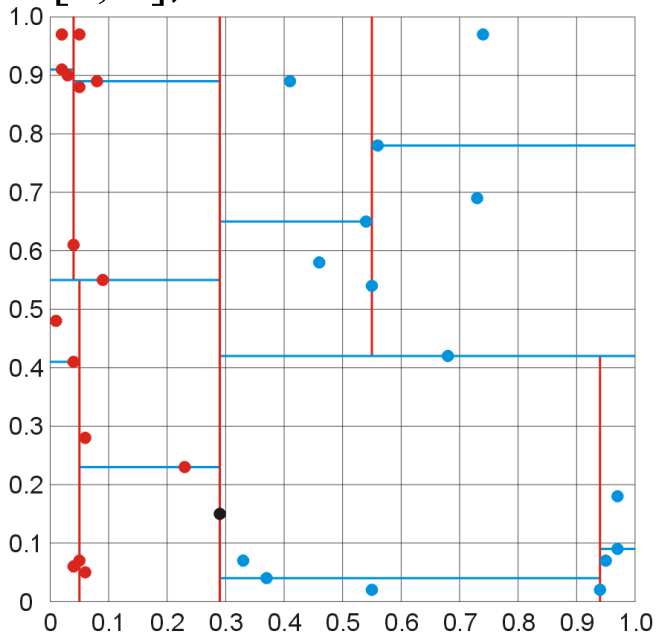
The result is a 2-dimensional k-d tree of the given 31 points

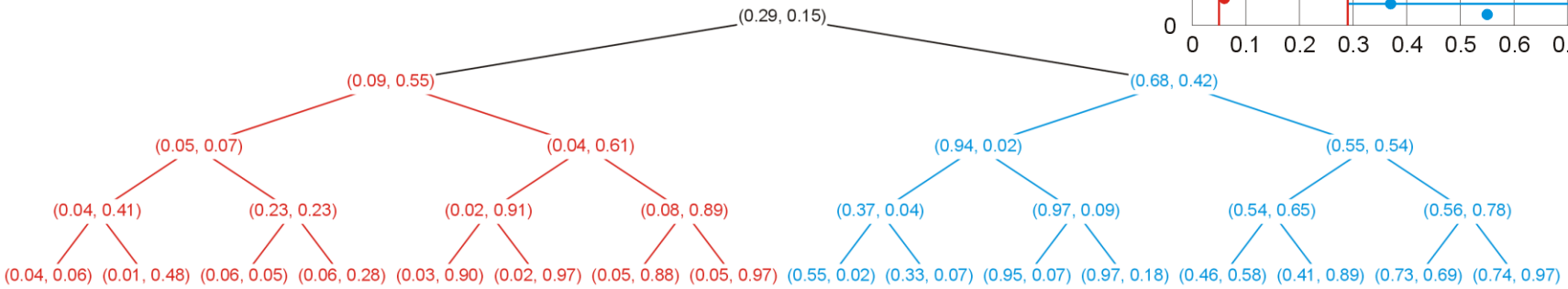


k-d trees

In this example, all points are in the box $[0, 1] \times [0, 1]$, however:

- all points in the left sub-tree are in the box $[0, 0.29] \times [0, 1]$
- all points in the right sub-tree are in the box $[0.29, 1] \times [0, 1]$

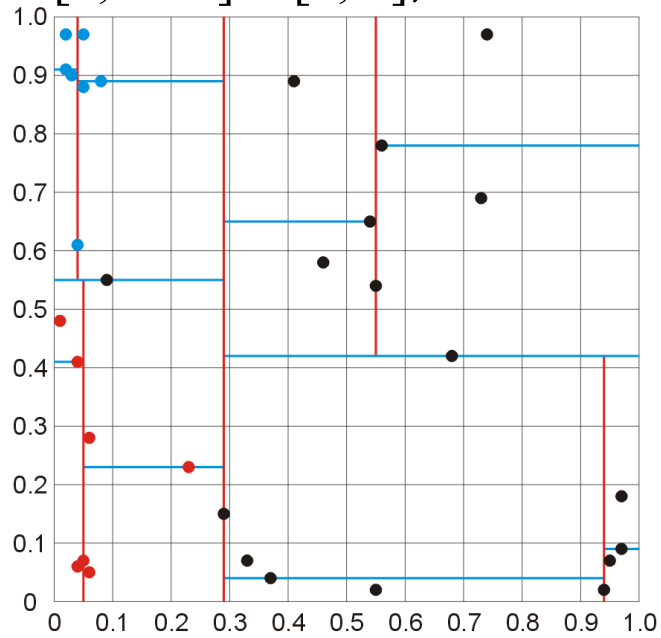


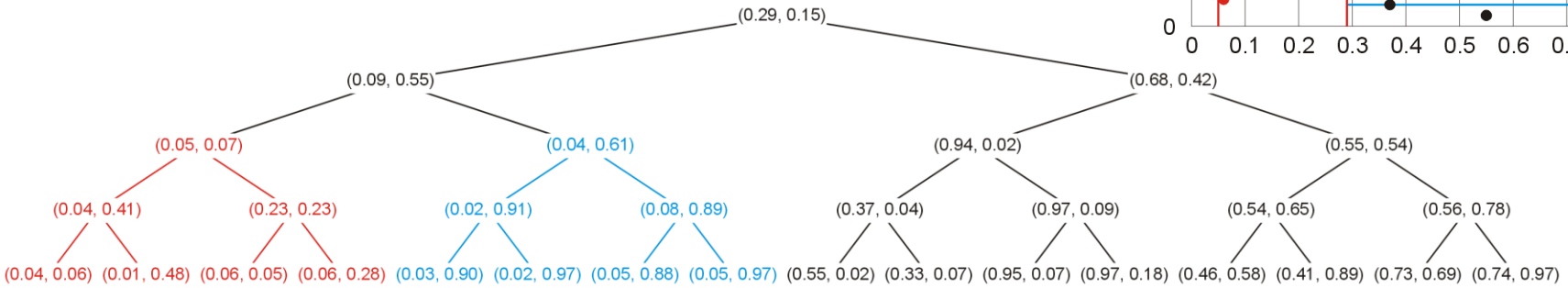


k-d trees

Looking at the left sub-tree, all points fall within $[0, 0.29] \times [0, 1]$, and

- all points in its left sub-tree are in $[0, 0.29] \times [0, 0.55]$
- all points in its right sub-tree are in $[0.29, 1] \times [0.55, 1]$

