CMSC 420: Extended KD Trees and Queries

Justin Wyss-Gallifent

April 18, 2023

1 Introduction

Next on our agenda is to dig a bit deeper into extended kd-trees and to look at nearest neighbor queries.

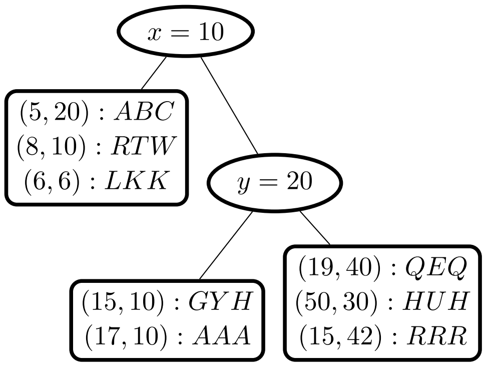
2 Extended KD-Trees

2.1 The Basics

The classic approach to modifying kd-trees is to use the leaves (the external nodes) to store all the data and the internal nodes to simply be guideposts to the data. We typically allow up to some maximum m points to be stored as a list in each leaf and each point may have some data associated to it.

In this sense each internal node would contain two pieces of information, the coordinate being split on and the splitting coordinate value, thus we'll just call them *splitting nodes* and each leaf contains a list of up to m points with associated data.

Example 2.1. For example a simple extended kd-tree with $k = 2$ and $m = 3$ might look like this:



Two important notes:

Note 2.1.1. We have not specified just yet exactly how the splitting nodes are created. The actual answer is that the splitting nodes arise as part of the insertion process. We will visit this next.

Note 2.1.2. Structurally speaking coordinates which are equal to splitting coordinates could go either left or right. This is done in order to keep the tree as balanced as possible. It is not a problem but we will comment where necessary how this might affect our various extended kd-tree operations.

2.2 Insertion and Splitting

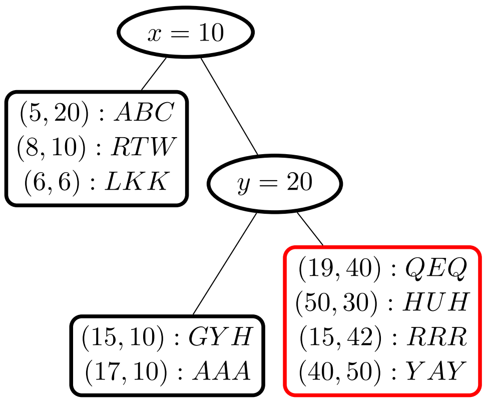
Suppose we wish to insert point into the tree. For each point we direct it to a leaf via the splitting nodes.

Note 2.2.1. By convention when we follow the splitting nodes for insertion we always go right on an equal coordinate. Another choice might be to go left or to randomly go left or right but we'll stick to right for consistency.

Once we reach the leaf node we first simply insert the point, then we ask if the node is overfull or not. If the leaf node is not overfull then there is nothing to do.

If the leaf node is overfull then we must split it.

Example 2.2. Here is a simple example. Suppose we insert the point $(40, 50) : YAY$ into our tree from earlier. They both end up in one leaf which is overfull since $m = 3$:



We must split this overfull node.

When we split an overfull we create a parent splitting node. This parent splitting node needs to have a splitting coordinate and a splitting value. There are two classic ways to approach this:

1. *Cycle Split*: We look at the leaf's parent and if it splits by coordinate α then we split by coordinate $\alpha + 1 \bmod k$. If there is no parent (the leaf is the root) then we split by the first coordinate. For example if $k = 3$ (three-dimensional) then if the parent splits by x then our new split is by y, if the parent splits by y then our new split is by z, and if the parent splits by z then our new split is by x. If there is no parent we split by x. This is basically how we traditionally work with kd-trees.
2. *Spread Split*: We pick the coordinate such that the point spread in that coordinate's direction is largest. For example if $k = 3$ and if the point spread in the x-direction is larger than either the y- or z-direction then we split by x. In the case of ties we pick the earliest coordinate with

the largest coordinate spread, so for example in the $k = 3$ case if the x-spread is 3, the y-spread is 4, and the z-spread is 4 we would choose the y-coordinate to split on.

Once we have chosen the splitting coordinate we sort the points in the leaf according to that coordinate with ties broken by the remaining coordinates in cycling order. We'll call this a *coordinate sort*.

