Multimedia Retrieval

7 Efficient Algorithms and Data Structures

Prof. Dr. Marcin Grzegorzek

Research Group for Pattern Recognition
www.pr.informatik.uni-siegen.de

Institute for Vision and Graphics
University of Siegen, Germany

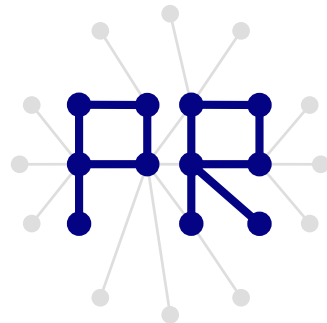


Table of Contents

Table of Contents

Table of Contents

Table of Contents

Overview

1. Hochdimensionale Indexstrukturen
2. Algorithmen zur Aggregation von Ähnlichkeitswerten

Einführung

- Algorithmen und Datenstrukturen für effiziente Ergebnisberechnung bzgl. einer Anfrage
- Erweiterung von Verfahren klassischer DBS um Behandlung von Ähnlichkeitswerten bzw. Unähnlichkeitswerten
  $\longrightarrow$ Übergang von Mengensemantik zur Listensemantik
  - hochdimensionale Indexstrukturen: zur effizienten Suche im hochdimensionalen Raum
  - Aggregation von Ähnlichkeitswerten: erforderlich etwa bei komplexen Anfragen

Overview

Allgemeines

- Strukturierung der Daten zur Unterstützung einer effizienten Suche
- klassische Datenstruktur in DBS: B-Baum und dessen Varianten
  - exakte Suche mit logarithmischem Aufwand
  - aber Einschränkung auf eine Dimension
    $\longrightarrow$ ungeeignet zur Ähnlichkeitssuche im hochdimensionalen Raum

Allgemeines

Anforderungen I

- Korrektheit und Vollständigkeit
- skalierbar bzgl. Dimensionsanzahl
- räumliche Ausdehnung der Objekte:
  - 0 Dimensionen: Punkt
  - 1 Dimension: Linie
  - 2 Dimensionen: Fläche
  - n Dimensionen: etwa Hyperwürfel

Anforderungen II

- Sucheffizienz, also Anzahl Seitenzugriffe muss besser als bei sequentiellem Durchlauf sein
- viele Anfragearten
- effiziente Update-Operationen
- verschiedene Distanzfunktionen
- speicherplatzsparend

Anfragearten

- Nächste-Nachbarsuche
- Approximative Nächste-Nachbarsuche
- Reverse-Nächste-Nachbarsuche
- Bereichssuche
- Punktsuche
- Partial-Match-Suche
- Ähnlichkeitsverbund