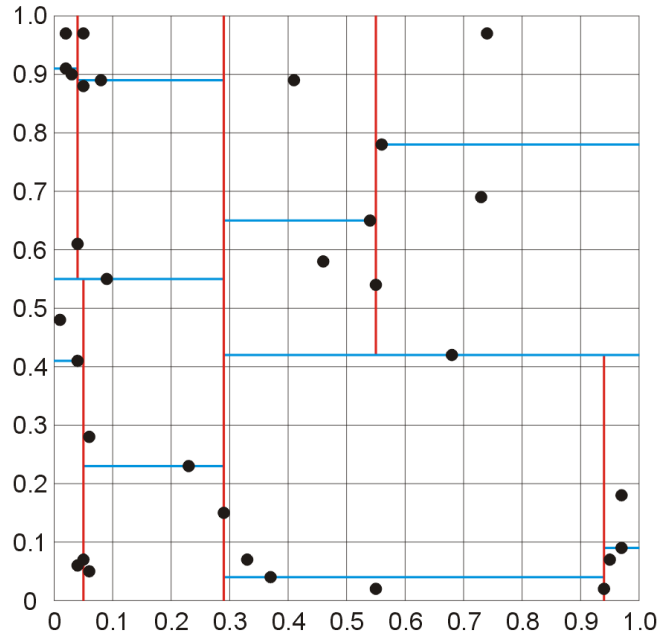


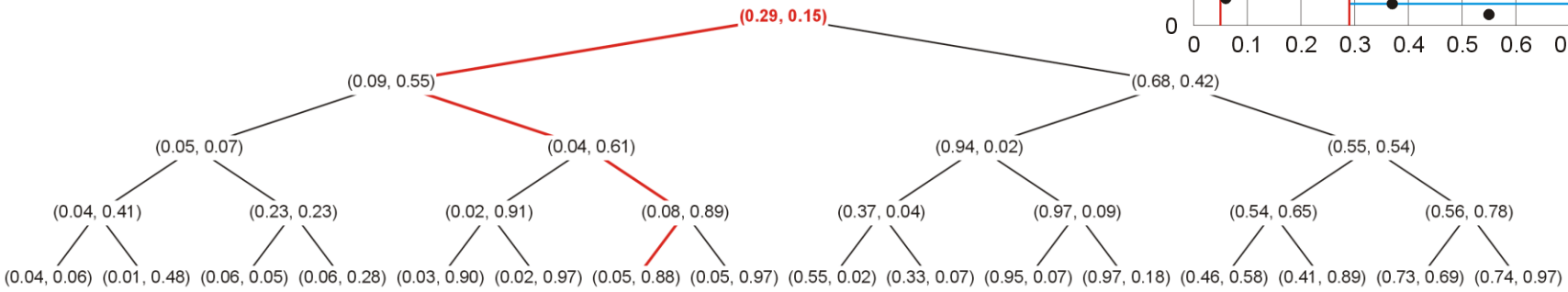


k-d trees

Following paths similarly limits the regions of the points within the sub-trees

In this example we will follow the red path starting at the root

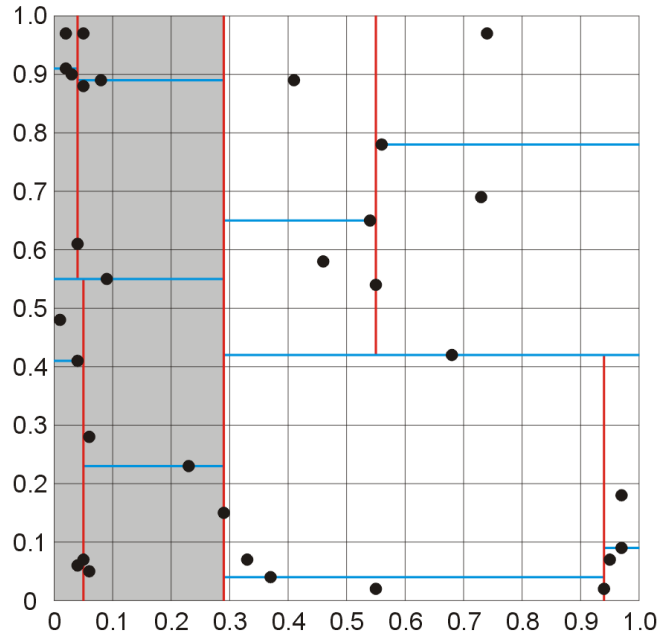


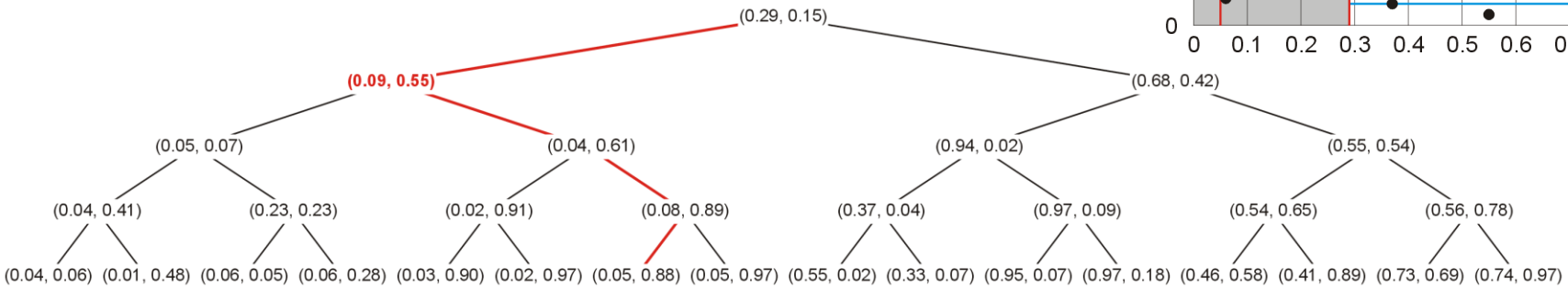


k-d trees

All points with (0.09, 0.55) as its root appear in the shaded area

This region is restricted to $[0, 0.29] \times [0, 1]$



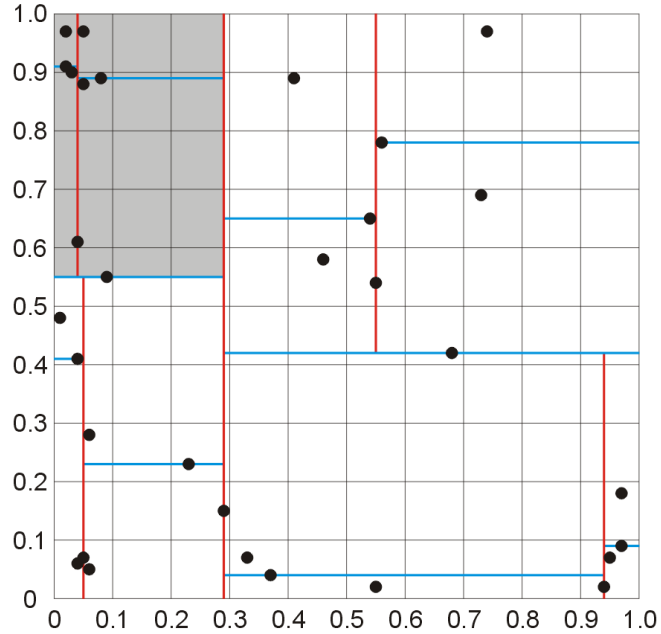


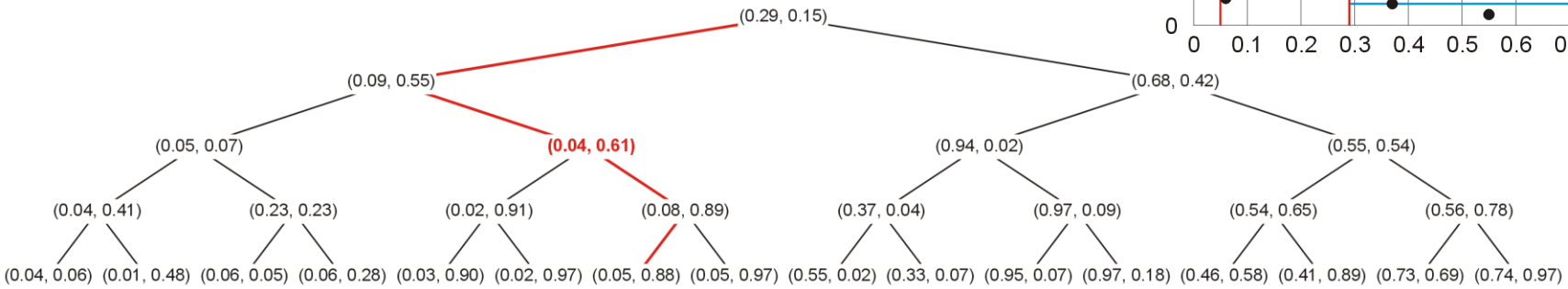
k-d trees

Moving down the tree, the points under (0.04, 0.61) are further restricted

This restriction is in the 2nd coordinate:

$$[0, 0.29] \times [0.55, 1]$$

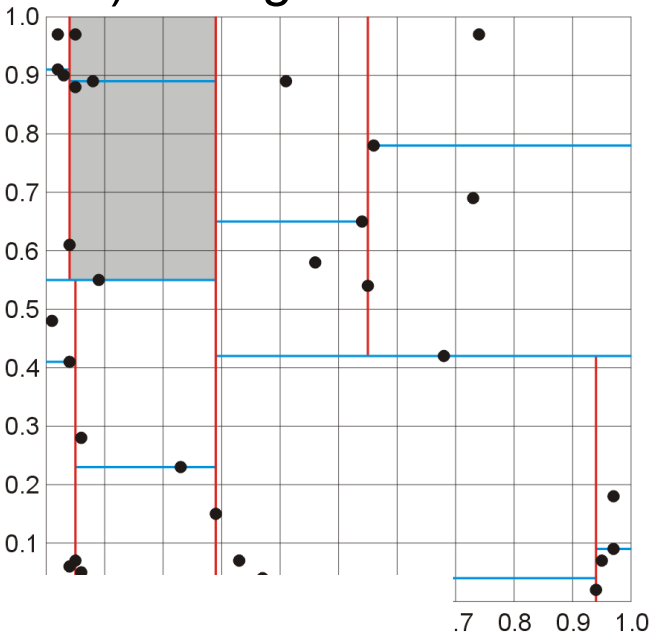


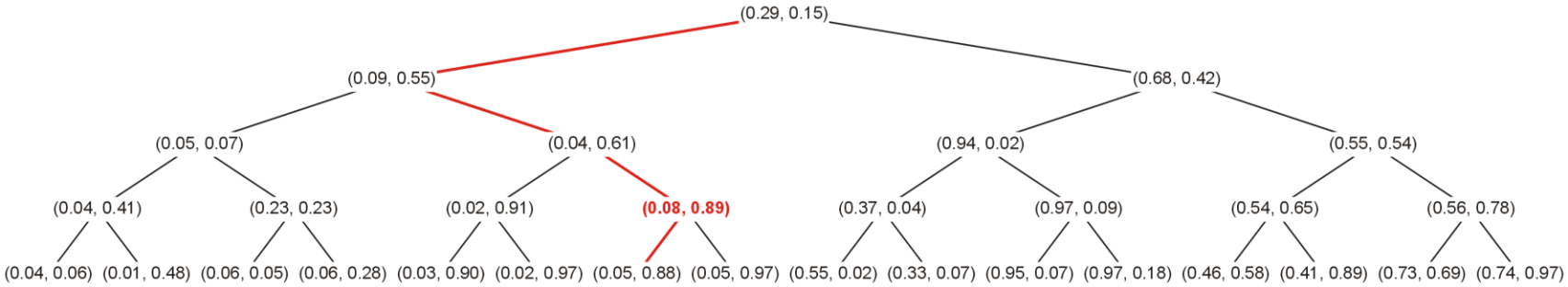


k-d trees

Stepping further down, all points below (0.08, 0.89) are again further restricted to

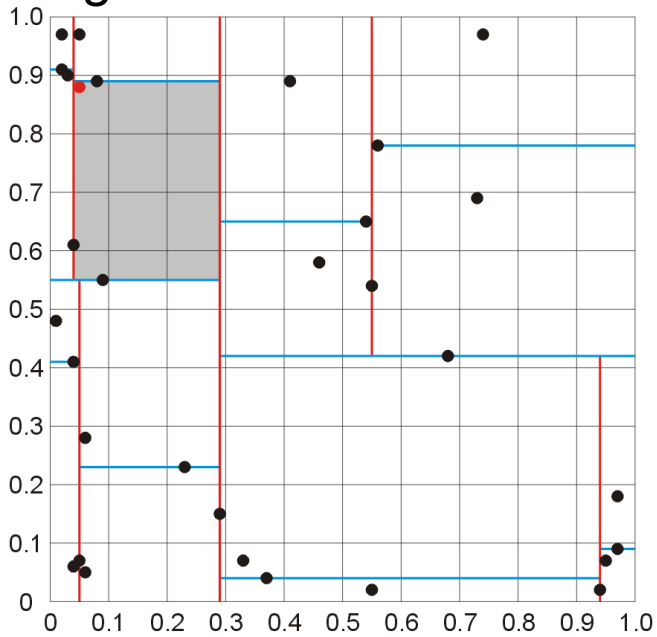
$$[0.08, 0.29] \times [0.55, 1]$$

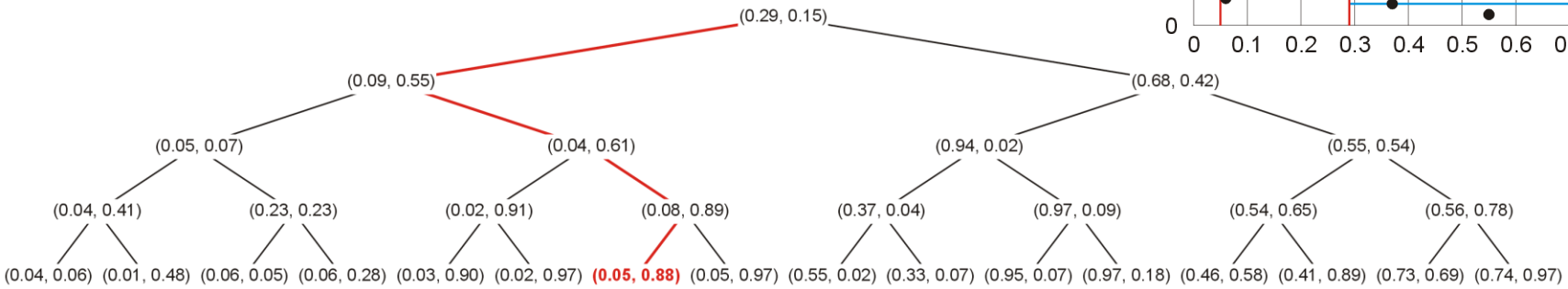




k-d trees

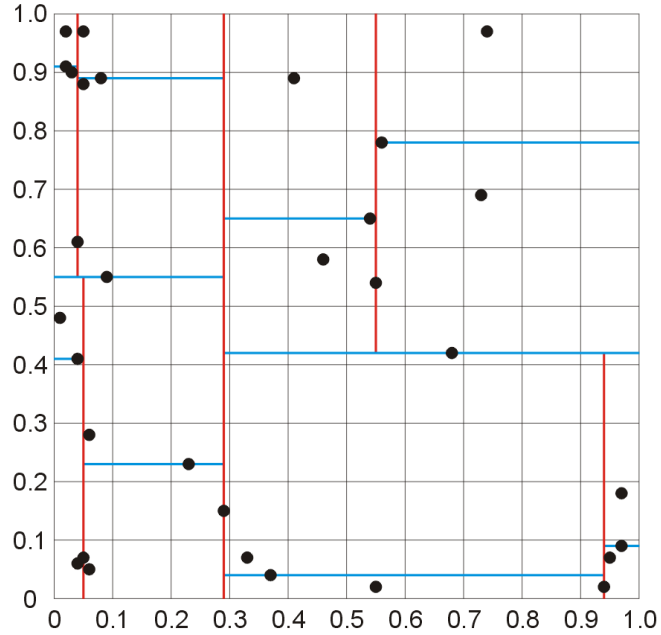
Finally, the last point, a leaf node, falls within the given box