Example 2.3. For example suppose $k = 3$ and we have the set of points which needs to be split:

$$(1, 2, 3), (3, 2, 1), (2, 2, 1), (2, 1, 2), (2, 1, 3), (3, 3, 3)$$

Suppose we are splitting by the y-coordinate. We break ties by the z-coordinate and further ties by the x-coordinate. One way to imagine this working is to "rotate" each point so it has the order (y, z, x), sort them in the obvious way, then "unrotate" back.

In the above example if we rotate we get:

$$(y, z, x) = (2, 3, 1), (2, 1, 3), (2, 1, 2), (1, 2, 2), (1, 3, 2), (3, 3, 3)$$

Sort:

$$(y, z, x) = (1, 2, 2), (1, 3, 2), (2, 1, 2), (2, 1, 3), (2, 3, 1), (3, 3, 3)$$

Unrotate:

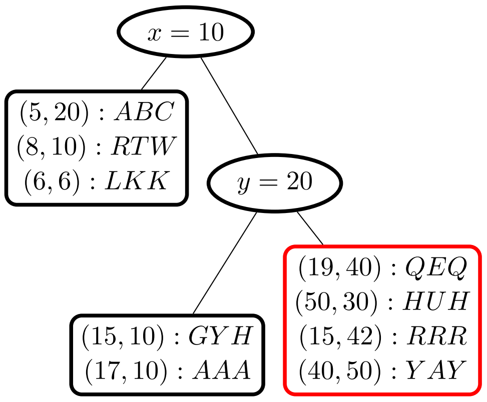
$$(x, y, z) = (2, 1, 2), (2, 1, 3), (2, 2, 1), (3, 2, 1), (1, 2, 3), (3, 3, 3)$$

Once we have sorted the n points in the leaf we split them into the first $\lfloor n/2 \rfloor$ and the rest.

Note 2.2.2. Note that when we split the points we could get points with the same splitting coordinate in both the left and right leaves. This will be relevant when we search.

Finally the splitting node value is assigned to be exactly the median.

Example 2.4. Returning to our example with an overfull node:

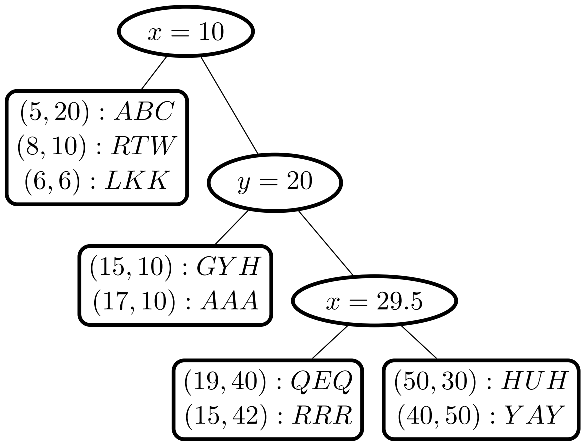


Suppose we are using a cycle split. Since the parent node splits by y our new split will be by x.

If we coordinate sort the points by x they are in order:

$$(15,42), (19,40), (40,50), (50,30)$$

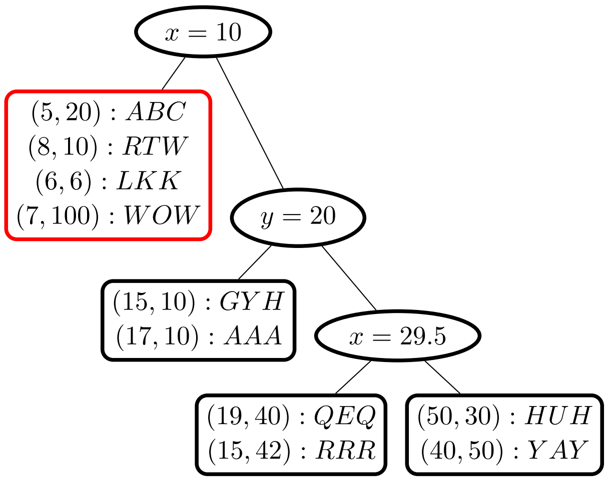
The median x-value is 29.5 and so the new splitting node will split by that x-coordinate. The first $\lfloor 4/2 \rfloor = 2$ points go left, the other 2 go right:



Note that the previous example would not change if we were using a spread split because the spread in the x-direction is 35, which is larger than the spread in the y-direction which is 20.

