

Übungsblatt 7

Julius Auer, Alexa Schlegel

Aufgabe 1 (Anfragezeit bei kd-Bäumen):

Aufgabe 2 (Implementierung):

Zum Ausführen des Codes muss bitte *orte_deutschland.txt* im selben Verzeichnis wie das Jar liegen, oder wahlweise das Jar mit Angabe des Pfads gestartet werden:

```
java -jar UB7.jar ./<path>/orte_deutschland.txt
```

Um einen kd-Baum in $O(n \cdot (k + \log n))$ konstruieren zu können, benötigt man einen $O(n)$ -Median Algorithmus. Wir haben zu diesem Zweck der Vollständigkeit halber einen *BFPRT* implementiert - dieser ist in der Praxis aber deutlich langsamer als das mittlere Element nach Sortieren mit *Timsort* zu ermitteln. Deshalb wird für den kd-Baum letztgenannter Ansatz verwendet.

Da für das Erstellen eines ausgeglichenen kd-Baums alle Elemente zur Konstruktions-Zeit bekannt sein müssen, wurde auf eine *insert*-Methode verzichtet.

Der kd-Baum akzeptiert als Elemente alle Objekte, die *KDKey* implementieren - über diese Schnittstelle kann für jedes Objekt auf das *i*-te Element des *k*-dimensionalen Schlüssels zugegriffen werden:

```
1 public interface KDKey {  
2     public Double getKey(int dimension);  
3 }
```

Alle übrigen Details sollten den Kommentaren zu entnehmen sein. Es folgt der vollständige Code:

```
1 public class KDTree<T extends KDKey> {  
2     private Node<T> root;  
3  
4     public KDTree(int dimension, List<T> points) {  
5         LinkedList<Integer> dimensions = new LinkedList<Integer>();  
6  
7         for(int i = 0; i < dimension; ++i)  
8             dimensions.addLast(i);  
9  
10        root = new Node<T>(points, dimensions);  
11    }  
12  
13    public boolean contains(T point) {  
14        return root.contains(point);  
15    }  
16  
17    public LinkedList<T> search(double[][] range) {  
18        return root.search(range);  
19    }  
20 }
```

```

1 public class Node<T extends KDKey> {
2     Node<T> leftChild = null;
3     Node<T> rightChild = null;
4     Node<T> midChild = null;
5     T median;
6     int d;
7
8     protected Node(List<T> points, LinkedList<Integer> dimensions) {
9         if(dimensions.isEmpty()) {
10             median = points.get(0);
11             return;
12         }
13
14         d = dimensions.removeFirst();
15         LinkedList<Integer> dimensions_mid = new LinkedList<Integer>();
16         LinkedList<Integer> dimensions_right = new LinkedList<Integer>();
17         dimensions_mid.addAll(dimensions);
18         dimensions.addLast(d);
19         dimensions_right.addAll(dimensions);
20         Collections.sort(points, new Comparator<T>() {
21
22             @Override
23             public int compare(T p1, T p2) {
24                 return p1.getKey(d).compareTo(p2.getKey(d));
25             }
26         });
27
28         median = points.get(points.size() / 2);
29         LinkedList<T> points_left = new LinkedList<T>();
30         LinkedList<T> points_mid = new LinkedList<T>();
31         LinkedList<T> points_right = new LinkedList<T>();
32
33         for(T p : points)
34             if(p.getKey(d).compareTo(median.getKey(d)) < 0)
35                 points_left.add(p);
36             else if(p.getKey(d).compareTo(median.getKey(d)) == 0)
37                 points_mid.add(p);
38             else
39                 points_right.add(p);
40
41         if(!points_left.isEmpty())
42             leftChild = new Node<T>(points_left, dimensions);
43
44         if(!points_mid.isEmpty())
45             midChild = new Node<T>(points_mid, dimensions_mid);
46
47         if(!points_right.isEmpty())
48             rightChild = new Node<T>(points_right, dimensions_right);
49     }
50
51     protected boolean contains(T point) {
52         if(point.equals(median))
53             return true;
54
55         if(point.getKey(d) < median.getKey(d))
56             return leftChild == null ? false : leftChild.contains(point);
57
58         if(point.getKey(d) > median.getKey(d))
59             return rightChild == null ? false : rightChild.contains(point);
60
61         return midChild == null ? false : midChild.contains(point);
62     }
63
64     protected LinkedList<T> search(double[][] range) {
65         LinkedList<T> result = new LinkedList<T>();
66
67         // if "a" is inside the actual range
68         if(median.getKey(d).compareTo(range[d][0]) >= 0 && median.getKey(d).compareTo(
69             range[d][1]) <= 0) {
70             boolean inRange = true;

```

```

70
71     for(int i = 0; i < range.length; ++i)
72         if(median.getKey(i) < range[i][0] || median.getKey(i) > range[i][1])
73             inRange = false;
74
75     if(inRange)
76         result.add(median);
77
78     if(midChild != null)
79         result.addAll(midChild.search(range));
80
81     if(leftChild != null && median.getKey(d).compareTo(range[d][0]) > 0)
82         result.addAll(leftChild.search(range));
83
84     if(rightChild != null && median.getKey(d).compareTo(range[d][1]) < 0)
85         result.addAll(rightChild.search(range));
86 }
87 // if "a" is outside the given range look in left or right child
88 else {
89     if(rightChild != null && median.getKey(d).compareTo(range[d][0]) < 0)
90         result.addAll(rightChild.search(range));
91     else if(leftChild != null && median.getKey(d).compareTo(range[d][1]) > 0)
92         result.addAll(leftChild.search(range));
93 }
94
95 return result;
96 }
97 }

```

Ein Test mit der großen *orte_welt.txt*-Datei wurde nicht durchgeführt, weil der File das Filesize-Limit von Github überschreitet - damit passt er nicht in unseren Workflow :)

Für deutsche Orte wurden alle Orte als schwarze Punkte eingezeichnet und die Orte aus der Ergebnismenge von Beispiel-Anfragen (markiert durch rote Boxen um den Anfrage-Bereich) als grüne Punkte (Abb. 1).

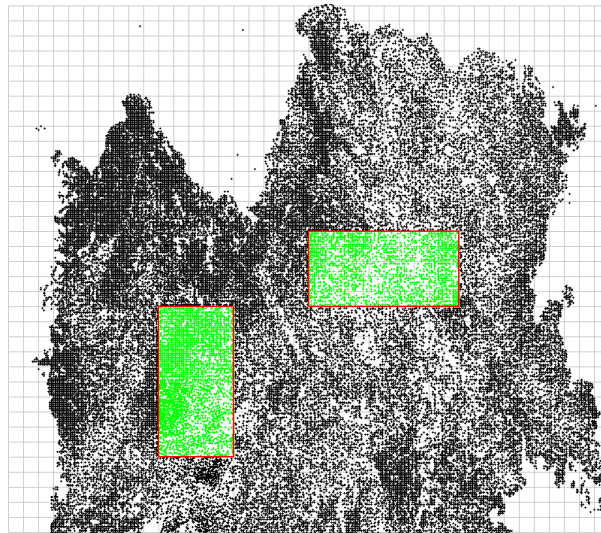


Abbildung 1: Zwei Beispielanfragen