k-d trees

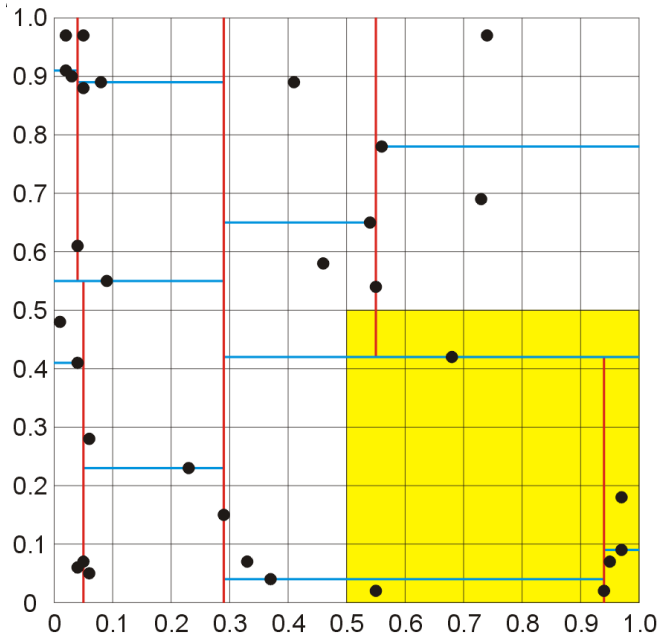
A useful application of a k-d tree provides an efficient data structure for counting the number of points which fall within a given k-dimensional rectangle



k-d trees

This is used in image processing: locating objects within a scene, ray tracing, etc.

Find the points which lie in the quadrant $[0.5, 1] \times [0, 0.5]$



k-d trees

The traversal rules we will follow are:

- we always match the coordinate corresponding to the level we are current at
- if that coordinate is less than the corresponding interval of the box, we only need to visit the right sub-tree
- if that coordinate is greater than the corresponding interval, we need only visit the left sub-tree
- otherwise, we check if the root is in the box and we visit both sub-trees

k-d trees

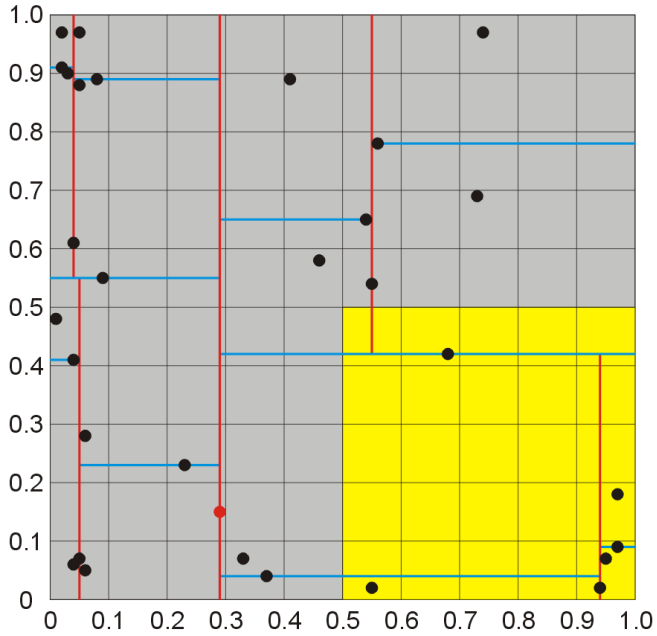
There is one special case:

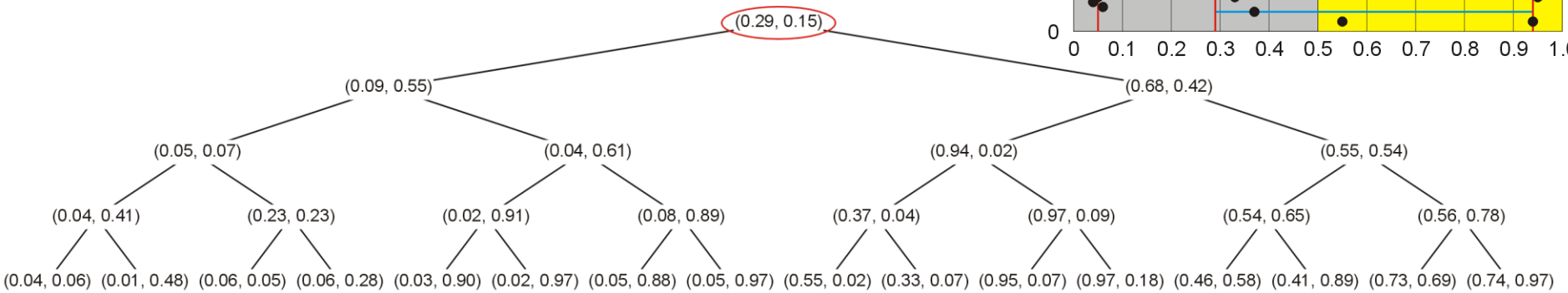
- Because we can track the sub-interval under which a tree may fall, it may be possible to add an entire sub-tree

k-d trees

Starting at the root:

- We examine the 1st coordinate
- Note: $0.29 < [0.5, 1]$
- We visit only the right sub-tree





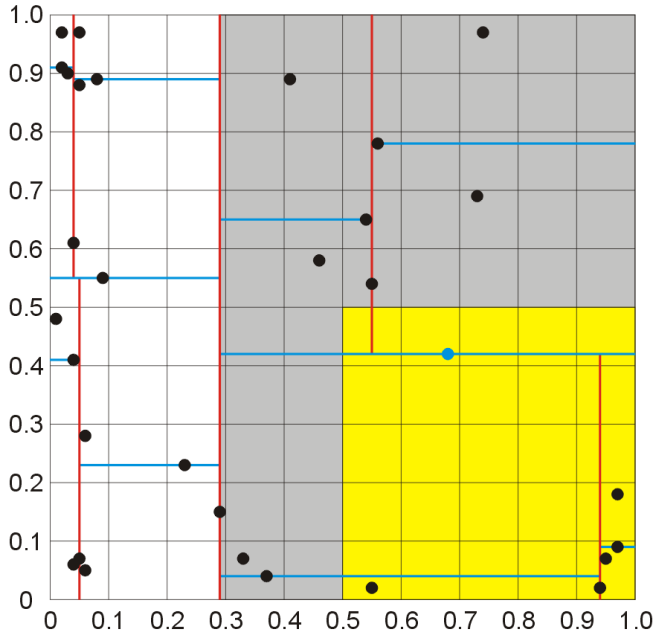
k-d trees

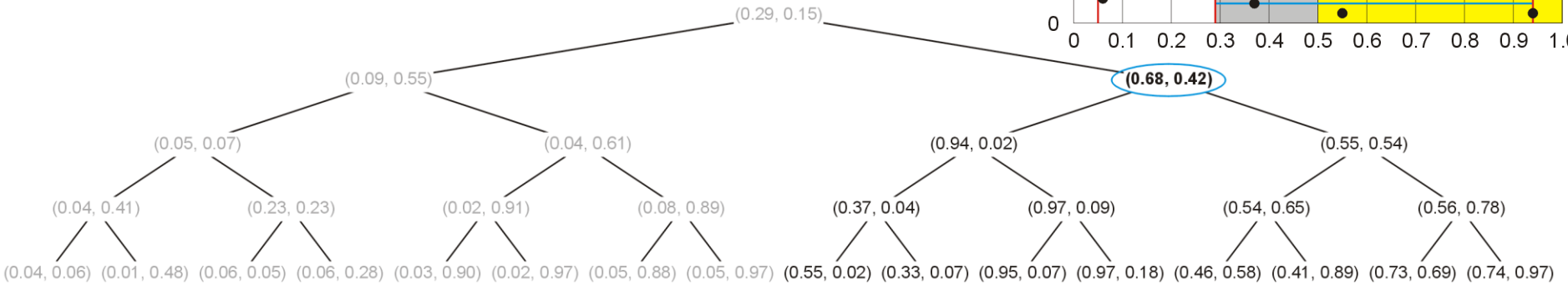
Visiting the right sub-tree: $0.42 \in [0, 0.5]$

Also

$(0.68, 0.42) \in [0.5, 1] \times [0, 0.5]$

and we visit both sub-trees





k-d trees

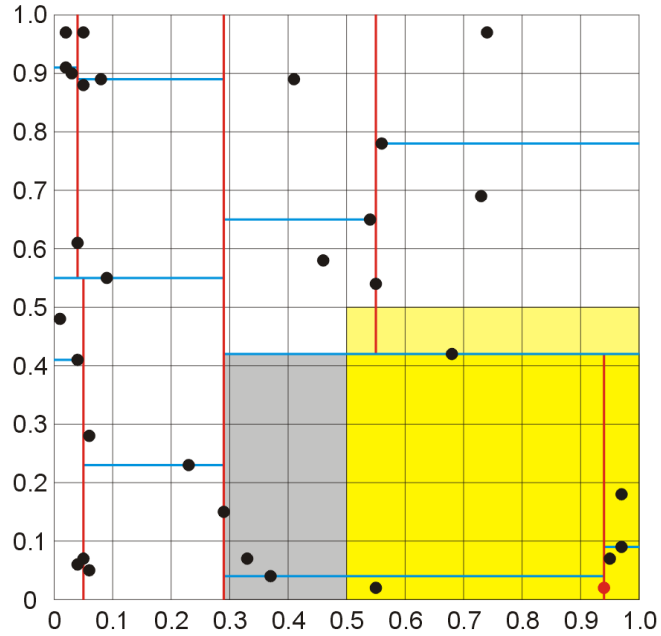
Starting with the left sub-tree:

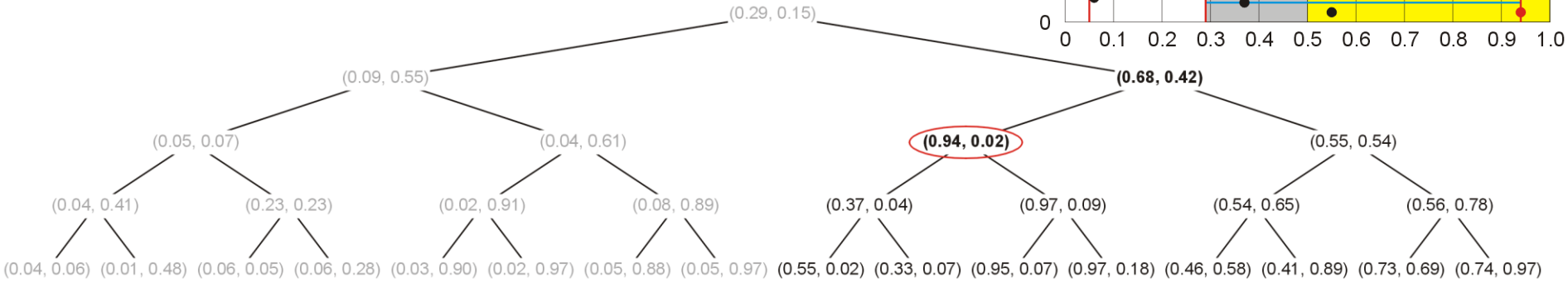
$0.94 \in [0.5, 1]$

We note that

$(0.94, 0.02) \in [0.5, 1] \times [0, 0.5]$

and we visit both sub-trees



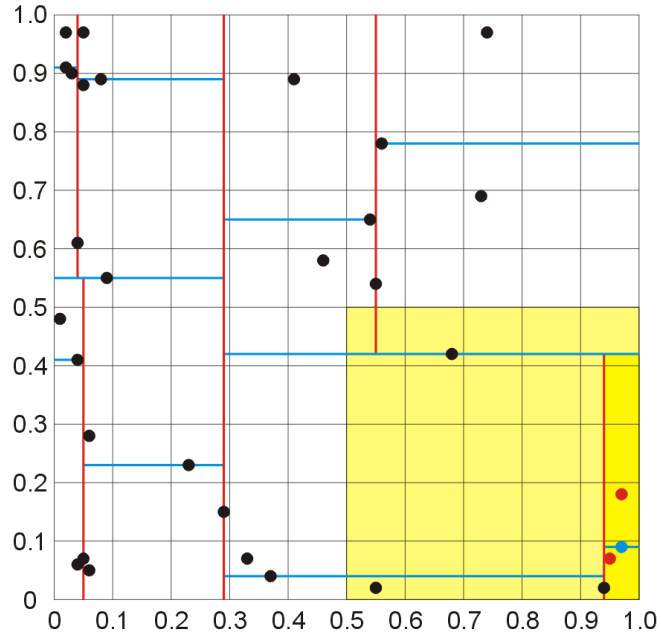


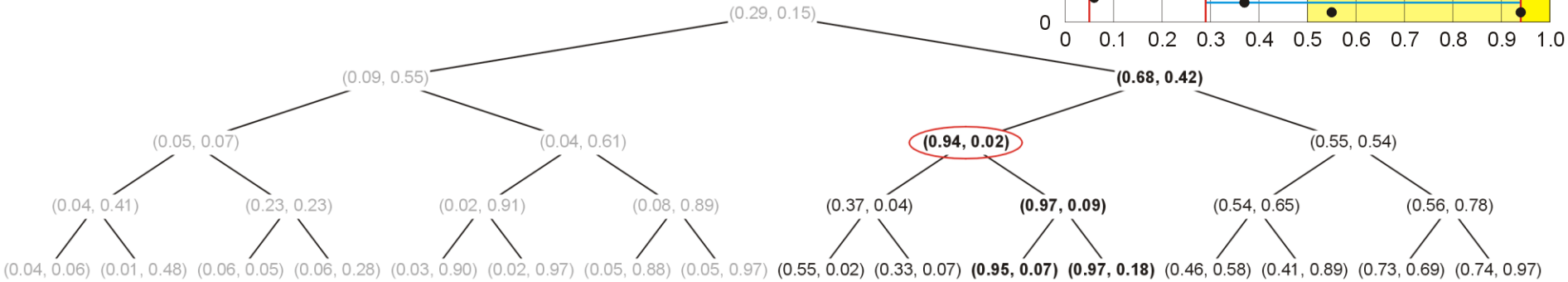
k-d trees

However, at this point, we notice that the right sub-tree is restricted to $[0.94, 1] \times [0, 0.42]$ and

$$[0.94, 1] \times [0, 0.42] \subseteq [0.5, 1] \times [0, 0.5]$$

Thus, the entire right sub-tree is in the box





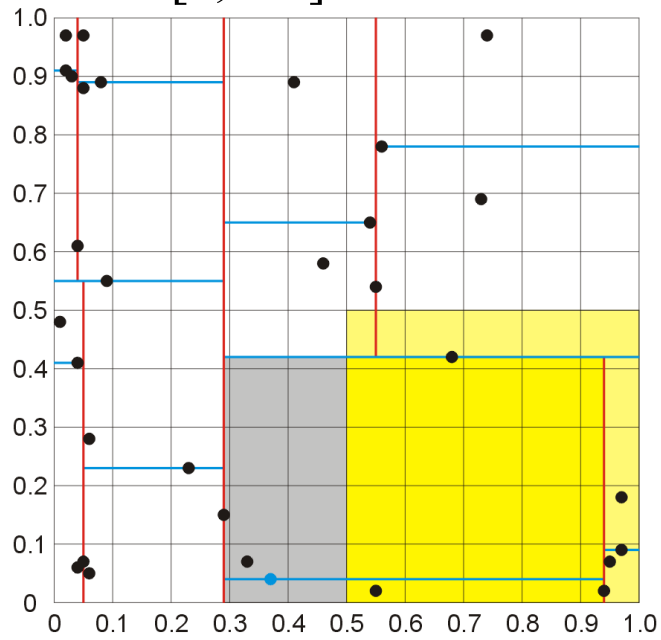
k-d trees

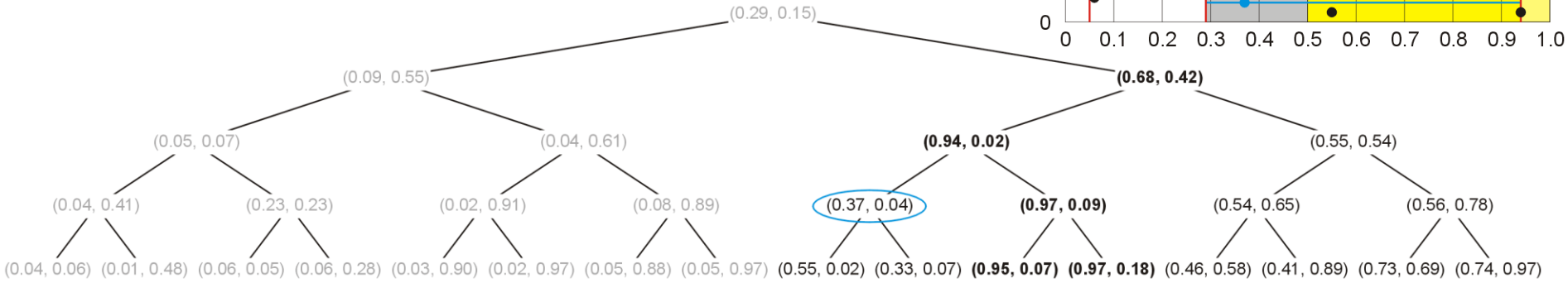
Continuing with the left sub-tree, we note that $0.04 \in [0, 0.5]$

We note that

$(0.37, 0.04) \notin [0.5, 1] \times [0, 0.5]$

but we still visit both sub-trees



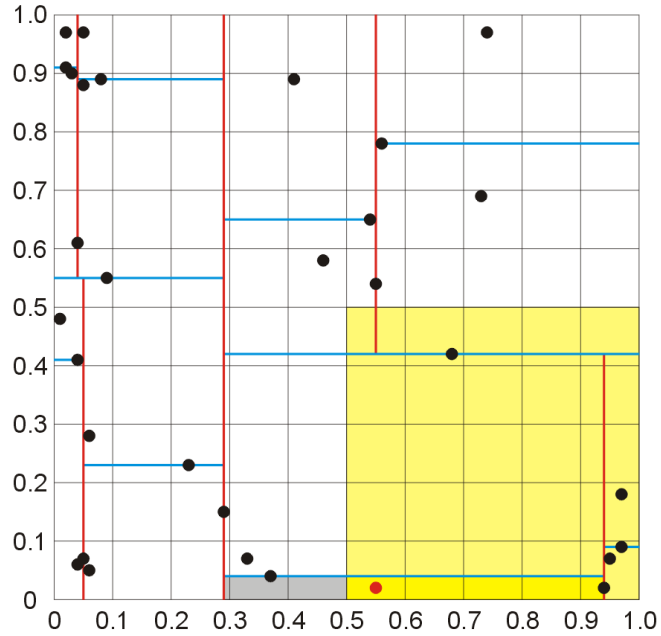


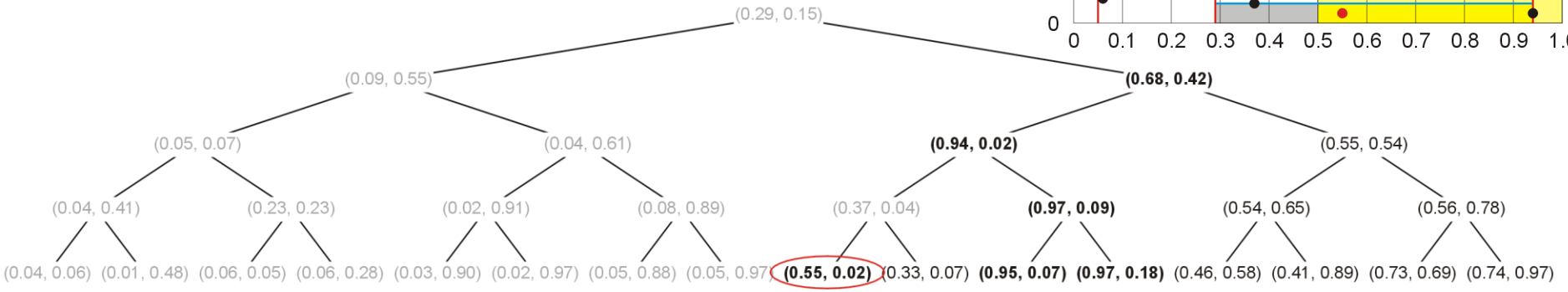
k-d trees

Inspecting the left leaf node

$(0.55, 0.02) \in [0.5, 1] \times [0, 0.5]$

and we are finished



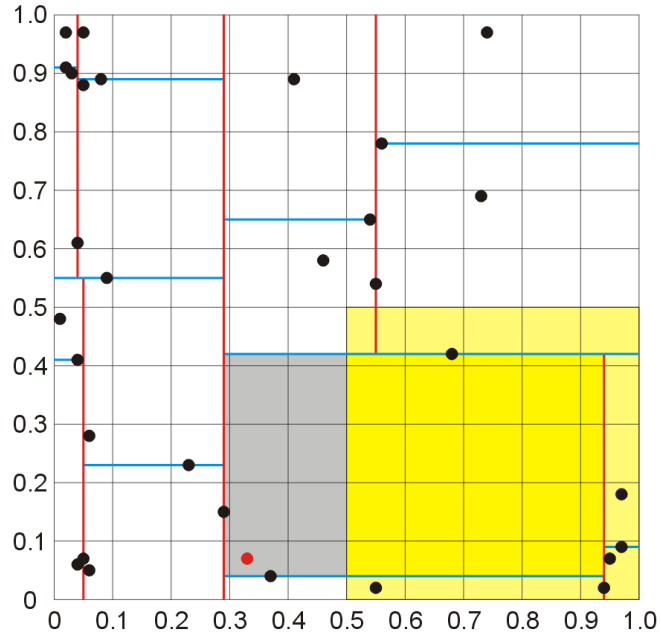


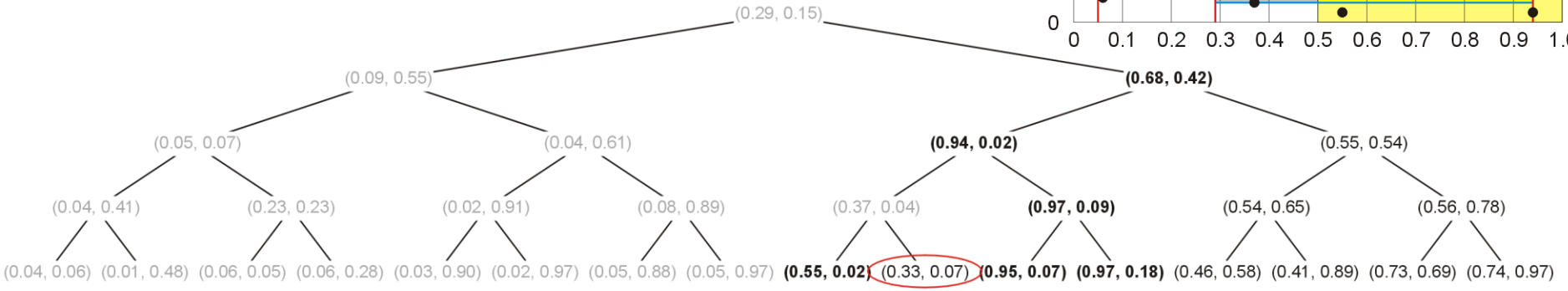
k-d trees

Inspecting the right leaf node, we note:

$(0.33, 0.07) \notin [0.5, 1] \times [0, 0.5]$

and we are finished



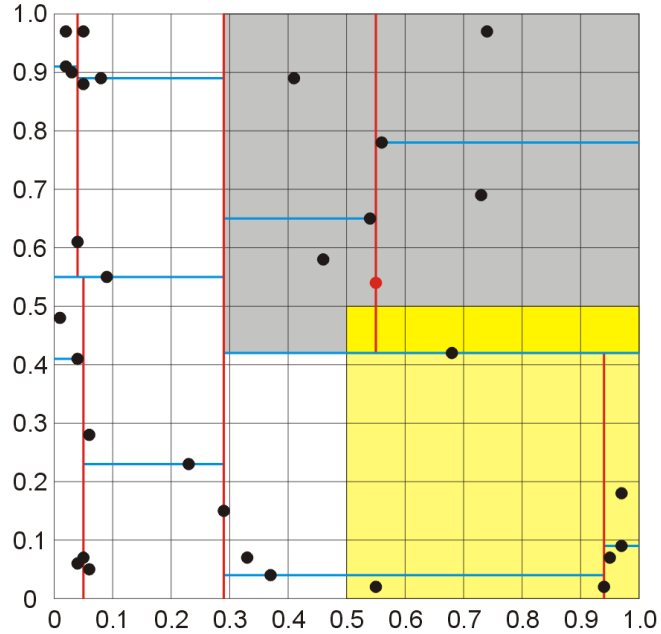


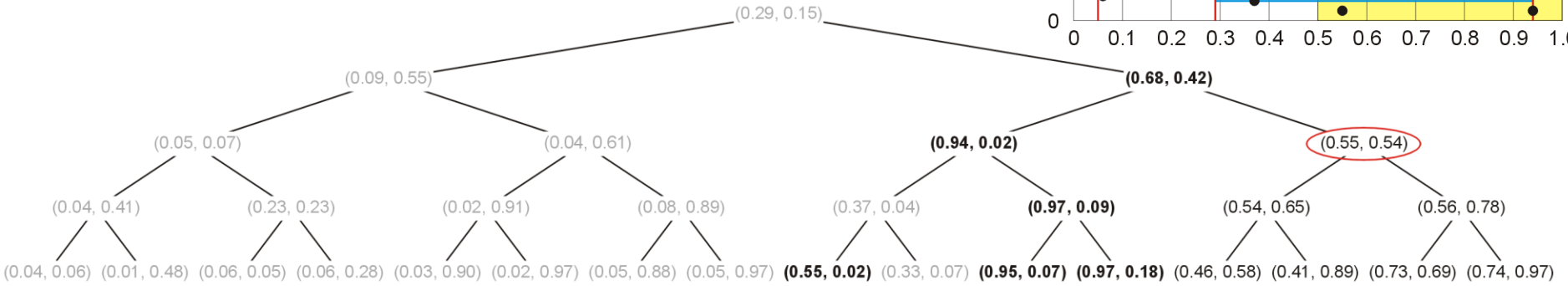
k-d trees

Continuing with the right sub-tree, we note that $0.55 \in [0.5, 1]$ (we must visit both sub-trees)

However, the point is not in the box

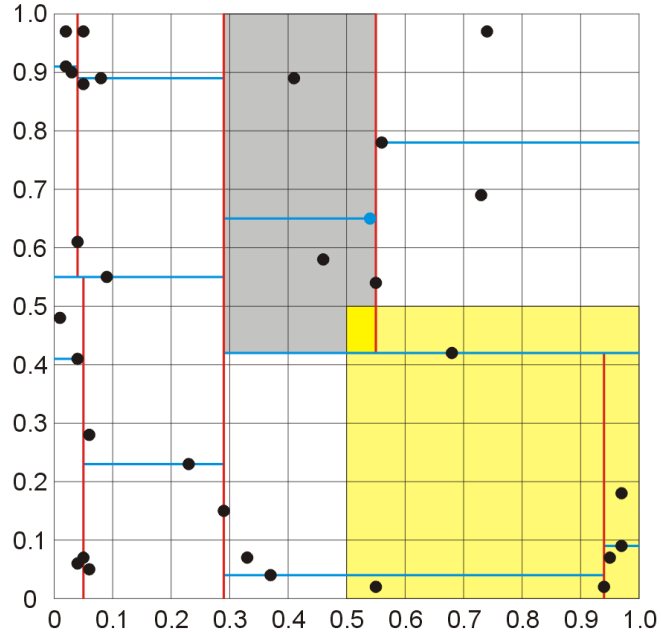
$$[0.5, 1] \times [0, 0.5]$$

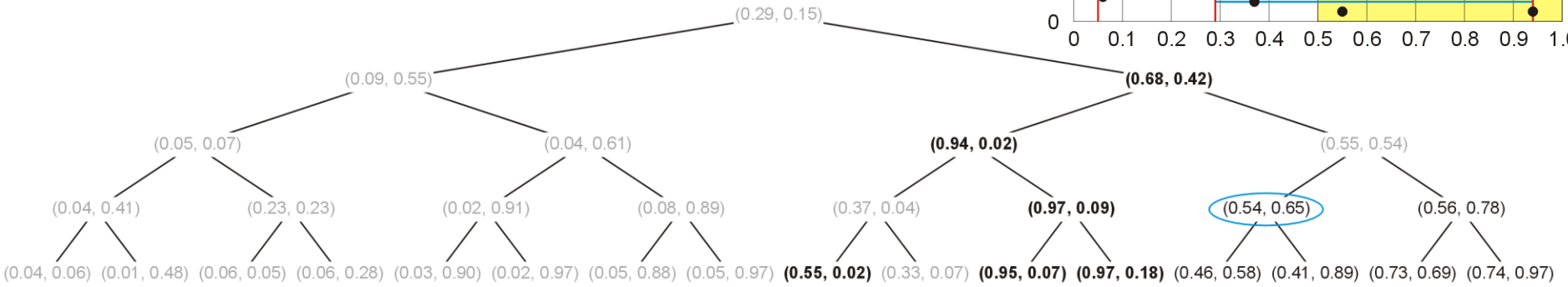




k-d trees

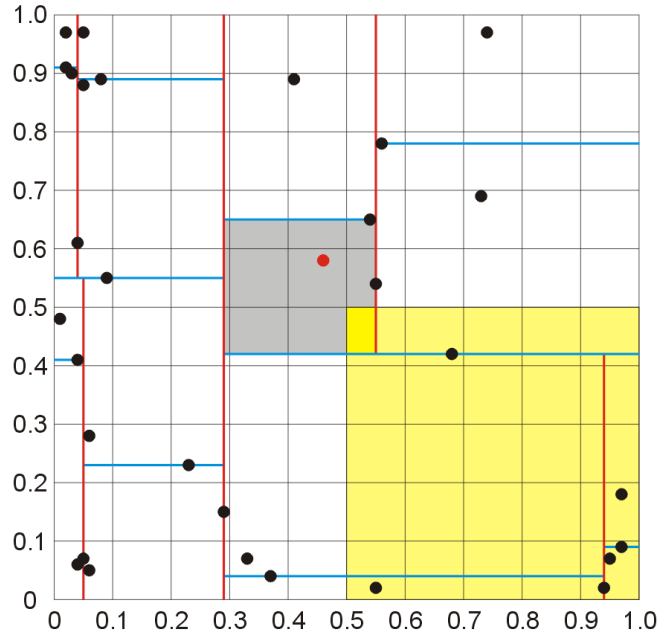
Visiting the next node, we note $(0.54, 0.65)$ is not in the box and $0.65 > 0.5$, hence we need only visit the left sub-tree

