Aufgabe zu KD Bäumen

Sven Eric Panitz
www.panitz.name

24. November 2010

Zusammenfassung

Eine simple Implementierung für KD-Bäume auf RGB Farbräumen wird als Aufgabe gestellt.

1 KD-Bäume

KD-Bäume sind eine spezielle Art von binären Suchbbäumen. Binäre Suchbbäume sind eine Datenstruktur für eine schnelle Suche in einer Menge von Elementen, für die eine Größer-Relation existiert. Jeder Baumknoten ist mit einem Element markiert. Für alle Baumknoten gilt: alle Elemente, die im linken Kindbaum des betrachteteten Knotens sind, sind kleiner in der Relation als das Element im Knoten, alle Elemente im rechten Kindbaum sind größer.

Werden neue Knoten in einem binären Suchbaum eingefügt, so werden sie als neue Blätter eingefügt und zwar so, dass die obige Eigenschaft nicht verletzt wird.

KD-Bäume verallgemeinern dieses Prinzip auf mehrdimensionale Daten. Nun wird zusätzlich in jedem Knoten vermerkt, auf welcher der Dimensionen in diesem Knoten der Vergleich einer Größer-Relation angewendet wird. Damit gilt die Eigenschaft:

Für alle Baumknoten gilt: alle Elemente, die im linken Kindbaum des betrachteteten Knotens sind, sind in der an diesem Baumknoten vermerkten Dimension kleiner in der Relation als das Element im Knoten, alle Elemente im rechten Kindbaum sind in dieser Dimension größer.

In den meisten Anwendungen von KD-Bäumen wird nicht eine Menge von mehrdimensionalen Daten gespeichert, sondern eine Abbildung (`Map`) von diesen mehrdimensionalen Daten auf eine beliebige Menge gespeichert. Eine Abbildung lässt sich stets gut als eine Menge von Paaren darstellen.

Somit werden wir die einfache Klasse für die Speicherung von Paaren von Elementen vorsehen.

Pair.java

```
1 public class Pair<A, B> {
2   A fst;
3   B snd;
4   public Pair(A fst, B snd) {
5     this.fst = fst;
6     this.snd = snd;
7   }
8   @Override
9   public String toString() {
```

```
10      return "("+fst+", "+snd+")";
11    }
12  }
```

2 Eine Schnittstelle für KD-Bäume auf RGB-Farbräume

Wir wollen in der Aufgabe KD-Bäume benutzen, um Farbpunkte des RGB-Farbraums zu klassifizieren. Farbpunkte sind dreidimensionale Daten mit den drei Dimensionen für rot, grün und blau.

Der Typ, auf den die Farbpunkte abgebildet werden kann offen gehalten werden. Daher sehen wir ein generisches Interface hierfür vor.

KD.java

```
1  import java.awt.Color;
2  import java.util.List;
3
4  public interface KD<A> {
```

Laut Definition von KD-Bäumen ist in jedem Knoten zu vermerken, nach welcher Dimension in diesem Knoten projeziert wird, um die Vergleichsrelation anzuwenden.

Hierzu sei eine interne Aufzählungsklasse definiert, die für die drei Farben definiert ist.

KD.java

```
5    enum ColorCut {red, green, blue;
```

Eine simple Methode soll uns ermöglichen in dieser Aufzählung uns einen nächsten Aufzählungswert zu geben. Diese kann benutzt werden, um Ebenen-weise im KD-Baum immer auf eine andere Farbe zu projezieren:

KD.java

```
6      ColorCut next(){return values()[(this.ordinal()+1)%3];};
```

Die folgende Methode extrahiert für ein `java.awt.Color` Objekt den Wert, der durch das Aufzählungsobjekt `ColorCut` festgelegten Farbkomponente, also der Rotwert, Grünwert oder Blauwert.

KD.java

```
7      int getColorValue(Color c){
8        switch (this) {
9          case red: return c.getRed();
10         case blue: return c.getBlue();
11         default: return c.getGreen();
12       }
13     }
14   }
```

Soweit zur inneren Aufzählungsklasse der Schnittstelle KD.

