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 }
```

```
public class Node<T extends KDKey> {
    Node<T> leftChild = null;
    Node<T> rightChild = null;
    Node<T> midChild = null;
    T median;
    int d;

    protected Node(List<T> points, LinkedList<Integer> dimensions) {
        if(dimensions.isEmpty()) {
            median = points.get(0);
            return;
        }

        d = dimensions.removeFirst();
        LinkedList<Integer> dimensions_mid = new LinkedList<Integer>();
        LinkedList<Integer> dimensions_right = new LinkedList<Integer>();
        dimensions_mid.addAll(dimensions);
        dimensions.addLast(d);
        dimensions_right.addAll(dimensions);
        Collections.sort(points, new Comparator<T>() {

            @Override
            public int compare(T p1, T p2) {
                return p1.getKey(d).compareTo(p2.getKey(d));
            }
        });

        median = points.get(points.size() / 2);
        LinkedList<T> points_left = new LinkedList<T>();
        LinkedList<T> points_mid = new LinkedList<T>();
        LinkedList<T> points_right = new LinkedList<T>();

        for(T p : points)
            if(p.getKey(d).compareTo(median.getKey(d)) < 0)
                points_left.add(p);
            else if(p.getKey(d).compareTo(median.getKey(d)) == 0)
                points_mid.add(p);
            else
                points_right.add(p);

        if(!points_left.isEmpty())
            leftChild = new Node<T>(points_left, dimensions);

        if(!points_mid.isEmpty())
            midChild = new Node<T>(points_mid, dimensions_mid);

        if(!points_right.isEmpty())
            rightChild = new Node<T>(points_right, dimensions_right);
    }

    protected boolean contains(T point) {
        if(point.equals(median))
            return true;

        if(point.getKey(d) < median.getKey(d))
            return leftChild == null ? false : leftChild.contains(point);

        if(point.getKey(d) > median.getKey(d))
            return rightChild == null ? false : rightChild.contains(point);

        return midChild == null ? false : midChild.contains(point);
    }

    protected LinkedList<T> search(double[][] range) {
        LinkedList<T> result = new LinkedList<T>();

        // if "a" is inside the actual range
        if(median.getKey(d).compareTo(range[d][0]) >= 0 && median.getKey(d).compareTo(
             range[d][1]) <= 0) {
            result.add(median);
```

```
70
71              if(midChild != null)
72                  result.addAll(midChild.search(range));
73
74              if(leftChild != null && median.getKey(d).compareTo(range[d][0]) > 0)
75                  result.addAll(leftChild.search(range));
76
77              if(rightChild != null && median.getKey(d).compareTo(range[d][1]) < 0)
78                  result.addAll(rightChild.search(range));
79          }
80          // if "a" is outside the given range look in left or right child
81          else {
82              if(rightChild != null && median.getKey(d).compareTo(range[d][0]) < 0)
83                  result.addAll(rightChild.search(range));
84              else if(leftChild != null && median.getKey(d).compareTo(range[d][1]) > 0)
85                  result.addAll(leftChild.search(range));
86          }
87
88          return result;
89      }
90  }
```

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).

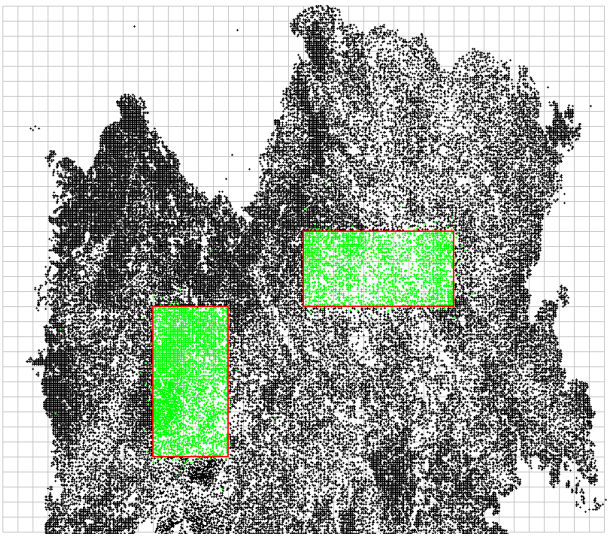

Abbildung 1: Zwei Beispielanfragen

Im Moment scheint es irgendwo noch einen Bug zu geben - grüne Orte außerhalb des Anfragebereichs sollten eigentlich nicht auftreten.