Anfragearten

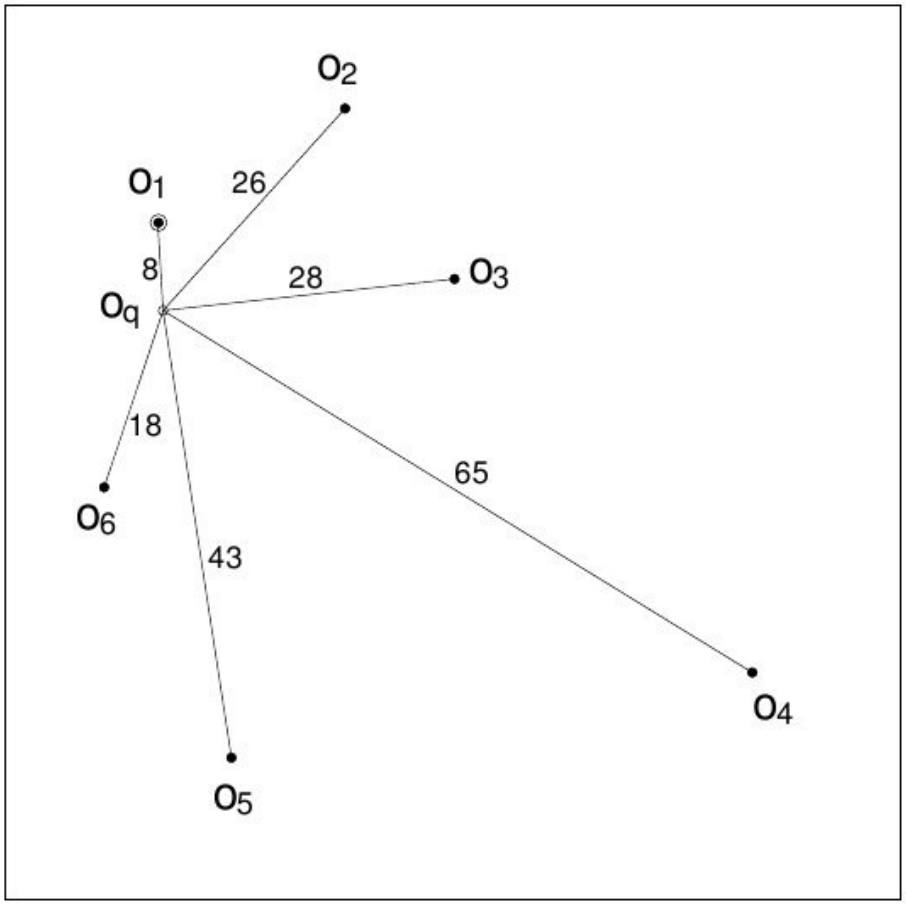
Nächste-Nachbarsuche

- Feature-Daten eines Anfragemedienobjekts: o_q
- Menge von Feature-Daten: FO
- binäre Distanzfunktion $d()$
- Finden des ähnlichsten Medienobjekts (das nächste Feature-Objekt)
- mehrere nächste Nachbarn möglich:

$$nn(o_q) \subseteq FO \text{ mit } \forall o_i \in FO : \forall o \in nn(o_q) : d(o_q, o) \leq d(o_q, o_i)$$