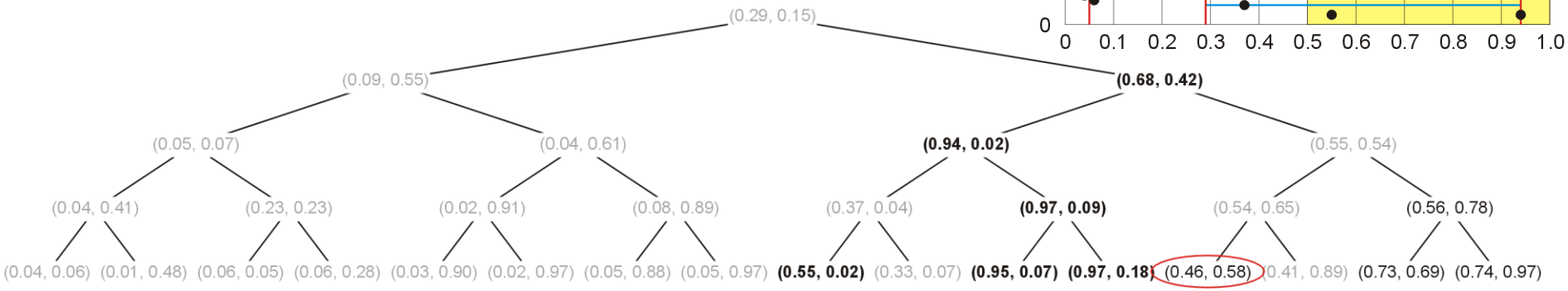




k-d trees

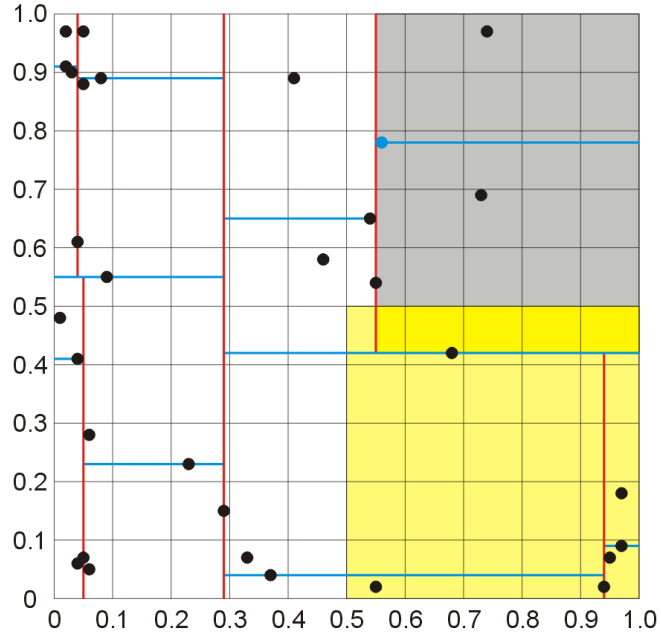
This leaf node is not in the region

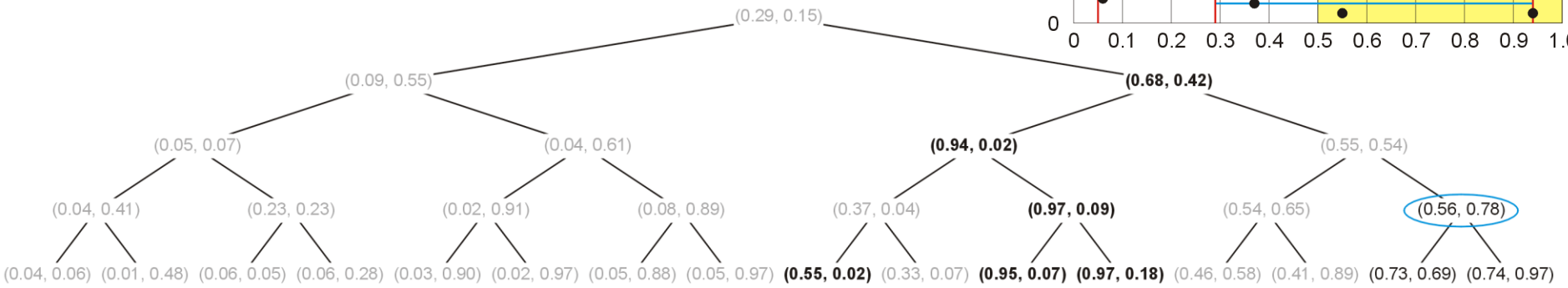




k-d trees

Visiting the next node, we note $(0.56, 0.78)$ is not in the box and $0.56 > 0.5$, hence we need only visit the left sub-tree

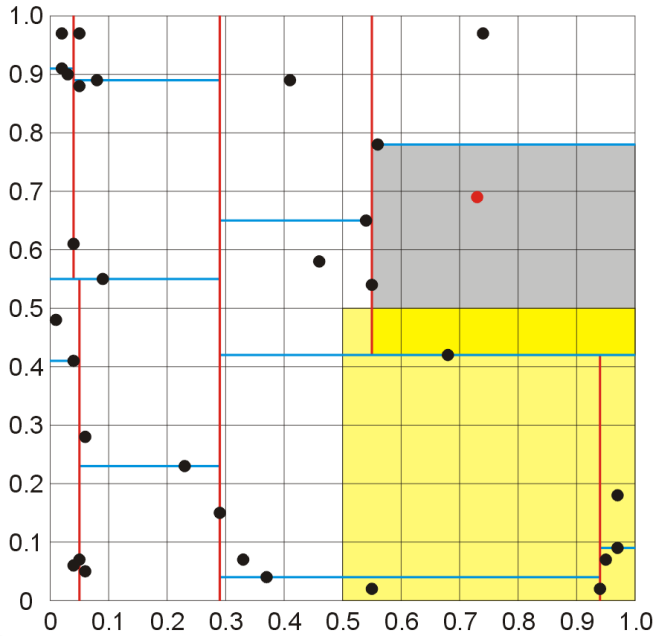


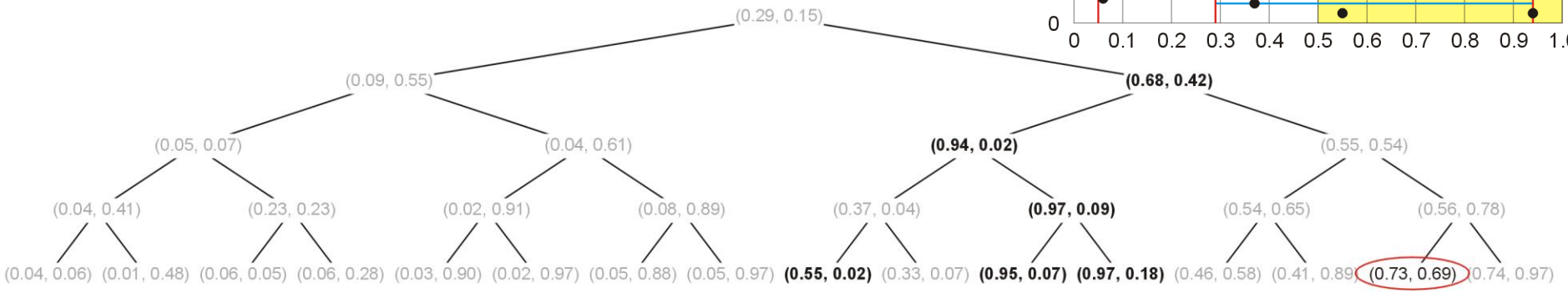


k-d trees

This leaf node is not in the region

- The traversal is finished

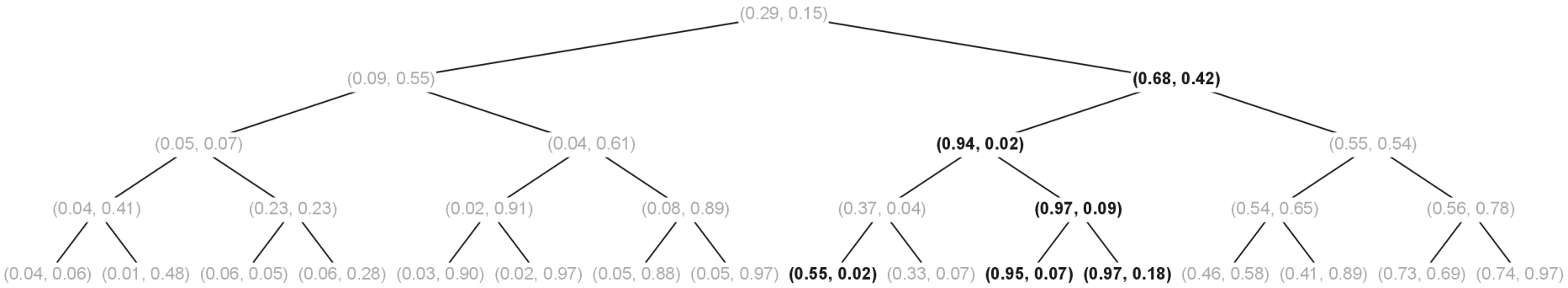




k-d trees

Thus, there are six points in the quadrant $[0.5, 1] \times [0, 0.5]$

- We had to visit fewer than half the points in the tree to determine this

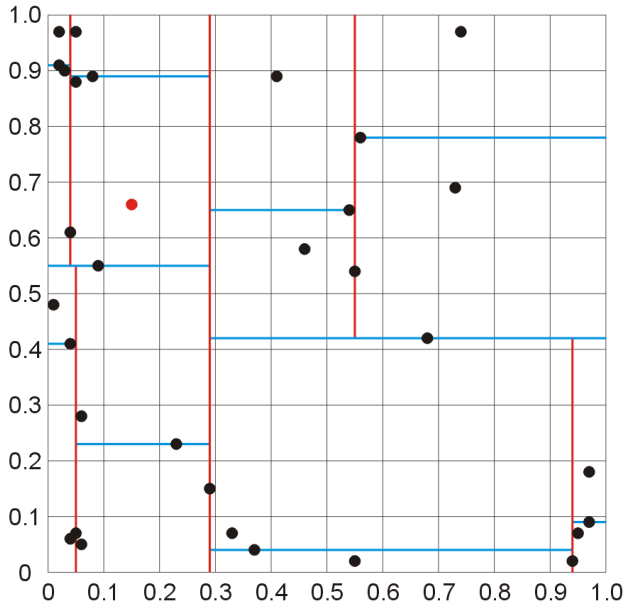


k-d trees

Suppose we want to add a new element to an existing k-d tree

For example, add the point (0.15, 0.66) to the k-d tree which we just built:

- we add the new element the same way we did with binary search trees, however, we must compare the appropriate coordinate at each level

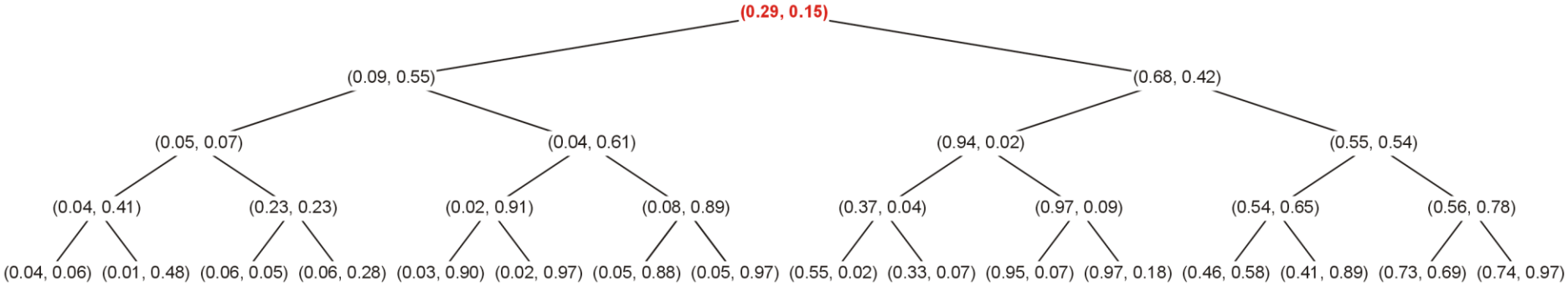


k-d trees

First, we compare the first coordinate:

$0.15 < 0.29$

Thus we add the new element to the left sub-tree

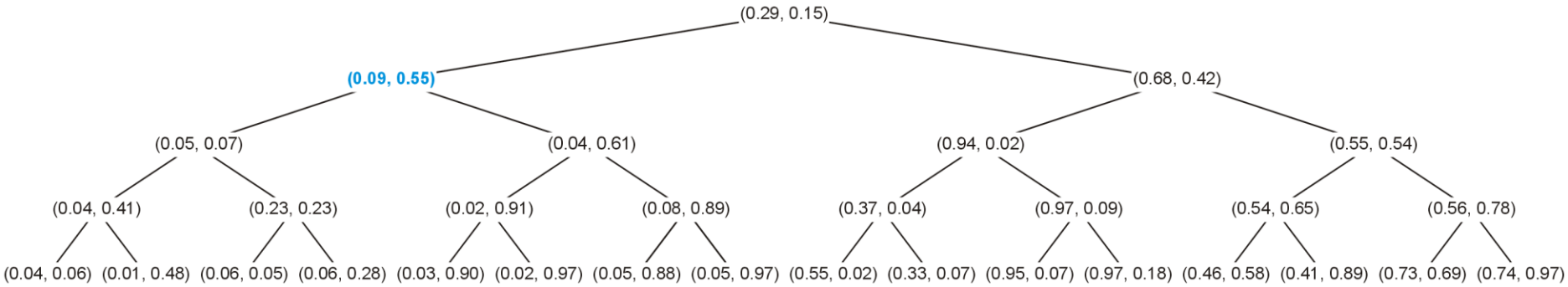


k-d trees

Next, we compare the second coordinate:

0.66 > 0.55

Thus we add the new element to the right sub-tree

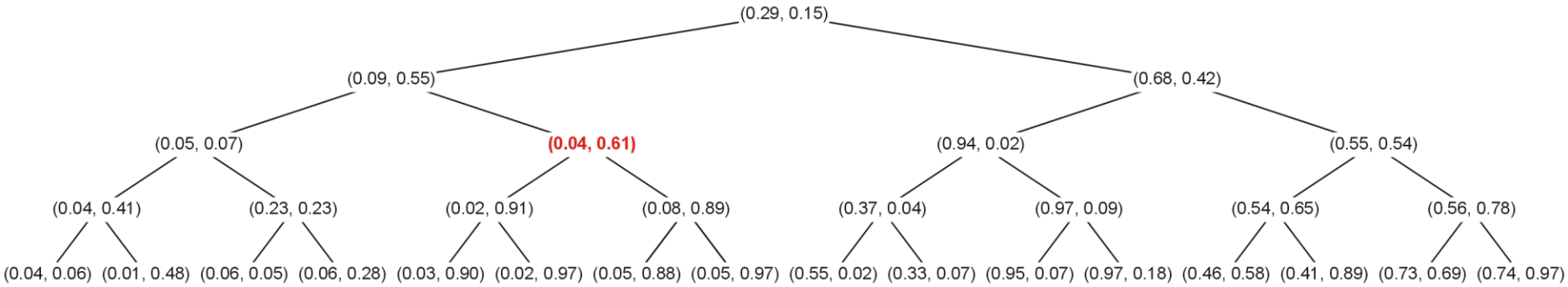


k-d trees

Next, we compare the first coordinate again:

0.15 > 0.04

Thus we add the new element to the right sub-tree

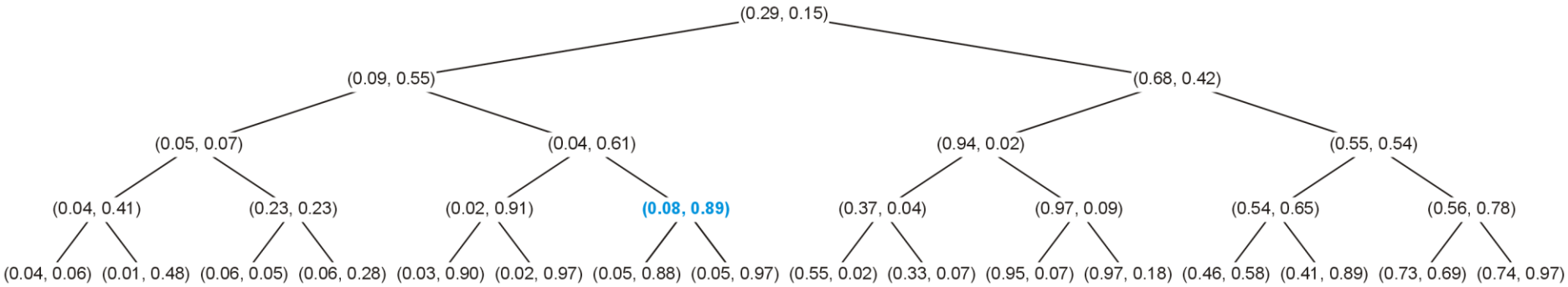


k-d trees

Next, we compare the first coordinate:

$$0.66 < 0.89$$

Thus we add the new element to the left sub-tree

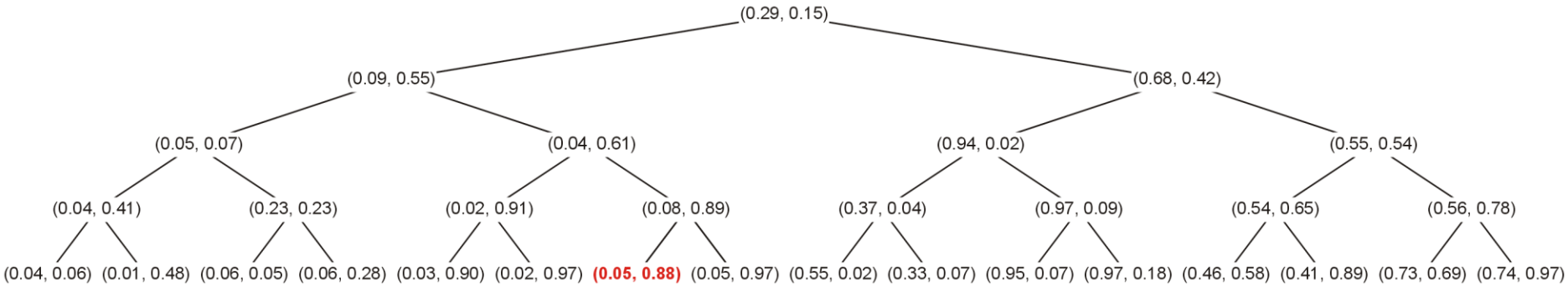


k-d trees

Finally, we compare the first coordinate again:

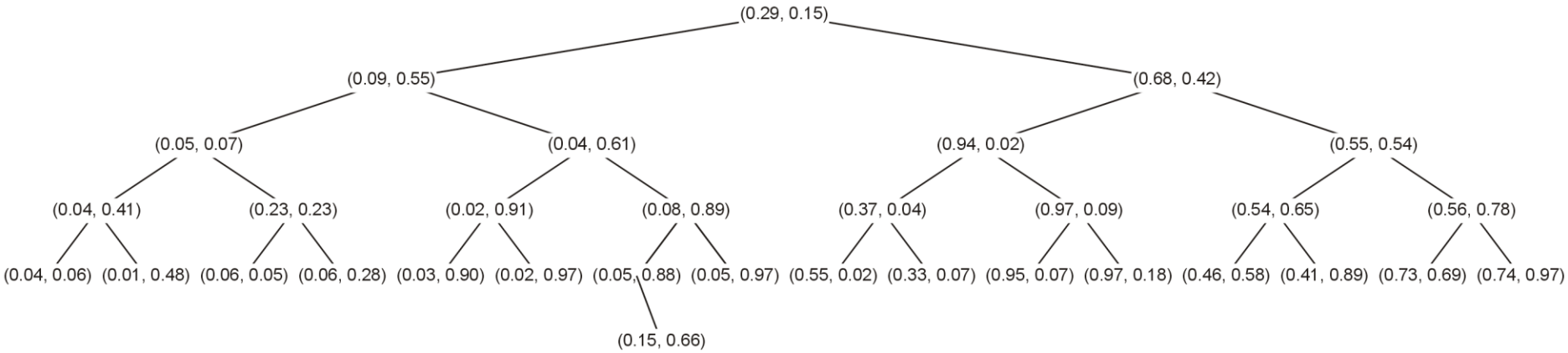
$$0.15 > 0.05$$

Thus we add the new element as a new leaf node to the right



k-d trees

Thus, we have successfully added the point (0.15, 0.66) to the existing k-d tree



k-d trees

It is also a reasonably fast algorithm to find a point's *nearest neighbour*

The nearest neighbour is a point which is closest to, but not equal to the given node

- first we search as if we are performing an insertion, but
- we may have to back up the tree until we ensure that neighbouring regions may not have a point closer

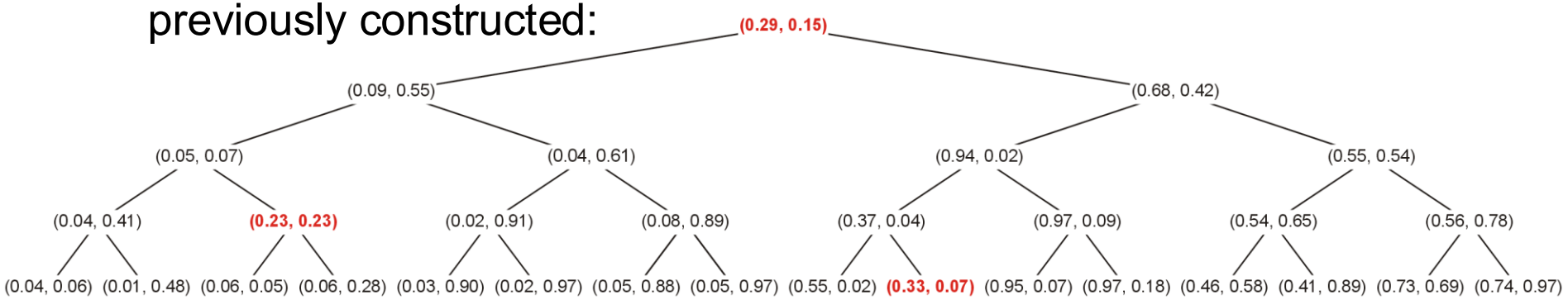
k-d trees

Removing elements is more difficult, as we must maintain the structure

Finding the “largest element” with respect to a particular coordinate may be very difficult

k-d trees

For example, consider removing the root node from the k-d tree we previously constructed:



We note:

- the minimum element in the right sub-tree is not in the obvious location, and
- neither is the maximum element in the left sub-tree...

k-d trees

It may be easier to flag removed nodes as being “deleted” and leave the tree unchanged otherwise

With numerous removals, this may significantly increase the run time of other operations...

k-d trees

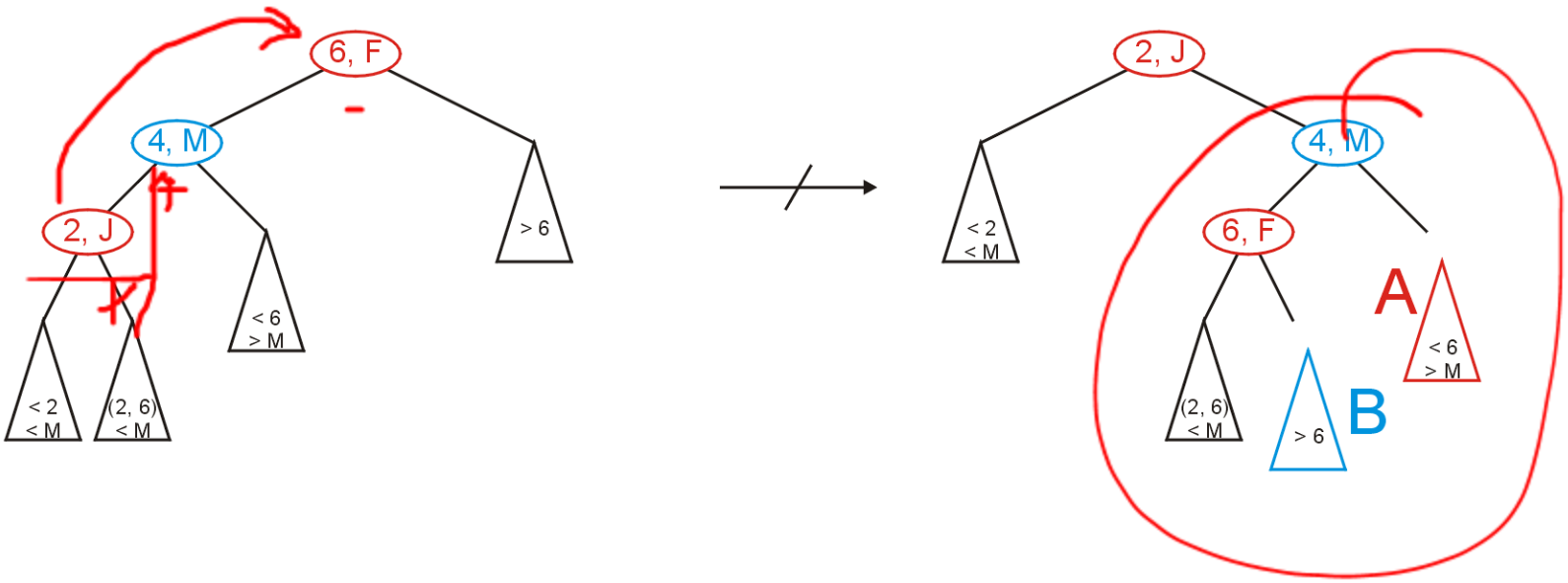
Balancing k-d trees is much more difficult than balancing AVL trees

