**RWTH Aachen**
**Lehrgebiet Theoretische Informatik**
Frohn–Reidl–Rossmanith–Sánchez–Ströder

**WS 2013/14**
Übungsblatt 7
09.12.2013

# Übung zur Vorlesung Programmierung

## Aufgabe T0

In dieser Aufgabe sollen Sie die Klasse *KMedian* (siehe unten), die eine Implementierung des Algorithmus $k$-Medians bereitstellt, vervollständigen. $k$-Medians ist ein Algorithmus, der beliebige Daten in Untermengen—auch Cluster genannt—unterteilt, sodass sich die Elemente in jedem Cluster möglichst ähnlich sind. Die Ähnlichkeit wird dabei über eine gegebene *Distanzfunktion* gemessen.

Genauer: Der Algorithmus $k$-Medians unterteilt die Menge der Eingabedaten $E$ in Teilmengen $E_1, \ldots, E_k$, sodass jedes Element der Menge $E$ in genau einer Menge $E_i, i \in \{1, \ldots, k\}$ enthalten ist. Dazu geht der Algorithmus wie folgt vor:

1. Wähle zufällig $k$ unterschiedliche Elemente $e_1, \ldots, e_k$ der Menge $E$ als *Mittelpunkte* der Mengen $E_1, \ldots, E_k$.
2. Ordne jedes Element $e \in E$ jener Menge $E_i$ zu, deren Mittelpunkt $e$ bezüglich der verwendeten Distanzfunktion am nächsten ist.
3. Lege für jede Menge $E_i$ den *Median* von $E_i$ als neuen Mittelpunkt $e_i$ fest.
4. Falls das Abbruchkriterium erfüllt ist, gebe $\{(e_1, E_1), \ldots (e_k, E_k)\}$ zurück, ansonsten fahre mit Schritt 2 fort.

Für eine gegebene Distanzfunktion *dist* ist die Funktion *median* folgendermaßen definiert:

$$median\left(\{m_1, \ldots, m_k\}\right) := \underset{m \in \{m_1, \ldots, m_k\}}{\arg\min} \left( \sum_{i=1}^{k} dist\left(m, m_i\right) \right)$$

Die Distanzfunktion wird dem Konstruktor der Klasse *KMedian* in Form eines Objektes übergeben, dessen Klasse das Interface *Metric* implementiert. Dieses Interface stellt die Methode *distance*($P$ *e1*, $P$ *e2*) zur Verfügung.

In dieser Aufgabe wählen wir als Abbruchkriterium schlicht, dass der Algorithmus $n$ Iterationen durchlaufen hat. Wie die Menge $E$ und der Wert $k$ ist auch $n$ ein Parameter der Methode *cluster*(*Collection* $e$, **int** $k$, **int** $n$).

Vervollständigen Sie die Klasse an allen Orten, die mit einem //TODO markiert sind.

```
public class KMedian⟨P⟩ {
  private final Metric⟨P⟩ metric;
  public KMedian(Metric⟨P⟩ metric) {
    this.metric = metric;
  }
  public Map⟨P, Set⟨P⟩⟩ cluster(Collection⟨P⟩ e, int k, int n) {
    // Create copy of set first
```

```
    ArrayList⟨P⟩ elements = new ArrayList⟨P⟩();
    elements.addAll(e);
    Random rnd = new Random();
    if (elements.size() < k) {
      // k is too large for the given set e
      return null;
    }
    ArrayList⟨P⟩ centers;
    // TODO Choose centers randomly
    Map⟨P, Set⟨P⟩⟩ clustering = null;
    for (int i = 0; i < n; i++) {
      // TODO Cluster
    }
    return clustering;
  }
  /* cluster the elements s.t. each element is mapped to the nearest center */
  public Map⟨P, Set⟨P⟩⟩ cluster(Collection⟨P⟩ centers, Collection⟨P⟩ elements) {
    // TODO
    return null;
  }
  /* find the new centers, i.e., calculate the median for each cluster */
  public ArrayList⟨P⟩ findNewCenters(Map⟨P, Set⟨P⟩⟩ clustering) {
    // TODO
    return null;
  }
}
```

