K-D tree

Author:

Contents

Question------------------------------------------2

Explain the existing code---------------------3

Implement and explain insert() and range()----------4

Analyze the time complexity of range query---------7

Visualize the time performance between k-d tree method and naive method--------8

Implement the nearest neighbor query---9

Q：

K-D tree

A k-d tree (short for k-dimensional tree) is a space-partitioning data structure for organizing points in a k-dimensional

space.

Please complete the code based on kd tree.py or KDTree.java. In this project, we only consider points in two

dimensions.

1. Explain the existing code (5 marks).

2. Implement and explain insert() and range() (English writing style will be judged, 22 marks).

3. Analyze the time complexity of range query (5 marks).

4. Visualize the time performance between k-d tree method and naive method (5 marks).

5. (Bonus) implement the nearest neighbor query (5 marks).

Note that if you are not able to explain your code in Q2, then you will be risking of losing all marks in Q2.

A：

1. Explain the existing code

The existing code appears to be a class for a k-d tree, with methods for inserting points and finding points within a given range. The class uses a binary tree structure, where each node represents a point in a two-dimensional space and has up to two children nodes. The points are organized in the tree based on their x and y coordinates, with the x coordinate being used as the primary sorting key and the y coordinate as the secondary key. This allows for efficient search and insertion of points, as the tree is balanced and the points are organized in a way that allows for quick search.

2. Implement and explain insert() and range()

To implement the insert() method, we can follow a similar approach to the one used in the existing code. First, we can check if the root node is empty. If it is, we can simply insert the point as the root. If the root is not empty, we can compare the x and y coordinates of the point to be inserted with the x and y coordinates of the root. Depending on which coordinate is smaller, we can either insert the point as the left child of the root or the right child. We can then repeat this process recursively until we reach an empty spot in the tree, at which point we can insert the point.

Insert():

def insert(self, point):

if not self.root:

self.root = Node(point)

else:

current = self.root

while current:

if point[0] < current.point[0]:

if not current.left:

current.left = Node(point)

break

current = current.left

elif point[0] > current.point[0] or (point[0] == current.point[0] and point[1] < current.point[1]):

if not current.right:

current.right = Node(point)

break

current = current.right

To implement the range() method, we can use a similar approach to search for points within a given range. We can start at the root of the tree and recursively search the left and right subtrees based on whether the point being searched for is within the range of the current node. If a point is within the range, we can add it to a list of points that will be returned. We can continue this process until we reach a leaf node, at which point we can return the list of points.

Range():

def range(self, low, high):

points = []

def search(node):

if not node:

return

if low[0] <= node.point[0] <= high[0] and low[1] <= node.point[1] <= high[1]:

points.append(node.point)

if low[0] <= node.point[0]:

search(node.left)

if high[0] >= node.point[0]:

search(node.right)

search(self.root)

return points

3. Analyze the time complexity of range query

The time complexity of the range() method will depend on the balance of the tree. In the worst case, the tree may be completely unbalanced, resulting in a time complexity of O(n) for a range query, where n is the number of points in the tree. In the best case, the tree may be perfectly balanced, resulting in a time complexity of O(log n) for a range query. This is because each level of the tree reduces the number of nodes that need to be searched by half, so the time complexity grows logarithmically with the number of points in the tree.

4. Visualize the time performance between k-d tree method and naive method

To visualize the time performance between the k-d tree method and a naive method, we can create a test dataset of points and measure the time it takes to perform a range query using both methods. For the naive method, we can simply iterate through all the points in the dataset and check if they are within the given range. For the k-d tree method, we can use the range() method that we implemented earlier. We can then plot the time it takes to perform the range query for both methods as the size of the dataset increases, and compare the performance.

5. Implement the nearest neighbor query

(Bonus) To implement the nearest neighbor query, we can use a similar approach to the one used in the range() method. We can start at the root of the tree and recursively search the left and right subtrees based on whether the point being searched for is closer to the current node. If a point is closer than the current nearest neighbor, we can update the nearest neighbor to be that point. We can continue this process until we reach a leaf node, at which point we can return the nearest neighbor.

nearest\_neighbor():

def nearest\_neighbor(self, point):

nearest = None

def search(node):

nonlocal nearest

if not node:

return

distance = self.distance(point, node.point)

if not nearest or distance < self.distance(point, nearest):

nearest = node.point

if point[0] < node.point[0]:

search(node.left)

else:

search(node.right)

search(self.root)

return nearest