Anfragearten

Nächste-Nachbarsuche graphisch



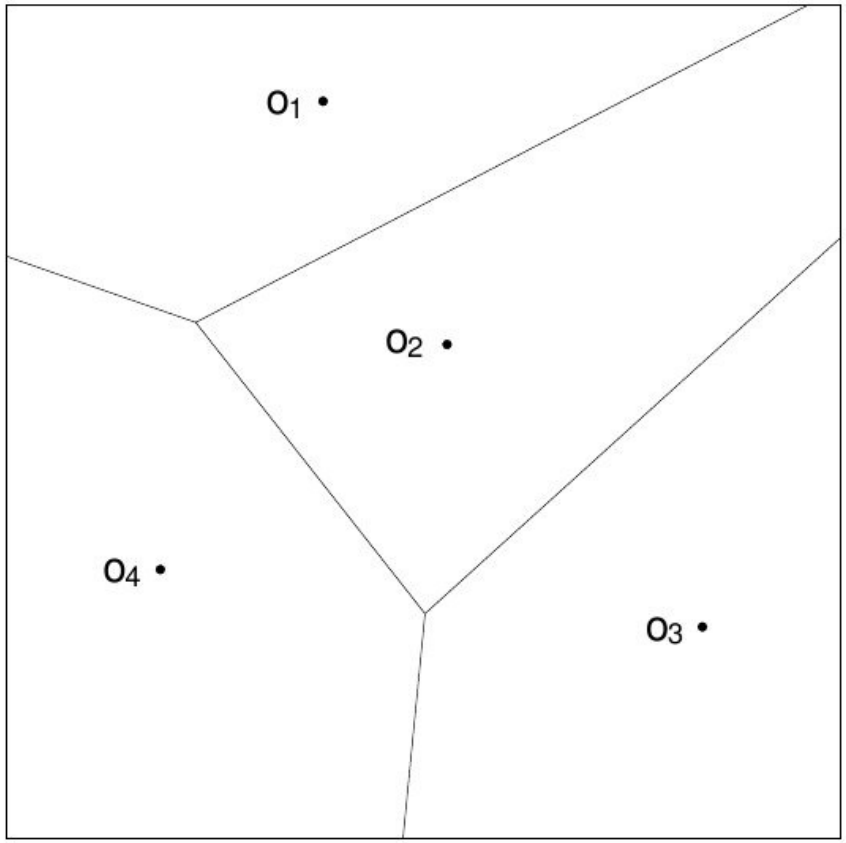
Anfragearten

Zweidimensionale Voronoi-Zellen

- NN-Suche auf punktförmigen Feature-Daten ist äquivalent zum Enthaltensseinstest in Voronoi-Zelle
- jedem Feature-Objekt ist eigene Voronoi-Zelle zugewiesen
- Voronoi-Zelle enthält alle Raumpunkte, die nächste Nachbarn des entspr. Feature-Objekts sind
- Idee: Vorausberechnung aller Voronoi-Zellen und anschließend Enthaltensseintest
  Problem: hohe Berechnungskomplexität für Enthaltensseinstest

Anfragearten

Voronoi-Zellen graphisch



Anfragearten

KNN-Suche

- die k nächsten Nachbarn werden gesucht

$$knn(o_q) \subseteq FO \text{ mit } |knn(o_q)| = k \text{ und } \forall o \in FO \setminus knn(o_q) :$$
$$\forall o_{knn} \in knn(o_q) : d(o_q, o) \geq d(o_q, o_{knn})$$

- bei gleichen Distanzen: nichtdeterministische Auswahl
- Ergebnisobjekte werden aufsteigend nach Distanz sortiert
- k ist üblicherweise so klein, das Ergebnis in Hauptspeicher passt
- ansonsten: Ergebnisobjekt sukzessive abholen (`getNext`-Semantik/`ranking`-Anfrage)

Anfragearten

Approximative Nächste-Nachbarsuche

- Effizienzgewinn bzgl. NN-Anfragen, wenn kleine Ungenauigkeiten tolerierbar
- ϵ als Maß der Ungenauigkeit

$$ann(o_q) = o \in FO \text{ wenn } \; d(o_q, o) \leq (1 + \epsilon) \cdot d(o_q, nn(o_q))$$

- mehrere Feature-Objekte können Bedingung erfüllen $\longrightarrow$ nichtdeterministische Auswahl
- Vorsicht: *ann*-Ergebnis muss nicht in Nähe des *nn*-Ergebnisses liegen

Anfragearten

PAC-NN-Suche

(probably approximative correct)

- weitere Abschwächung einer ANN-Suche
- Forderung: Wahrscheinlichkeit der Abweichung von *ann*-Bedingung muss vorgebbaren Mindestwert überschreiten
- ermöglicht effizientere Suche

Anfragearten

Reverse-Nächste-Nachbarsuche

- Suche nach Feature-Objekten, deren nächster Nachbar der Anfragepunkt ist
  (etwa Suche nach bestem Ort für neuen Einkaufsmarkt)

$$rnn(o_q) = \{o \in FO | o_q \in nn(o)\}$$

- Achtung: Ergebnis oft anderes als bei NN-Suche, da Nächste-Nachbarrelation nicht symmetrisch ist

Anfragearten

Bereichssuche

- Anfrage definiert einen Bereich (Region) im hochdimensionalen Raum
- Ergebnis sind alle Feature-Objekte, die Anfragebereich schneiden