**Lösungsvorschlag**

```
public class KMedian⟨P⟩ {
  private final Metric⟨P⟩ metric;
  public KMedian(Metric⟨P⟩ metric) {
    this.metric = metric;
  }
  public Map⟨P, Set⟨P⟩⟩ cluster(Collection⟨P⟩ e, int k, int n) {
    ArrayList⟨P⟩ elements = new ArrayList⟨P⟩();
    elements.addAll(e);
    Random rnd = new Random();
    if (elements.size() < k) {
      return null;
    }
    ArrayList⟨P⟩ centers = new ArrayList⟨P⟩();
    for (int i = 0; i < k; i++) {
      int num = rnd.nextInt(elements.size());
      centers.add(elements.remove(num));
    }
    elements.addAll(centers);
    Map⟨P, Set⟨P⟩⟩ clustering = null;
```

```
  for (int i = 0; i < n; i++) {
    clustering = cluster(centers, elements);
    centers = findNewCenters(clustering);
  }
  return clustering;
}
public Map<P, Set<P>> cluster(Collection<P> centers, Collection<P> elements) {
  HashMap<P, Set<P>> clustering = new HashMap<P, Set<P>>();
  for (P c : centers) {
    clustering.put(c, new HashSet<P>());
  }
  for (P element : elements) {
    P center = findClosestCenter(centers, element);
    clustering.get(center).add(element);
  }
  return clustering;
}
public ArrayList<P> findNewCenters(Map<P, Set<P>> clustering) {
  ArrayList<P> centers = new ArrayList<P>();
  for (Map.Entry<P, Set<P>> entry : clustering.entrySet()) {
    Set<P> cluster = entry.getValue();
    centers.add(findMedian(cluster));
  }
  assert clustering.keySet().size() == centers.size();
  return centers;
}
private P findClosestCenter(Collection<P> centers, P element) {
  assert!centers.isEmpty();
  double minDistance = Double.MAX_VALUE;
  P center = null;
  for (P c : centers) {
    double dist = metric.distance(c, element);
    if (dist < minDistance) {
      minDistance = dist;
      center = c;
    }
  }
  return center;
}
private P findMedian(Set<P> cluster) {
  P center = null;
  double minDistance = Double.MAX_VALUE;
  for (P candidate : cluster) {
    double dist = 0;
    for (P other : cluster) {
      dist += metric.distance(candidate, other);
      if (dist > minDistance) {
        break;
```

```
        }
      }
      if (dist < minDistance) {
        minDistance = dist;
        center = candidate;
      }
    }
    return center;
  }
}
```

**Aufgabe H0 (5 Punkte)**[2]

Implementieren Sie eine Klasse *MapInverter*, die eine Methode zur Inversion einer *Map* zur Verfügung stellt. Eine Inversion ist hierbei eine neue Map, bei der die *values* auf die *keys* der alten Map abgebildet werden. Da die ursprüngliche Map nicht zwangsläufig injektiv ist[1], muss das Ergebnis der Methode eine Map des Typs $Map\langle B, Set\langle A\rangle\rangle$ sein, wobei die übergebene Map den Typ $Map\langle A, B\rangle$ hat.

Die Methode soll also folgende Signatur haben:

**public** *java.util.Map*$\langle B, java.util.Set\langle A\rangle\rangle$ *invert*(*java.util.Map*$\langle A, B\rangle$ *map*)

Sie können bei der Lösung der Aufgabe davon ausgehen, dass alle Objekte des Typs $A$ oder $B$ sinnvolle Implementierungen der Methoden *hashCode* und *equals* zur Verfügung stellen.

**Lösungsvorschlag**

```
public class MapInverter⟨A, B⟩ {
  public Map⟨B, Set⟨A⟩⟩ invert(Map⟨A, B⟩ map) {
    Map⟨B, Set⟨A⟩⟩ res = new HashMap⟨⟩();
    for (Entry⟨A, B⟩ e : map.entrySet()) {
      A key = e.getKey();
      B value = e.getValue();
      if (!res.containsKey(value)) {
        res.put(value, new HashSet⟨A⟩());
      }
      res.get(value).add(key);
    }
    return res;
  }
}
```

**Aufgabe H1 (5 Punkte)**

In dieser Aufgabe sollen Sie beim Entwurf eines EDV-Systems für einen Fahrzeughändler mithelfen. Der Fahrzeughändler handelt mit

[1] Das heißt es kann sein, dass die Map unterschiedlichen *keys* den gleichen *value* zuordnet, wobei zwei Valueobjekte *o1* und *o2* in diesem Kontext als "gleich" betrachtet werden, wenn *o1.equals*(*o2*) gilt.

- Fahrrädern,
- Einrädern,
- Mopeds,
- Motorrollern,
- PKWs,
- LKWs,
- Kutschen,
- Draisinen und
- Zügen.

