

WIKIPEDIA

k-d tree

In computer science, a ***k-d tree*** (short for *k-dimensional tree*) is a space-partitioning data structure for organizing points in a *k*-dimensional space. *k*-d trees are a useful data structure for several applications, such as searches involving a multidimensional search key (e.g. range searches and nearest neighbor searches) and creating point clouds. *k*-d trees are a special case of binary space partitioning trees.

Contents

Description

Operations on *k*-d trees

Construction

Example implementation

Adding elements

Removing elements

Balancing

Nearest neighbour search

Range search

Degradation in performance with high-dimensional data

Degradation in performance when the query point is far from points in the *k*-d tree

Complexity

Variations

Volumetric objects

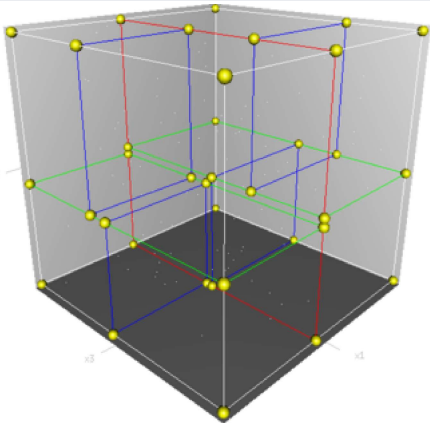
Points only in leaves

See also

Open source implementations

References

k-d tree



A 3-dimensional *k*-d tree. The first split (the red vertical plane) cuts the root cell (white) into two subcells, each of which is then split (by the green horizontal planes) into two subcells. Finally, four cells are split (by the four blue vertical planes) into two subcells. Since there is no more splitting, the final eight are called leaf cells.

Type	Multidimensional	BST
Invented	1975	
Invented by	Jon Louis Bentley	
Time complexity in big O notation		
Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$

Description

The *k*-d tree is a binary tree in which *every* node is a *k*-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces. Points to the left of this hyperplane are represented by the left subtree of that node and points to the right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the *k* dimensions, with the hyperplane perpendicular to that dimension's axis. So, for example, if for a particular split the "x"

axis is chosen, all points in the subtree with a smaller "x" value than the node will appear in the left subtree and all points with a larger "x" value will be in the right subtree. In such a case, the hyperplane would be set by the x value of the point, and its normal would be the unit x-axis.^[1]

Operations on *k*-d trees

Construction

Since there are many possible ways to choose axis-aligned splitting planes, there are many different ways to construct *k*-d trees. The canonical method of *k*-d tree construction has the following constraints:^[2]

- As one moves down the tree, one cycles through the axes used to select the splitting planes. (For example, in a 3-dimensional tree, the root would have an x-aligned plane, the root's children would both have y-aligned planes, the root's grandchildren would all have z-aligned planes, the root's great-grandchildren would all have x-aligned planes, the root's great-great-grandchildren would all have y-aligned planes, and so on.)
- Points are inserted by selecting the median of the points being put into the subtree, with respect to their coordinates in the axis being used to create the splitting plane. (Note the assumption that we feed the entire set of n points into the algorithm up-front.)

This method leads to a balanced *k*-d tree, in which each leaf node is approximately the same distance from the root. However, balanced trees are not necessarily optimal for all applications.

Note that it is not *required* to select the median point. In the case where median points are not selected, there is no guarantee that the tree will be balanced. To avoid coding a complex $O(n)$ median-finding algorithm^{[3][4]} or using an $O(n \log n)$ sort such as heapsort or mergesort to sort all n points, a popular practice is to sort a *fixed* number of *randomly selected* points, and use the median of those points to serve as the splitting plane. In practice, this technique often results in nicely balanced trees.

Given a list of n points, the following algorithm uses a median-finding sort to construct a balanced *k*-d tree containing those points.

```
function kdtree (list of points pointList, int depth)
{
    // Select axis based on depth so that axis cycles through all valid values
    var int axis := depth mod k;

    // Sort point list and choose median as pivot element
    select median by axis from pointList;

    // Create node and construct subtree
    node.location := median;
    node.leftChild := kdtree(points in pointList before median, depth+1);
    node.rightChild := kdtree(points in pointList after median, depth+1);
    return node;
}
```

It is common that points "after" the median include only the ones that are strictly greater than the median. For points that lie on the median, it is possible to define a "superkey" function that compares the points in all dimensions. In some cases, it is acceptable to let points equal to the median lie on one side of the median, for example, by splitting the points into a "lesser than" subset and a "greater than or equal to" subset.

This algorithm creates the invariant that for any node, all the nodes in the left subtree are on one side of a splitting plane, and all the nodes in the right subtree are on the other side. Points that lie on the splitting plane may appear on either side. The splitting plane of a node goes through the point associated with that node (referred to in the code as *node.location*).