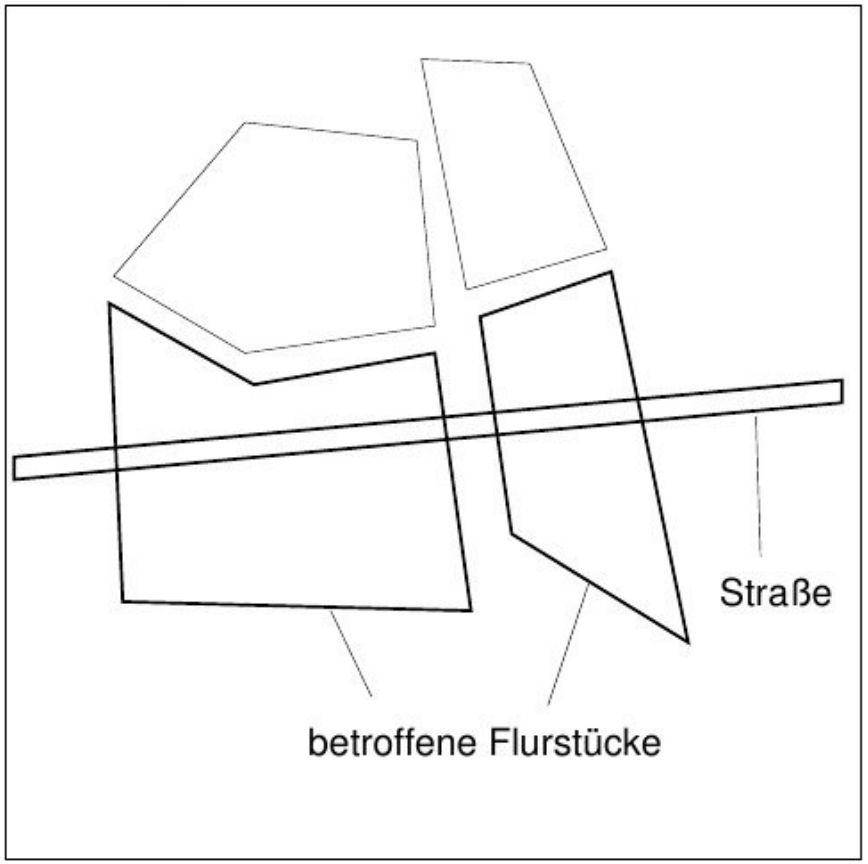
$$range(o_q) = \{o \in FO | o \cap o_q \neq \emptyset\}$$

- Varianten
  - begrenzte versus unbegrenzte Bereiche
  - Spezialfall Hyperkugel und Hyperrechteck

Anfragearten

Bereichssuche graphisch

Beispiel: Straßenplanung im Katasteramt:



Anfragearten

Punktsuche

- Suche anhand eines gegebenen Feature-Objekts o_q
- Test auf Enthaltensein (exakte Überdeckung)

$$punkt(o_q) = \begin{cases} \texttt{wahr} & : \exists o \in FO : o_q = o \\ \texttt{falsch} & : \text{sonst} \end{cases}$$