Here is an example with a spread split:

Example 2.5. Suppose we took the ending result from the above example and inserted $(7,100) : WOW$. The result is:

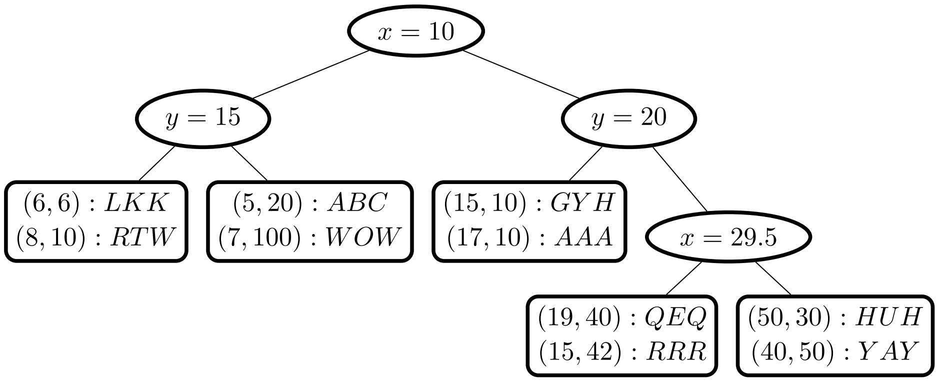


If we examine the points in the overfull node we see that the spread in the x-direction is 3 whereas the spread in the y-direction is 94. Since the spread in the y-direction is larger we split using the y-coordinate.

We coordinate sort the points by y-coordinate:

$$(6, 6), (8, 10), (5, 20), (7, 100)$$

The median y-value is 15 and so the new splitting node will split by that y-coordinate. The first $\lfloor 4/2 \rfloor = 2$ points go left, the other 2 go right:

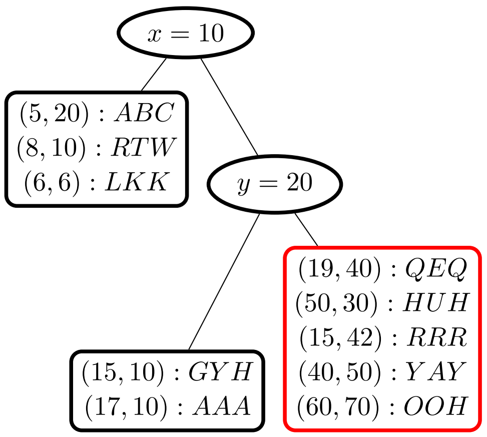


2.3 Bulk Insertion and Splitting

It's not uncommon to choose to bulk-insert points into an extended kd-tree. In such a situation we first insert all the points without regard for whether the leaf nodes are overfull or not. Once the insertion is done we then go back and check all the leaf nodes which gained points. We split them if necessary and, if necessary, do this recursively on the results.

Here is a simple example.

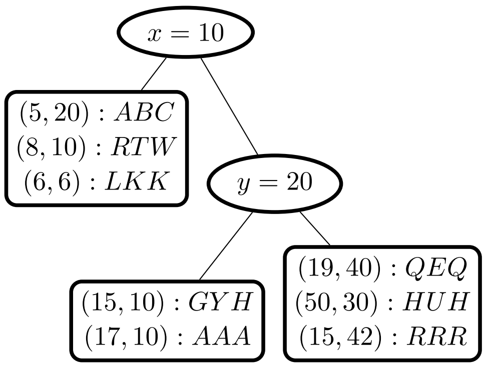
Example 2.6. Suppose we insert the two points $(40, 50) : YAY$ and $(60, 70) : OOH$ into our tree from earlier. They both end up in one leaf which is overfull since $m = 3$:



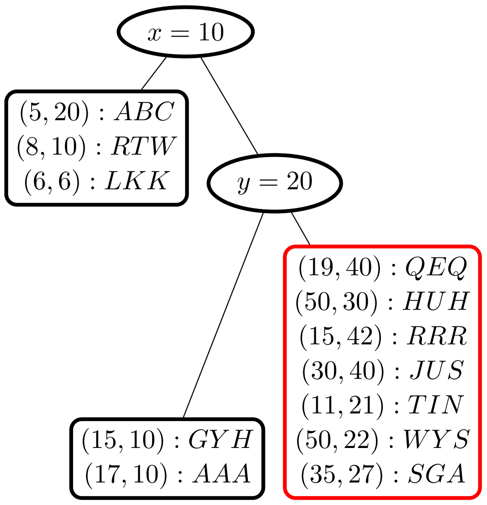
We must split this overfull node.

