

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 AlGeo.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     /**
3      * Get the value of this key at given dimension
4      *
5      * @param dimension
6      * @return value of this key at dimension 'dimension'
7      */
8     public double getKey(int dimension);
9 }
```

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     /**
9      * Build the tree:
10     * - find the median of points
11     * - split the list into three lists: <, =, >
12     * - create a new child node for each non empty list
13     *
14     * @param points
15     * @param dimensions list of indices of still valid dimensions (ref inline
16     * doc for details)
17     */
18     protected Node(List<T> points, LinkedList<Integer> dimensions) {
19         // at the end of a -=branch there must be only one element left (or
20         // several elements with the same key). Elements with duplicate keys are
21         // ignored silently. One could add the whole list instead, but we
22         // don't.
23         if(dimensions.isEmpty()) {
24             median = points.get(0);
25             return;
26         }
27
28         // 'dimensions' keeps track of the dimensions which are not excluded yet
29         // by following a -=branch. The first one becomes the split-dimension
30         // for this node and is moved to the end of the list for the < and >
31         // branches. For the -=branch this dimension is removed from the list.
32         d = dimensions.removeFirst();
33         LinkedList<Integer> dimensions_mid = new LinkedList<Integer>();
34         dimensions_mid.addAll(dimensions);
35         dimensions.addLast(d);
36
37         // find the median on dimension d
38         Collections.sort(points, new Comparator<T>() {
39
40             @Override
41             public int compare(T p1, T p2) {
42                 return new Double(p1.getKey(d)).compareTo(new Double(p2.getKey(d)));
43             }
44         });
45
46         median = points.get(points.size() / 2);
47
48         // split the list
49         LinkedList<T> points_left = new LinkedList<T>();
50         LinkedList<T> points_mid = new LinkedList<T>();
51         LinkedList<T> points_right = new LinkedList<T>();
52
53         for(T p : points)
54             if(p.getKey(d) < median.getKey(d))
55                 points_left.add(p);
56             else if(p.getKey(d) == median.getKey(d))
57                 points_mid.add(p);
58             else
59                 points_right.add(p);
60
61         // create a new child for each non empty list
62         if(!points_left.isEmpty())
63             leftChild = new Node<T>(points_left, dimensions);
64
65         if(!points_mid.isEmpty())

```

```

66         midChild = new Node<T>(points_mid, dimensions_mid);
67
68         if(!points_right.isEmpty())
69             rightChild = new Node<T>(points_right, dimensions);
70     }
71
72     protected boolean contains(T point) {
73         if(point.equals(median))
74             return true;
75
76         if(point.getKey(d) < median.getKey(d))
77             return leftChild == null ? false : leftChild.contains(point);
78
79         if(point.getKey(d) > median.getKey(d))
80             return rightChild == null ? false : rightChild.contains(point);
81
82         return midChild == null ? false : midChild.contains(point);
83     }
84
85     /**
86      * Find all elements in a given range
87      *
88      * @param range one dupel (min,max) for each dimension of the data
89      * @return list of elements
90      */
91     protected LinkedList<T> search(double[][] range) {
92         LinkedList<T> result = new LinkedList<T>();
93
94         // median is inside range at dimension d
95         if(median.getKey(d) >= range[d][0] && median.getKey(d) <= range[d][1]) {
96             // check if median is inside range at all dimensions
97             boolean inRange = true;
98
99             for(int i = 0; i < range.length; ++i)
100                 if(median.getKey(i) < range[i][0] || median.getKey(i) > range[i][1]) {
101                     inRange = false;
102                     break;
103                 }
104
105             // add valid elements to result set
106             if(inRange)
107                 result.add(median);
108
109             if(midChild != null)
110                 result.addAll(midChild.search(range));
111
112             if(leftChild != null && median.getKey(d) > range[d][0])
113                 result.addAll(leftChild.search(range));
114
115             if(rightChild != null && median.getKey(d) < range[d][1])
116                 result.addAll(rightChild.search(range));
117         }
118         // median is NOT inside range at dimension d
119         else {
120             if(rightChild != null && median.getKey(d) < range[d][0])
121                 result.addAll(rightChild.search(range));
122             else if(leftChild != null && median.getKey(d) > range[d][1])
123                 result.addAll(leftChild.search(range));
124         }
125
126         return result;
127     }
128 }

```

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).

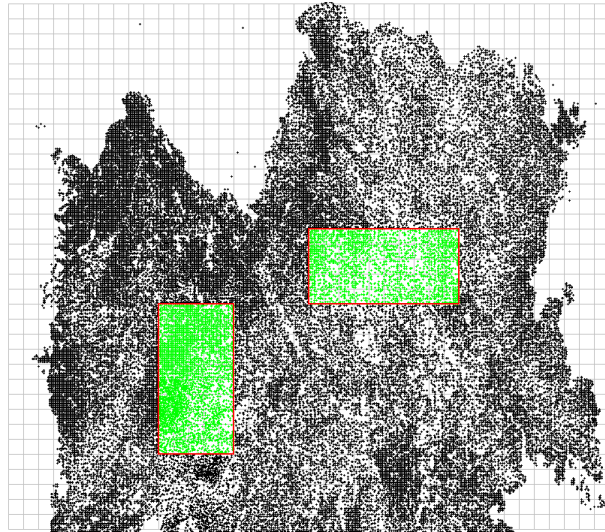


Abbildung 1: Zwei Beispielanfragen