Alternative algorithms for building a balanced k -d tree presort the data prior to building the tree. Then, they maintain the order of the presort during tree construction and hence eliminate the costly step of finding the median at each level of subdivision. Two such algorithms build a balanced k -d tree to sort triangles in order to improve the execution time of ray tracing for three-dimensional computer graphics. These algorithms presort n triangles prior to building the k -d tree, then build the tree in $O(n \log n)$ time in the best case.^{[5][6]} An algorithm that builds a balanced k -d tree to sort points has a worst-case complexity of $O(kn \log n)$.^{[7][8]} This algorithm presorts n points in each of k dimensions using an $O(n \log n)$ sort such as Heapsort or Mergesort prior to building the tree. It then maintains the order of these k presorts during tree construction and thereby avoids finding the median at each level of subdivision.

Example implementation

The above algorithm implemented in the Python programming language is as follows:

```
from collections import namedtuple
from operator import itemgetter
from pprint import pformat

class Node(namedtuple("Node", "location left_child right_child")):
    def __repr__(self):
        return pformat(tuple(self))

def kdtree(point_list, depth: int = 0):
    if not point_list:
        return None

    k = len(point_list[0]) # assumes all points have the same dimension
    # Select axis based on depth so that axis cycles through all valid values
    axis = depth % k

    # Sort point list by axis and choose median as pivot element
    point_list.sort(key=itemgetter(axis))
    median = len(point_list) // 2

    # Create node and construct subtrees
    return Node(
        location=point_list[median],
        left_child=kdtree(point_list[:median], depth + 1),
        right_child=kdtree(point_list[median + 1:], depth + 1),
    )

def main():
    """Example usage"""
    point_list = [(7, 2), (5, 4), (9, 6), (4, 7), (8, 1), (2, 3)]
    tree = kdtree(point_list)
    print(tree)

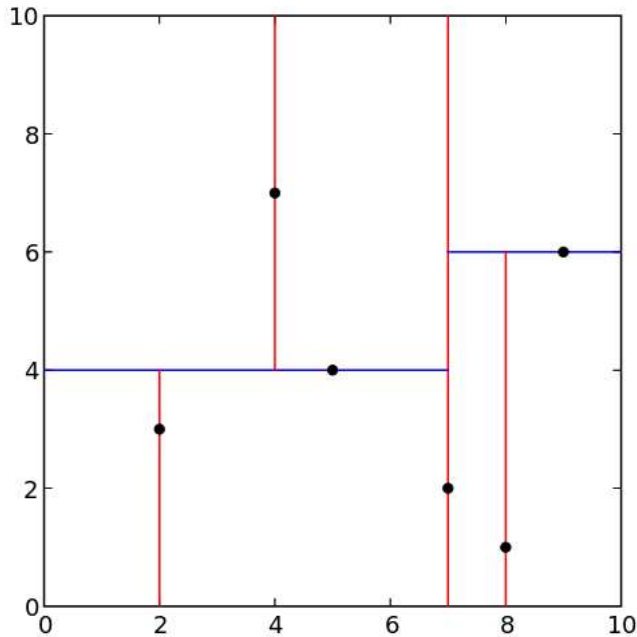
if __name__ == "__main__":
    main()
```

Output would be:

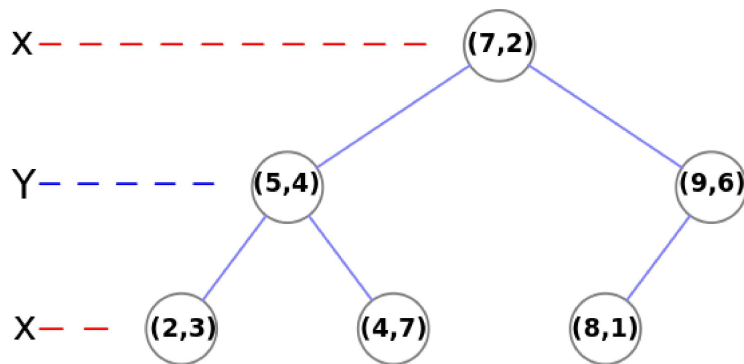
```
((7, 2),
 ((5, 4), ((2, 3), None, None), ((4, 7), None, None)),
```

((9, 6), ((8, 1), None, None), None))

The generated tree is shown below.



k-d tree decomposition for the point set
(2,3), (5,4), (9,6), (4,7), (8,1), (7,2).



The resulting *k*-d tree.

Adding elements

One adds a new point to a *k*-d tree in the same way as one adds an element to any other search tree. First, traverse the tree, starting from the root and moving to either the left or the right child depending on whether the point to be inserted is on the "left" or "right" side of the splitting plane. Once you get to the node under which the child should be located, add the new point as either the left or right child of the leaf node, again depending on which side of the node's splitting plane contains the new node.

Adding points in this manner can cause the tree to become unbalanced, leading to decreased tree performance. The rate of tree performance degradation is dependent upon the spatial distribution of tree points being added, and the number of points added in relation to the tree size. If a tree becomes