Here is an example where we need to do it recursively:

Example 2.7. Let's return to our example from the start:



Let's insert four new points into the tree, $(30, 40) : JUS$, $(11, 21) : TIN$, $(50, 22) : WYS$, and $(35, 27) : SGA$. Now we have a really overfull node:

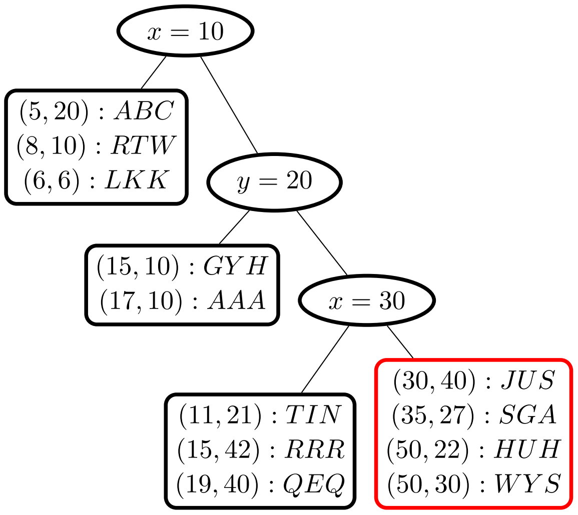


Suppose we are using cycle split. Since the parent node splits by y we must split by x.

We coordinate sort the points by the x-coordinate:

$$(11,21), (15,42), (19,40), (30,40), (35,27), (50,22), (50,30)$$

The median of the x-coordinates is 30 so our splitting node has a value of 30. The first $\lfloor 7/2 \rfloor = 3$ points go left and the remaining 4 go right. Notice that the new right leaf is still overfull:

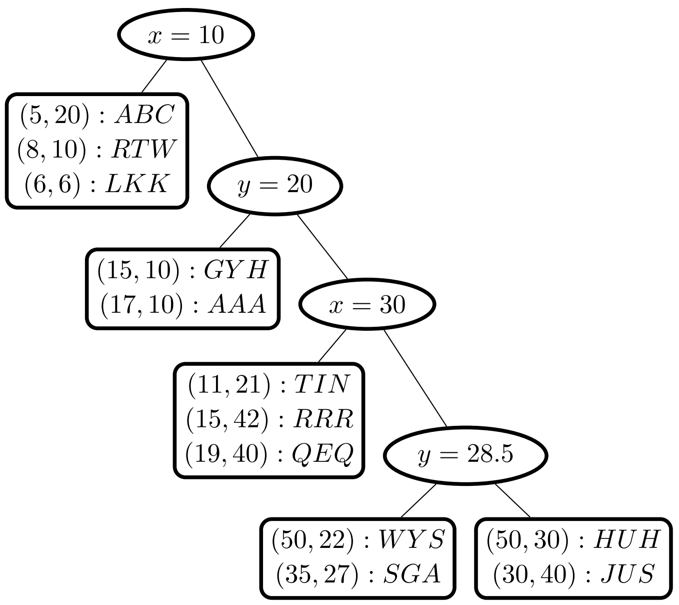


We must then split again, this time by the y-coordinate.

We coordinate sort the points by the y-coordinate:

$$(50,22),(35,27),(50,30),(30,40)$$

The median of the y-coordinates is 28.5 so our splitting node has a value of 28.5. The points are split half and half:



Now we are done.

2.4 Searching

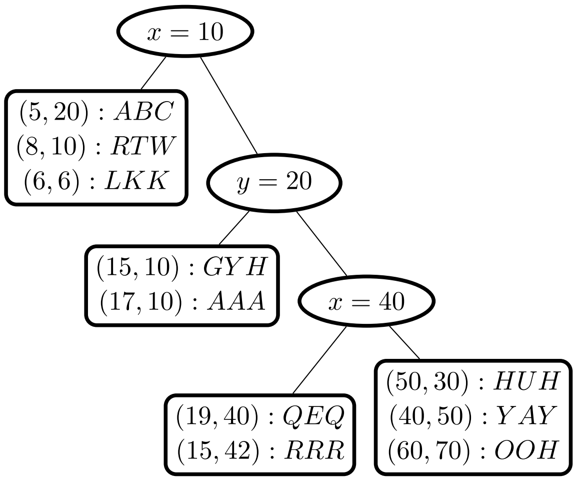
Searching can be a bit awkward because we are not insisting that a point with coordinate equal to a splitting coordinate value must go right. As a reminder we do this because it guarantees a bit better balance.