Ihre Kollegen haben bereits festgelegt, dass zur Umsetzung des Systems folgende Typen benötigt werden:

- ein Typ für jede gehandelte Fahrzeugkategorie,
- *Fahrzeug*,
- *Helmpflichtig* (Beachten Sie, dass die Helmpflicht für Ein- und Fahrräder zwischenzeitlich eingeführt wurde!),
- *Zweiraedrig*,
- *MotorisiertesFahrzeug*,
- *Vierraedrig*,
- *UnmotorisiertesFahrzeug*,
- *PersonenTransportfaehig* (für Fahrzeuge, die dafür ausgelegt sind, neben dem Fahrer mindestens zwei weitere Personen zu transportieren) und
- *Pedalgetrieben*

Welche dieser Typen würden Sie als Interfaces, welche als abstrakte Klassen und welche als konkrete Klassen umsetzen? Geben Sie jeweils eine kurze Begründung für Ihre Entscheidung an! Falls Ihre Begründungen in mehreren Fällen gleich sind, ist es ausreichend, die Begründung einmal anzugeben, falls deutlich wird, für welche Typen sie zutrifft.

**Lösungsvorschlag**
*Fahrrad*, *Einrad*, ... sollten konkrete Klassen sein, da dies die spezialisiertesten Typen der modellierten Domäne sind.
*Fahrzeug*, *MotorisiertesFahrzeug* und *UnmotorisiertesFahrzeug* sollten abstrakte Klassen sein, da zu diesen kein eins-zu-eins Äquivalent in der modellierten Domäne existiert: Es gibt kein Fahrzeug, das weder ein motorisiertes Fahrzeug noch ein unmotorisiertes Fahrzeug ist; Es gibt kein unmotorisiertes Fahrzeug, das weder Einrad noch Fahrrad usw. ist.
Alle anderen Typen sollten als Interfaces realisiert werden, da sie Eigenschaften von Objekten benennen, für sich betrachtet aber kein Gegenstück in der modellierten Domäne haben: Es existiert kein “Helmpflichtig”, es existieren aber helmpflichtige Fahrzeuge.

**Aufgabe H2 (6 Punkte)**

Betrachten Sie die folgende Klasse *Element⟨E⟩*, welche ein Element einer zyklischen Liste implementiert und dabei eine innere Klasse *Parity* bereitstellt, welche gerade und ungerade Zahlen voneinander unterscheiden, sonst aber alle Zahlen als gleich ansehen soll.

```
import java.util.*;
public class Element⟨E⟩ {
    public static class Parity {
        private int value;
        public Parity(int v) {
            this.value = v;
        }
        public boolean equals(Object o) {
            return o instanceof Parity && ((Parity) o).value%2 == this.value%2;
        }
        public int getCreationValue() {
            return this.value;
        }
        public int hashCode() {
            return this.value%3;
        }
        public boolean isEven() {
            return this.value%2 == 0;
        }
        public String toString() {
            return "" + this.value;
        }
    }
    public static void main(String[ ] args) {
        Element⟨Parity⟩ e = new Element⟨Parity⟩(new Parity(1));
        e.add(new Parity(2));
        e.add(new Parity(3));
        System.out.println(e.contains(new Parity(2)));
        Set⟨Parity⟩ set = new LinkedHashSet⟨Parity⟩();
        set.add(new Parity(1));
        set.add(new Parity(2));
        Parity p = new Parity(3);
        if (!set.contains(p)) {
            for (Parity o : set) {
                if (o.equals(p)) {
                    System.out.println("This must not happen!");
                }
            }
        }
        set.add(p);
        set.add(new Parity(0));
        set.add(new Parity(6));
        System.out.println(set);
```

```
    }
    private Element<E> next;
    private final E value;
    public Element(E v) {
        assert (v != null) : "Value must not be null!";
        this.value = v;
        this.next = this;
    }
    public void add(E v) {
        Element<E> e = new Element<E>(v);
        e.next = this.next;
        this.next = e;
    }
    public boolean contains(Object o) {
        return this.value.equals(o) || this.next.contains(this, o);
    }
    public boolean equals(Object o) {
        return o instanceof Element && this.value.equals(((Element<?>) o).value);
    }
    public Element<E> getNext() {
        return this.next;
    }
    public E getValue() {
        return this.value;
    }
    public int hashCode() {
        return 3 * this.value.hashCode();
    }
    private boolean contains(Element<E> origin, Object o) {
        return!this.equals(origin) && (this.value.equals(o) || this.next.contains(origin, o));
    }
}
```

Wie lautet die Ausgabe dieses Programms bei Ausführung der *main* Methode und warum? Welche Fehler sind in diesem Programm enthalten und wie kann man sie mit Änderungen an genau zwei Zeilen beheben (die *main* Methode soll dabei unverändert bleiben)? Wie lautet die Ausgabe der *main* Methode im korrigierten Programm?