You cannot perform rotations in the same way in which rotations are performed with AVL trees

k-d trees

Consider the following naïve rotation

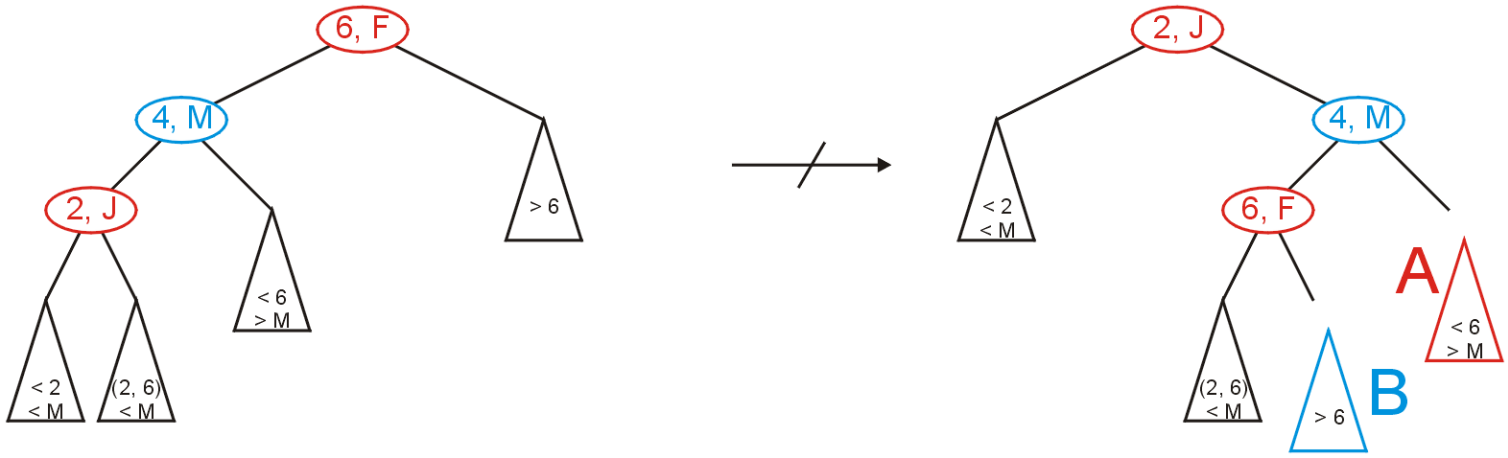
We will rotate (2, J) to the root and we reattach nodes (4, M) and (6, F) as appropriate, as well as two sub-trees



k-d trees

Unfortunately, tree A may have:

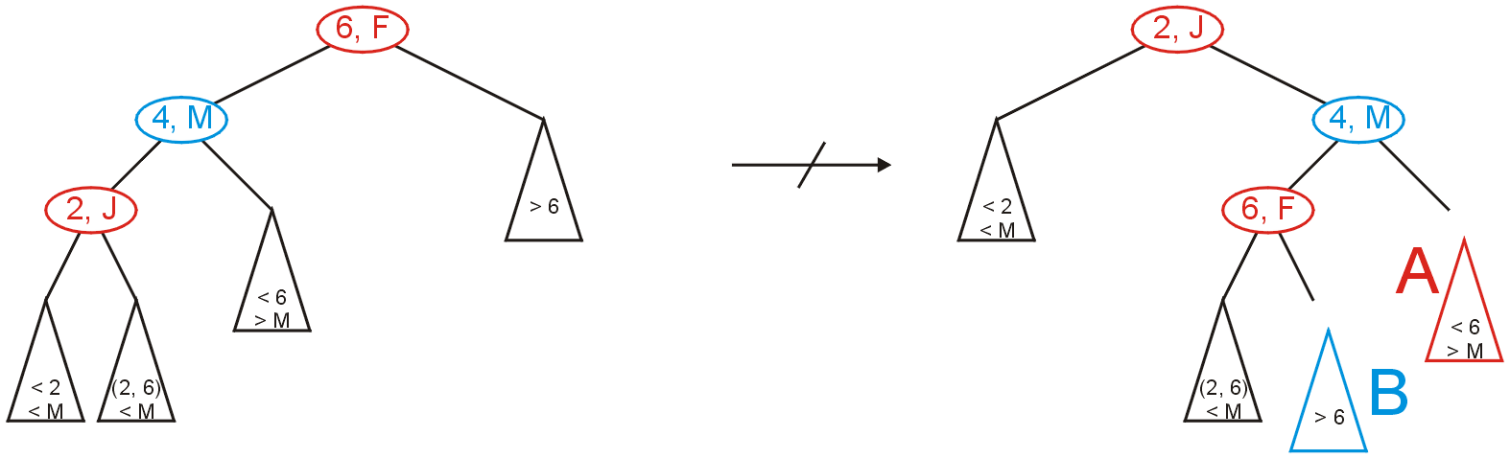
- elements less than 2, and
- if non-empty, all elements are greater than M



k-d trees

Similarly, tree B may have:

- has all elements greater than 6 (good), but
- there is no restriction which guarantees that the elements are all < M or all > M



k-d trees

The use of k-d trees is not only restricted to 2d, 3d, or higher dimensional spaces

Consider the following problem:

- suppose we have a database of phone numbers of individuals within a particular province
- find all individuals 20 km from some point who are between the ages of 19 and 32
- we could design a 3-dimensional k-d tree based on the three coordinates (x, y, age)

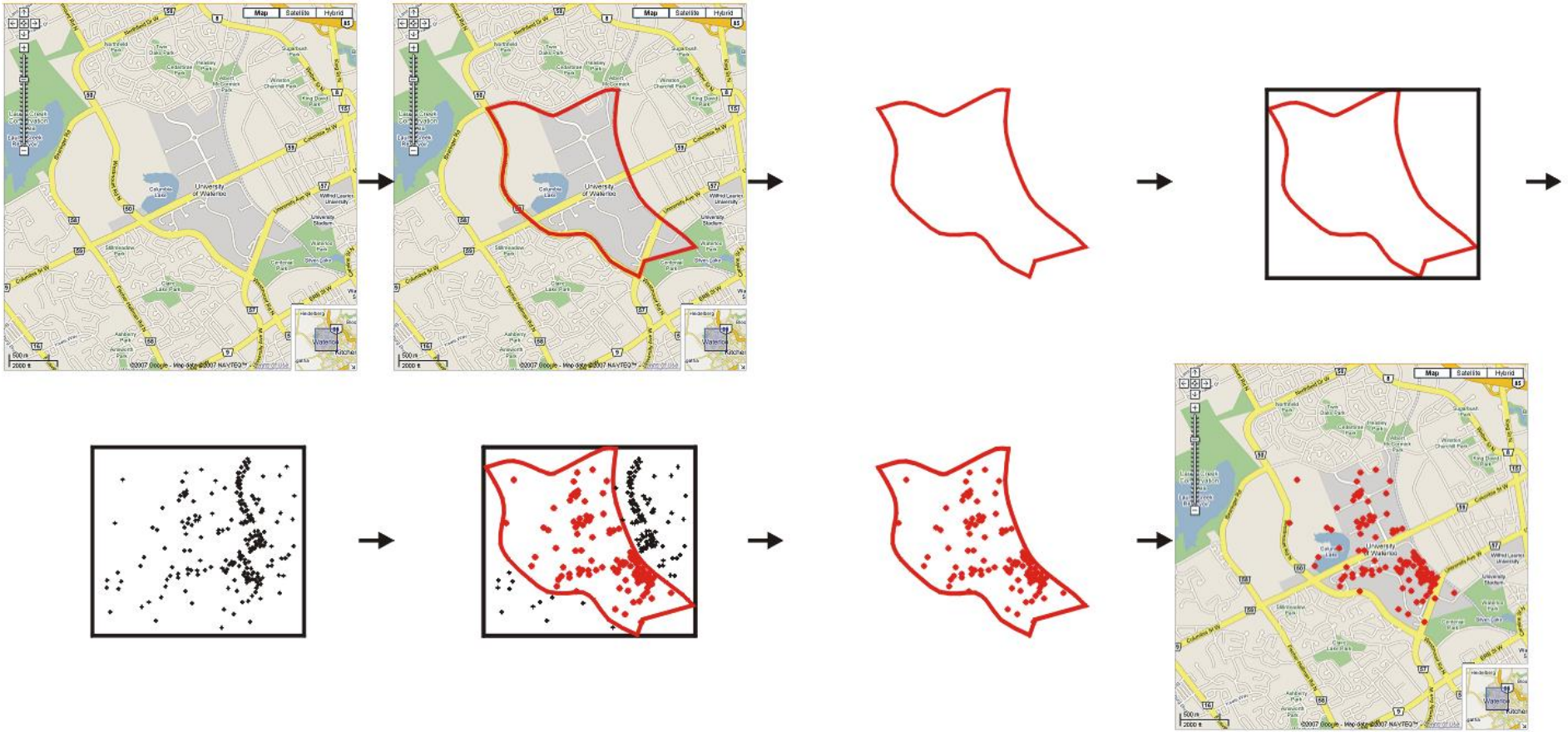
k-d trees

The previous example introduces an interesting problem: 20 km from some point is a circular region

- k-d trees can only detect boxed regions
- thus, we find the smallest bounding box of the region, find all points in the box, and select those within the region

k-d trees

Suppose we wish to find all Blackberry users within UW east of Westmount Dr.



k-d trees

In the example, we started with a given set and by using the median, we arrived at a balanced tree

If we are adding or removing points, we may require rotations to keep the tree balanced

These rotations are more complex than rotations for, say, an AVL tree

k-d trees

In this topic, we have covered red-black trees

- Simple rules govern how nodes must be distributed based on giving each node a colour of either red or black
- Insertions and deletions may be performed without recursing back to the root
- Only one bit is required for the “colour”
- This makes them, under some circumstances, more suited than AVL trees

References

[1] Weiss, Data Structures and Algorithm Analysis in C++, 3rd Ed., Addison Wesley, §12.6, p.549-53.