As a consequence if we are searching and we arrive at a splitting node and our search point has a coordinate equal to the splitting coordinate value we must check both left and right.

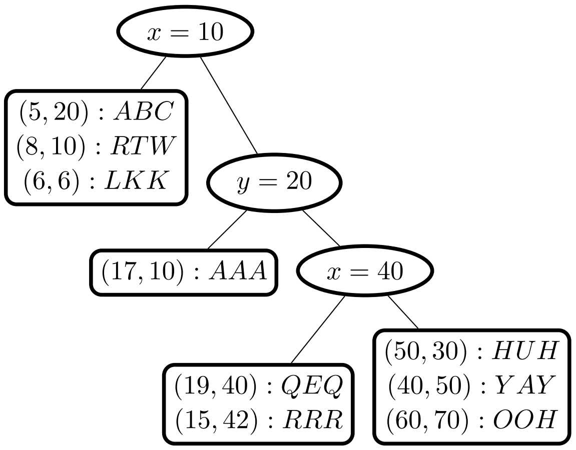
2.5 Deletion

Deletion is particularly easy in an extended kd-tree. We simply delete the point from its leaf. the only issue that might arise is if the leaf becomes empty. In such a case the splitting node pointing to it becomes redundant. What we do in such a case is that we remove the splitting node and connect the sibling of the empty leaf directly to the parent of the splitting node.

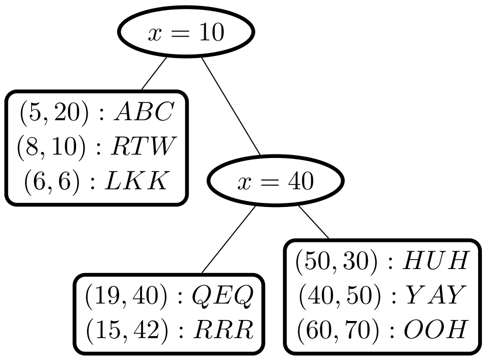
Example 2.8. Returning to the above example:



If we delete just $(15, 10) : GYH$ then we have no issues:



If we also delete $(17, 10) : AAA$ then the splitting node $y = 20$ becomes irrelevant and we remove and splice:



3 Nearest Neighbor Query

3.1 Introduction

The idea behind a nearest neighbor query is that we have a target point P which may or may not be in the tree and we wish to find the n points in the tree which are closest to P.

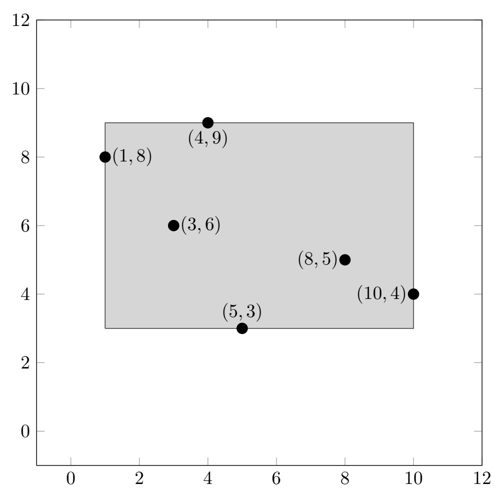
To avoid issues of square roots (and floating point calculations) we'll calculate and use the squares of distances rather than distances. In this discussion we'll refer to the $d^2 = ...$ to mean the square of the distance to our target point P.

We could do this really crudely, of course, simply by visiting every leaf, checking every point, calculating every d^2 value, and picking out the n closest. However we'd like to do this more efficiently.

3.2 Bounding Boxes

To make our algorithm more efficient each node has an associate bounding box. Intutively this bounding box is a k-dimensional box which is as small as possible and still contains all the points in the subtree rooted at that node.

Example 3.1. Suppose a subtree contains the points $(1, 8)$, $(5, 3)$, $(4, 9)$, $(8, 5)$, $(3, 6)$, and $(10, 4)$. The mininum and maximum x-values are 1 and 10 respectively and the minimum and maximum y-values are 3 and 9 respectively. The corresponding bounding box would be the smallest box which contains all five points, $[1, 10] \times [3, 9]$, as shown here.