**Lösungsvorschlag**

Die Ausgabe ist:

*false*
*This must not happen*!
$[1, 2, 3, 0]$

Der Grund für die unerwartete erste Ausgabe ist der erste Aufruf von *equals* in der privaten *contains* Methode. Dieser Aufruf muss durch einen direkten Referenzvergleich

ersetzt werden (also statt $!this.equals(origin)$ muss es $this != origin$ heißen), da hier überprüft werden muss, ob die gleiche Referenz wie zu Beginn der Listentraversierung erreicht wurde. Sollten vorzeitig bereits Elemente in der Liste erreicht werden, die bzgl. *equals* gleich sind, bricht die Suche zu früh ab. Die zweite Ausgabe entsteht, weil die Methode *hashCode* in der inneren Klasse *Parity* inkonsistent zur entsprechenden *equals* Methode implementiert ist. Falls zwei Objekte bzgl. *equals* gleich sein sollten, müssen sie dies auch bzgl. *hashCode* sein. Dies ist hier nicht der Fall, wodurch das *Set* in seiner *contains* Methode nicht erkennt, dass es bereits ein Element beinhaltet, das gleich dem Objekt $p$ ist. Aus dem gleichen Grund werden die Elemente 3 und 0 der Menge hinzugefügt, obwohl bereits Elemente im *Set* enthalten sind, die gleich 3 und 0 sein sollten. Das Element 6 wird hingegen nicht mehr hinzugefügt, da hier sowohl die *hashCode* Methode als auch die *equals* Methode die Gleichheit von 6 und 0 erkennen. Um den zweiten Fehler zu verbessern, muss also der Rückgabewert der *hashCode* Methode für die *Parity* Klasse zu $this.value\%2$ korrigiert werden. Dann lautet die Ausgabe des Programms:

*true*
$[1, 2]$

**Aufgabe H3 (10 Punkte)**[2]

Auf der Webseite der Vorlesung ist eine Implementierung von $k$-Medians für Bilder hinterlegt sowie eine Sammlung von Verkehrsschildern.

Implementieren Sie eine Distanzfunktion für Bilder—insbesondere die mitgelieferten Verkehrsschilder—welche ihre Ähnlichkeit möglichst gut widerspiegelt. Implementieren Sie dazu eine Klasse *ImageMetric* welche das Interface $lufti.kmedians.Metric\langle BufferedImage\rangle$ implementiert. Die Klasse *java.awt.image.BufferedImage* wird von Java bereitgestellt, um mit Bitmaps zu arbeiten. Wir empfehlen, die Methoden *getWidth*, *getHeight* und *getRGB* dieser Klassen nachzuschlagen. Zur Bearbeitung von Farbwerten bietet sich die Klasse *java.awt.Color* an.

So können Sie z.B. den Blauanteil des Pixels mit den Koordinaten $(17, 34)$ feststellen:

```
import java.awt.Color;
import java.awt.image.BufferedImage;
...
int rgb = image.getRGB(17, 34);
Color col = new Color(rgb);
int blauAnteil = col.getBlue();
```

Erläutern Sie kurz, warum Sie diese Distanzfunktion gewählt haben.

Verwenden Sie den Klassifikator aus der Tutorübung, um die vorgegebenen Verkehrszeichen in 2, 3, 4 oder 5 Mengen zu klassifizieren.

Was erhalten Sie als Ergebnis? Sind Sie damit zufrieden?

**Lösungsvorschlag**

Eine mögliche Lösung findet sich im Code auf der Homepage, dort werden zwei Metriken miteinander multipliziert: die erste Metrik vergleicht schlicht, wie viele Pixel an der gleichen Stelle der zwei Bilder identisch sind (sind die Bilder identisch, so ist diese Metrik

null, sind alle Pixel verschieden so ist sie eins). Diese Metrik soll (grob) die Form der Schilder erkennen. Die zweite Metrik vergleicht die Farben der Bilder, dazu bestimmt sie den prozentualen Rot-, Grün und Blauanteil (gewichtet mit den Transparentwerten) in einem Vektor $(r, g, b)$. Die Metrik bestimmt dann die euklidische Distanz dieser Farbvektoren. Diese sehr einfach Variante funktioniert schon erstaunlich gut zum Sortieren der Verkehrsschilder.

## Abgabe zum 17.12.2013

[2]Bitte Quelltext für die Abgabe ausdrucken und zusätzlich per E-mail an den jeweiligen Tutor senden.