Um neue leer KD-Bäume aus einem existieren KD-Baum erzeugen zu können, sei eine einfache Methode vorgeschrieben, die neue Instanzen erzeugen kann.

KD.java

```
15   KD<A> newInstance();
```

2.1 Getter und Setter

Die Methoden in diesem Abschnitt geben im Prinzip vor, welche Felder eine Implementierung von KD Bäumen braucht. Es sind jeweils Getter-/Setter-Paare für die erwarteten Felder.

Da es sich um Binärbäume handelt muss es Möglichkeiten geben die Kinder zu erfragen und zu setzen. Daher ein entsprechendes Methodenpaar für das linke Kind:

KD.java

```
16  void setLeft(KD<A> p);
17  KD<A>  getLeft();
```

Wenn es kein linkes Kind gibt, soll die Get-Methode `null` als Ergebnis liefern.

Analog die beiden Methoden für das rechte Kind.

KD.java

```
18  void setRight(KD<A> p);
19  KD<A>  getRight();
```

Die Implementierung der KD-Bäume soll auch die Möglichkeit eröffnen, im Baum von den Kind auf sein Elternknoten zu gehen. Daher ist auch hierfür ein Methodenpaar vorgesehen.

KD.java

```
20  KD<A> getParent();
21  void setParent(KD<A> p);
```

Nur für den Wurzelknoten liefert die Get-Methode `null` als Ergebnis.

Eine weitere wichtige Eigenschaft eines Baumknotens ist, auf welche Dimension hier zum Vergleich projeziert wird. Hierzu wurde die innere Aufzählungsklasse definiert. Für diese Information muss auch ein Getter-/Setter-Paar implementiert werden.

KD.java

```
22  void setColorCut(ColorCut cc);
23  ColorCut getColorCut();
```

Und schließlich soll ja in jedem Baumknoten noch das eigentliche Element gespeichert sein. Ein Element besteht in unserem Fall aus einem Farbobjekt gepaart mit der Kategorie, die diesem Farbwert zugeordnet ist. So ergeben sich die folgende Get- und Set-Methoden.

KD.java

```
24  Pair<Color, A> getElement();
25  void setElement(Pair<Color, A> element);
```

2.2 Einfügen neuer Knoten

Um einen KD-Baum mit korrekter Sucheigenschaft aufzubauen, dürfen nicht willkürlich Kinderknoten gesetzt werden, sondern die neue Elemente sind als neues Blatt entsprechend einzuhängen. Daher ist die zentrale Methode die Methode zum Einfügen eines Knotens.

KD.java

```
26  void insert(Pair<Color,A> p);
```

Algorithmisch ist diese Aufgabe, wie man es von binären Suchbäumen kennt, umzusetzen. Zu berücksichtigen ist, dass nur der Schlüssel, also das `Color`-Objekt, also das Element `fst` des Paares hierbei betrachtet wird.

Betrachte den einzufügenden Knoten:

- Hat er dieselbe Farbe, wie das Element, an diesem Baumknoten, so braucht nicht mehr eingefügt zu werden.
- Andernfalls vergleiche den einzufügenden Knoten auf der Dimension, auf der dieser Baumknoten zum Vergleich projeziert. Entscheide, ob das neue Element im linken oder rechten Kindbaum einzufügen ist.
  - Ist der entsprechende Kindbaum `null`, dann erzeuge dort ein neues Blatt, mit dem einzufügenden neuen Element und setze diesen Knoten als Elternknoten des neuen Blattes.
  - Andernfalls rufe die Einfügemethode rekursiv für das entsprechende Kind auf.

2.3 Suche nach dem nächsten Nachbarn

Die entscheiden Anwendung von KD-Bäumen ist nicht, eine Menge zu realisieren, in der exakt ein Wert gesucht wird, sondern den Wert zu finden, der einem Suchwert am nächsten kommt.

KD.java

```
27  Pair<Color,A> nextNeighbour(Color c);
```

In einem einfachen binären Suchbaum geht diese Prüfung relativ einfach. Hier braucht der nur Baum wie beim Einfügen durchlaufen zu werden, bis zu der Stelle an der wenn man in den Baum absteigt, sich der Wert vom Suchwert weiter entfernen würde.