There are various ways to work with bounding boxes. One typical way is to just use the minimum and maximum for each coordinate. For the above we would store $[xmin, xmax] = [1, 10]$ and $[ymin, ymax] = [3, 9]$. This generalizes conveniently to higher k values.

Note 3.2.1. Whether we store the bounding box information in a node or whether we compute it on the fly is a matter of programmatical choice. Computing it on the fly increases the time complexity of the query whereas storing it requires that we update it with each point insertion or deletion.

Given a point, calculating the d^2 value from a point P to a bounding box is easy. Let's look at the 2-d version. Suppose we have $[xmin, xmax]$ and $[ymin, ymax]$ stored and $P = (x, y)$. We calculate:

- The distance from x to the interval $[xmin, xmax]$.
- The distance from y to the interval $[ymin, ymax]$
- Take the sum of the squares of these distances.

Note that if P is inside the bounding box then this will return a value of 0, which makes sense.

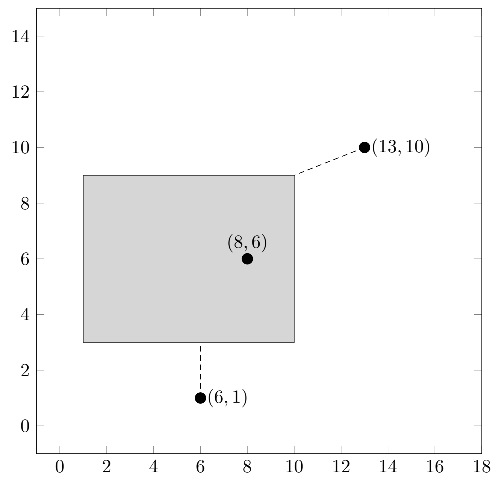
If we have more than $k = 2$ dimensions we just calculate more distances and sum the squares. If i_{min} and i_{max} are the minimum and maximum coordinates for the bounding box in k dimensions and if P_i is the i^{th} coordinate of the point P then we would have:

$$d^2 \text{ value} = \sum_{i=1}^{k} \text{dist}(P_i, [i_{min}, i_{max}])^2$$

Note 3.2.2. The distance from a coordinate c to an interval $[cmin, cmax]$ is easy: If $c < cmin$ it's $cmin - c$, if $c > cmax$ it's $c - cmax$, otherwise it's 0.

Here is an example to illustrate that this method works:

Example 3.2. Consider the bounding box from earlier and consider the three points $P_1 = (13, 10)$, $P_2 = (6, 1)$, and $P_3 = (8, 6)$. The d^2 values are shown as dashed, except for $(8, 6)$ whith $d^2 = 0$ since it's inside the bounding box.



We have $[xmin, xmax] = [1, 10]$ and $[ymin, ymax] = [3, 9]$.

For these points we have:

1. $P_1 = (13, 10)$:
   - Distance from 13 to $[1, 10]$ is 3.
   - Distance from 10 to $[3, 9]$ is 1.
   - The result is then $3^2 + 1^2 = 10$.
2. $P_2 = (6, 1)$:
   - Distance from 6 to $[1, 10]$ is 0.
   - Distance from 1 to $[3, 9]$ is 2.
   - The result is then $0^2 + 2^2 = 4$.
3. $P_3 = (8, 6)$:
   - Distance from 8 to $[1, 10]$ is 0.
   - Distance from 6 to $[3, 9]$ is 0.
   - The result is then $0^2 + 0^2 = 0$.

3.3 Algorithm Commentary

Broadly speaking we'll keep track of a working list of up to n points (which will begin empty) and only update that list if we find closer points than the points in our list or if the list is not full. We can then use this list and our bounding boxes to do a more efficient job of searching.

To clarify this efficiency, suppose we are looking for some points closest to P and we arrive at a splitting node. Consider these two possibilities:

- If the bounding box for one subtree is closer to P than the bounding box for the the other subtree then the points we are looking for are more likely to be in that subtree and so we should visit it first.
- Suppose we have our working list and it's full, containing n points, and the furthest point in that list has d^2 value equal to, say, 10. If the bounding box for one of the subtrees is more than $d^2 = 10$ away from P then nothing in that entire subtree will be closer than the points in our working list and there's no reason to visit that subtree.

