Übungsblatt 7

Julius Auer, Alexa Schlegel

Aufgabe 1 (Anfragezeit bei kd-Bäumen):

Die Argumentation ist die Selbe wie für $k = 2$:
(1) Seien für einen Knoten v, dessen Region R_v und eine Suchregion U mit einer beliebigen, festen Kante L zunächst $R_v \cap L \neq \emptyset$. Für $k = 2$ war zu beobachten, dass von den 4 primären Enkelregionen von v maximal 2 von L geschnitten werden können. Diese Beobachtung lässt sich alternativ auch so interpretieren, dass in der Dimension in der L eine Kante ist, auf jeden Fall ein Kind nicht betrachtet wird. Auch für $k > 2$ wird in dieser einen Dimension ein Kind nicht betrachtet - in allen $k - 1$ anderen können beide Pfade im Baum verfolgt werden. Die Rekursion für die Anzahl besuchter Primärknoten p_s in Tiefe s ergibt sich somit zu:

$$p_{s+k} \leq 2 \cdot (k-1) \cdot p_s$$

mit $k - 1$ Rekursionsankern. Für $k = 2$ und $k = 3$ erhält man exemplarisch:

$p_0 = 1$	$p_0 = 1$
$p_2 \leq 2$	$p_3 \leq 4$
$p_4 \leq 4$	$p_6 \leq 16$
$p_6 \leq 8$	$p_9 \leq 64$
$p_8 \leq 16$	$p_{12} \leq 256$
...	...
$p_s \leq 2^{\lceil \frac{s}{2} \rceil}$	$p_s \leq 2^{\lceil \frac{2}{3} \cdot s \rceil}$

Oder im Allgemeinen für beliebiges k:

$$p_s \leq 2^{\lceil \left(1 - \frac{1}{k}\right) \cdot s \rceil}$$

Diese Beobachtung verbleibt - wie auch in der Vorlesung im Fall $k = 2$ gesehen - ohne Beweis.

Alles Weitere verhält sich wie für den Fall $k = 2$ in der Vorlesung betrachtet und sei im Folgenden nur kurz zusammengefasst:

Neben dem oben geschilderten Fall können noch zwei weitere auftreten:

(2) $R_v \subset U$: ist durch $O(m)$ abgedeckt.

(3) Suchen in einem =-Teilbaum benötigt höchstens (wenn die Region nicht ganz in L liegt) $2 \cdot (\log n - s)$ Schritte.

Somit erhält man insgesamt für die Anzahl besuchter Knoten:

$$
\begin{aligned}
&= \sum_{s=0}^{\lceil \log n \rceil} \overbrace{2^{\lceil 1-\frac{1}{k} \rceil \cdot s}}^{(1)} \cdot \left(\overbrace{1}^{v} + \overbrace{2 \cdot (\log n - s)}^{(3)} \right) \\
&\leq 2 \cdot \sum_{s=0}^{(\log n)+1} 2^{(1-\frac{1}{k}) \cdot s} \cdot (1 + 2 \cdot (\log n - s)) \\
&= 2 \cdot \sum_{s=0}^{(\log n)+1} \left(2^{1-\frac{1}{k}}\right)^{\log n} \cdot \frac{\left(2^{1-\frac{1}{k}}\right)^{s}}{\left(2^{1-\frac{1}{k}}\right)^{\log n}} \cdot (1 + 2 \cdot (\log n - s)) \\
&= 2 \cdot n^{1-\frac{1}{k}} \cdot \sum_{s=0}^{(\log n)+1} \left(2^{1-\frac{1}{k}}\right)^{s - \log n} \cdot (1 + 2 \cdot (\log n - s)) \\
&= 2 \cdot n^{1-\frac{1}{k}} \cdot \sum_{s=0}^{(\log n)+1} \frac{1 + 2 \cdot (\log n - s)}{\left(2^{1-\frac{1}{k}}\right)^{\log n - s}}
\end{aligned}
$$

Der Term in der Summe konvergiert und unter Berücksichtigung von (2) ergibt sich die Anzahl ingesamt besuchter Knoten zu:

$$
O\left(n^{1-\frac{1}{k}} + m\right)
$$

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:

```java
public interface KDKey {
    /**
     * Get the value of this key at given dimension
     *
     * @param dimension
     * @return value of this key at dimension 'dimension'
     */
    public double getKey(int dimension);
}
```