Bei KD-Bäumen reicht nicht aus, sich nur das Kind anzuschauen, in dem der Suchknoten eingehängt werden muss.

Hierzu muss man sich noch einmal klar machen, was Nähe im mehrdimensionalen Raum bedeutet. Es ist der euklidischer Abstand. Dieser ändert sich natürlich mit den Werten aller Dimensionen, nicht nur mit der Dimension, auf der für den Vergleich projeziert wird; d.h. man muss sich beide Kinder anschauen, um einen nächsten Nachbarn zu suchen. Damit hätte man durch die Baumstruktur nichts gewonnen, denn so müsste ja doch der komplette Baum durchsucht werden.

Es gibt aber eine Situation, in der das andere Kind nicht mehr betrachtet werden muss. Wenn der Abstand vom aktuellen Knoten allein auf der projezierten Dimension schon weiter ist als der Gesamtabstand der bisher als bestmöglich gefunden wurde, braucht nicht mehr in das zweite Kind zur Suche abgestiegen zu werden. Denn hier kommen wir auf dieser Dimension nicht mehr näher dran.

In weniger Prosa ausgedrückt, kann man den Algorithmus in Pseudocode wie in Wikipedia wie folgt aufschreiben:

```
1 function kdsearchnn( here, point, best )
2     if here == nil then
3         return best
4     end
```

```
5
6       if best == nil then
7           best = here
8       end
9
10      -- consider the current node --
11      if distance(here,point) < distance(best,point) then
12          best = here
13      end
14
15      -- search the near branch --
16      child = child_near(here,point)
17      best = kdsearchnn( child, point, best )
18
19      -- search the away branch - maybe --
20      if distance_axis(here,point) < distance(best,point) then
21          child = child_away(here,point)
22          best = kdsearchnn( child, point, best )
23      end
24      return best
25  end
```

Zum Verständnis: `here` ist der aktuelle Knoten, `point` ist unserem Fall, die Farbe, für die wir die nächst ähnliche Farbe suchen und `best` ist der Baumknoten mit der nächstmöglichen Farbe, die wir suchen. Die Funktion wird am sinvollsten mit der Wurzel als `here` und als `best` sowie der Suchfarbe als `point` aufgerufen.

Der obige Algorithmus benutzt drei Hilfsfunktionen, die nicht ganz selbsterklärend sind:

- `child_near`, dieses ist das Kind, in dem der Suchknoten einzufügen wäre.
- `child_away`, dieses ist das Kind, in dem der Suchknoten nicht einzufügen wäre.
- `distance_axis` ist die Entfernung allein auf der Dimension, auf der in diesem Baumknoten projeziert wurde.

Mit diesen Zusatzinformationen sollte sich der obige Pseudocode ziemlich eins zu eins in Ihre Javaimplementierung umsetzen lassen.

2.4 Kategorisieren

Ziel des ganzen Spiels ist es, zunächst anhand eines Lernbildes Farbpunkten Kategorien zuzurodenen und anschließend für weitere Farben eine Kategorie zu bestimmen. Hierzu sei die Methode `categorize` vorgesehen

KD.java

```
26   List<A> categorize(Color c);
27  }
```

Diese gibt nicht eine einzige Kategorie als Ergebnis zurück, sondern eine Liste von Kategoriewerten. Mit unserer obigen Methode für den nächsten Nachbarn, lässt sich die Methode simpel

so implementieren, dass eine einelementige Liste mit der Kategorie, die beim nächsten Nachbarn gespeichert ist, als Ergebnis zurück gegeben wird.

In der Praxis wird bei Bilderkennungsverfahren aber nicht nur der allernächste Nachbar zum Kategorisieren genommen, sondern die k nächsten Nachbarn und die Liste derer Kategorien. Dann kann eventuell ein Mittelwert von diesen gebildet werden, oder eine Mehrheitsentscheidung getroffen werden.

Die k-nächsten Nachbarn zu berechnen, soll nicht mehr Teil des Aufgabenblattes sein und steht jedem frei als zusätzliche Übung zu implementieren.