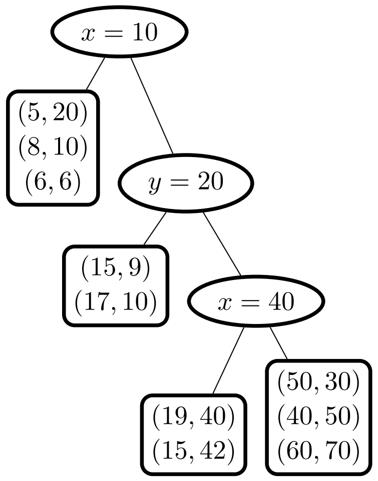
3.4 Algorithm

This suggests the following recursive algorithm when searching for the n points closest to a target P. Starting with an empty list of length n and starting at the root:

1. If we are at a leaf, check all the points to see if their d^2 values are smaller than any of the points in our list and if they are, replace the further points in our list with closer ones.
2. If we are at a splitting node and if our list is full and if the bounding box for a subtree is further away than the furthest point in our list, ignore that subtree. Note that this could mean ignoring both subtrees!
3. If we are at a splitting node and there are two subtrees to visit, visit the subtree whose bounding box is closest to P first.

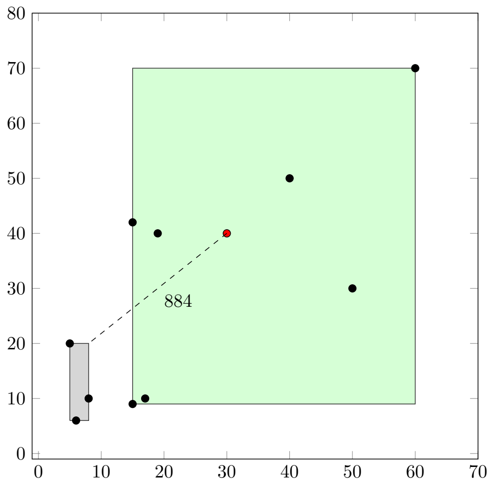
Note 3.4.1. There are various conventions for breaking ties, for example if we have to choose between two points with the same d^2 value. These conventions are typically application-specific.

Example 3.3. Consider this extended kd-tree. Here I've suppressed any letter-data



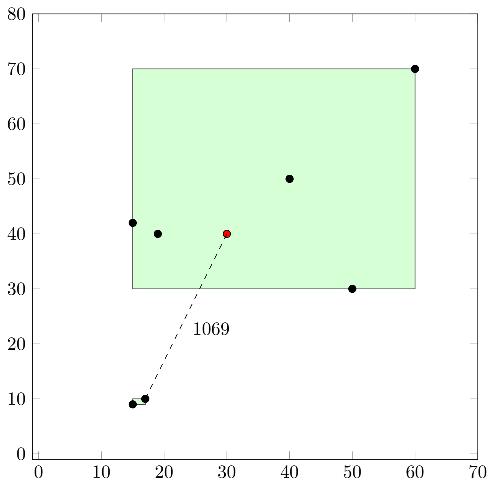
Suppose we want the $n = 2$ closest points to $(30, 40)$. We begin with an empty list $L = [,]$ of size $n = 2$.

We start at the root $x = 10$. The bounding box for the left subtree has $[xmin, xmax] = [5, 8]$ and $[ymin, ymax] = [6, 20]$ and the bounding box for the right subtree has $[xmin, xmax] = [15, 60]$ and $[ymin, ymax] = [9, 70]$. The point $(30, 40)$ is closer to the right bounding box (it's inside that bounding box!) so we explore right first.

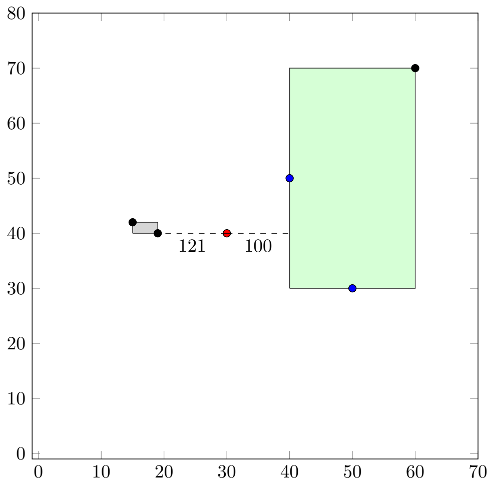


We are at $y = 20$. The bounding box for the left subtree has $[xmin, xmax] = [15, 17]$ and $[ymin, ymax] = [9, 10]$ and the bounding box for the right subtree has $[xmin, xmax] = [15, 60]$ and $[ymin, ymax] = [30, 70]$. The point $(30, 40)$ is closer to the right bounding box (it's inside that bounding box!)

so we explore right first.