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

```java
public class KDTree<T extends KDKey> {
    private Node<T> root;

    public KDTree(int dimension, List<T> points) {
        LinkedList<Integer> dimensions = new LinkedList<Integer>();

        for(int i = 0; i < dimension; ++i)
            dimensions.addLast(i);

        root = new Node<T>(points, dimensions);
    }

    public boolean contains(T point) {
        return root.contains(point);
    }

    public LinkedList<T> search(double[][] range) {
        return root.search(range);
    }
}
```

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

    /**
     * Build the tree:
     * - find the median of points
     * - split the list into three lists: <, =, >
     * - create a new child node for each non empty list
     *
     * @param points
     * @param dimensions list of indizes of still valid dimensions (ref inline
     * doc for details)
     */
    protected Node(List<T> points, LinkedList<Integer> dimensions) {
        // at the end of a =-branch there must be only one element left (or
        // several elements with the same key). Elements with duplicate keys are
        // ignored silently. One could add the whole list instead, but we
        // don't.
        if(dimensions.isEmpty()) {
            median = points.get(0);
            return;
        }

        // 'dimensions' keeps track of the dimensions which are not excluded yet
        // by following a =-branch. The first one becomes the split-dimension
        // for this node and is moved to the end of the list for the < and >
        // branches. For the =-branch this dimension is removed from the list.
        d = dimensions.removeFirst();
        LinkedList<Integer> dimensions_mid = new LinkedList<Integer>();
        dimensions_mid.addAll(dimensions);
        dimensions.addLast(d);

        // find the median on dimension d
        Collections.sort(points, new Comparator<T>() {

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

        median = points.get(points.size() / 2);

```

```
        // split the list
        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) < median.getKey(d))
                points_left.add(p);
            else if(p.getKey(d) == median.getKey(d))
                points_mid.add(p);
            else
                points_right.add(p);

        // create a new child for each non empty list
        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);
    }

    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);
    }

    /**
     * Find all elements in a given range
     *
     * @param range one dupel (min,max) for each dimension of the data
     * @return list of elements
     */
    protected LinkedList<T> search(double[][] range) {
        LinkedList<T> result = new LinkedList<T>();

        // median is inside range at dimension d
        if(median.getKey(d) >= range[d][0] && median.getKey(d) <= range[d][1]) {
            // check if median is inside range at all dimensions
            boolean inRange = true;

            for(int i = 0; i < range.length; ++i)
                if(median.getKey(i) < range[i][0] || median.getKey(i) > range[i][1]) {
                    inRange = false;
                    break;
                }

            // add valid elements to result set
            if(inRange)
                result.add(median);

            if(midChild != null)
                result.addAll(midChild.search(range));

            if(leftChild != null && median.getKey(d) > range[d][0])
                result.addAll(leftChild.search(range));

            if(rightChild != null && median.getKey(d) < range[d][1])
                result.addAll(rightChild.search(range));
        }
```

```java
        // median is NOT inside range at dimension d
        else {
            if(rightChild != null && median.getKey(d) < range[d][0])
                result.addAll(rightChild.search(range));
            else if(leftChild != null && median.getKey(d) > range[d][1])
                result.addAll(leftChild.search(range));
        }

        return result;
    }
}
```

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).

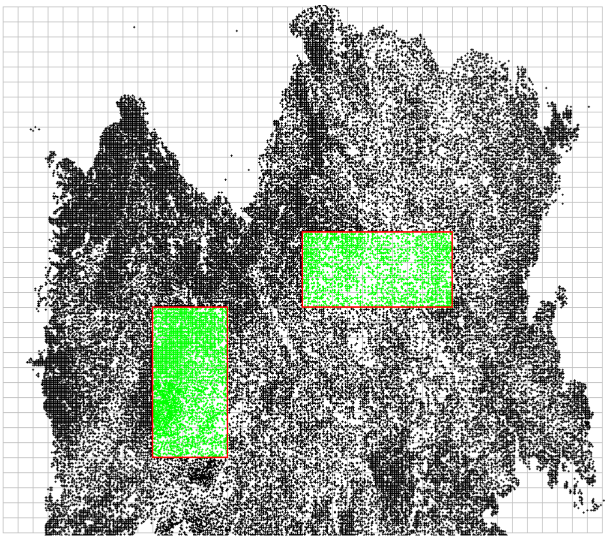

Abbildung 1: Zwei Beispielanfragen