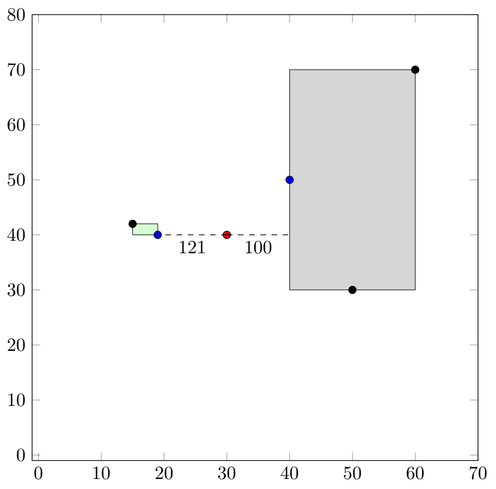


We are at $x = 40$. The bounding box for the left subtree has $[xmin, xmax] = [15, 19]$ and $[ymin, ymax] = [40, 42]$ and the bounding box for the right subtree has $[xmin, xmax] = [40, 60]$ and $[ymin, ymax] = [30, 70]$. The point $(30, 40)$ is closer to the right bounding box $d^2 = 100$ vs $d^2 = 121$ (check this!) so we explore right first.



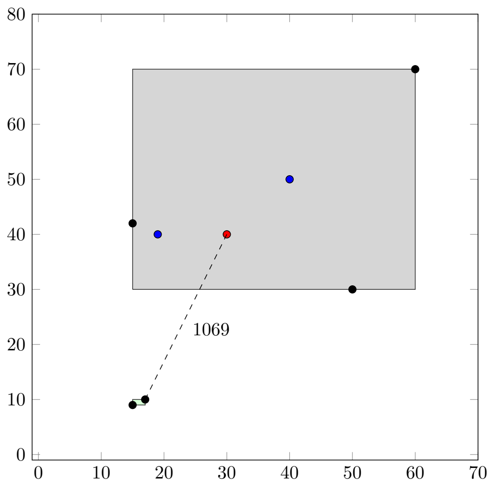
In the right subtree we have $(50, 30)$, $(40, 50)$, and $(60, 70)$. Since our list is empty we put the two points closest to $(30, 40)$ in it. Now we have $L = [(40, 50), (50, 30)]$. Observe that these have $d^2 = 200$ ane $d^2 = 500$ respectively. These are marked blue above.

We back up and look at the left subtree of $x = 40$:



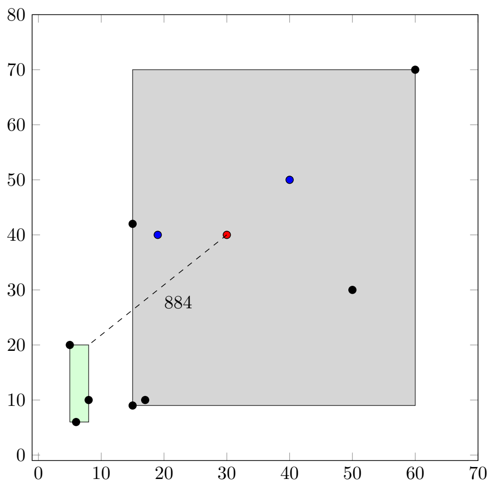
The bounding box for the left subtree has $[xmin, xmax] = [15, 19]$ and $[ymin, ymax] = [40, 42]$ and this has $d^2 = 121$ which is less than both 200 and 500 so it's worth checking. We have $(19, 40)$ and $(15, 42)$ with d^2 values of 121 and 229. The first one is better than 500 so we throw out $(50, 30)$ and put $(19, 40)$ in our list, so now $L = [(40, 50), (19, 40)]$ with $d^2 = 200$ and $d^2 = 121$ respectively. These are marked blue above.

We back up and look at the left subtree of $y = 20$:



The bounding box for the left subtree has $[xmin, xmax] = [15, 17]$ and $[ymin, ymax] = [9, 10]$ and this has $d^2 = 1069 > 200$ so it's not even worth looking there.

We back up and look at the left subtree of $x = 10$:



The bounding box for the left subtree has $[xmin, xmax] = [5, 8]$ and $[ymin, ymax] = [6, 20]$ and this has $d^2 = 884 > 200$ so it's not even worth looking there.

Thus we end with $L = [(40, 50), (19, 40)]$ and those are our two closest points.