- Punktsuche in MMDB ist relativ selten

Anfragearten

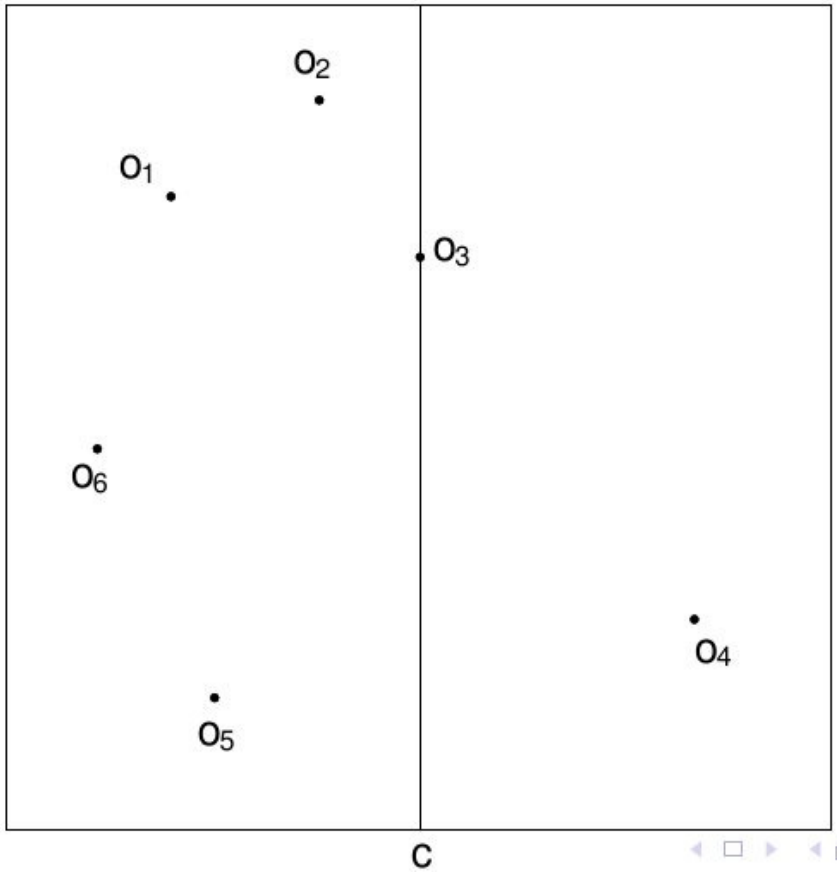
Partial-Match-Suche

- Punktsuche kann als Complete-Match-Suche aufgefasst werden
- bei Partial-Match-Suche Übereinstimmung nur in einigen Dimensionen
  (restliche Dimensionen werden ignoriert)
- ist Spezialfall der Bereichsanfrage mit teilweise unbegrenztem Bereich

Anfragearten

Partial-Match-Suche graphisch

Suchbereich ist senkrechte Linie (Übereinstimmung in nur einer Dimension)



Anfragearten

Ähnlichkeitsverbund

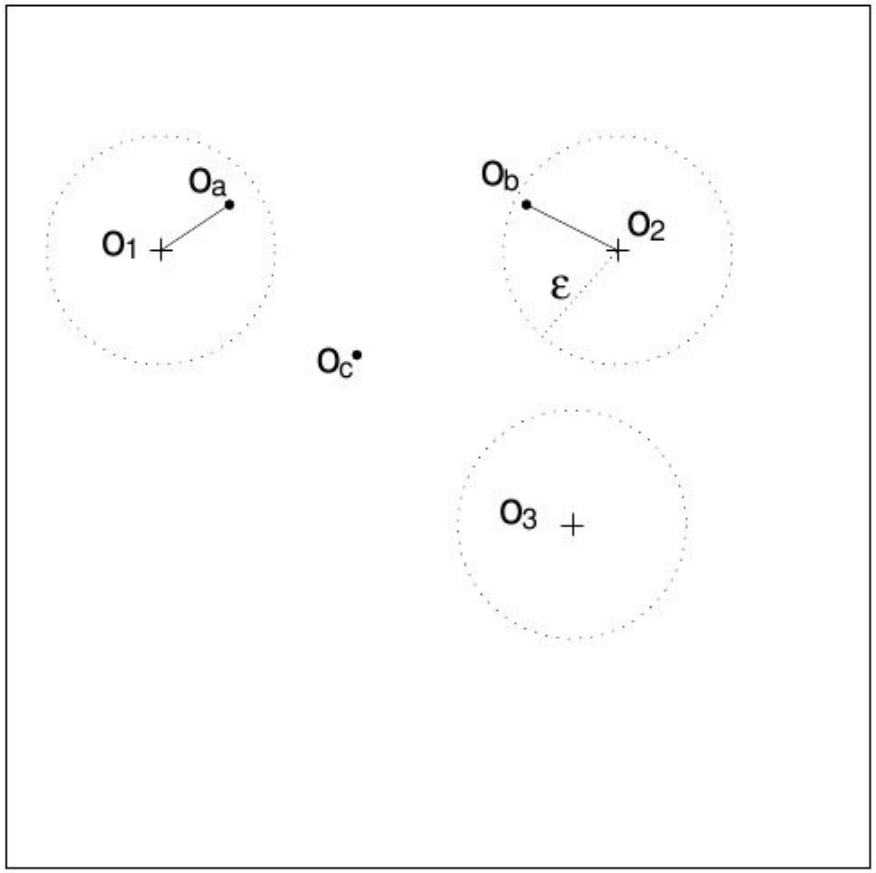
- Operation auf zwei Mengen von Feature-Objekten
- Verbund findet Paare, deren Distanz kleiner als vorgebener Schwellenwert ϵ ist

$$sj(FO_1, FO_2, \epsilon) = \{(o_1, o_2) | o_1 \in FO_1 \wedge o_2 \in FO_2 \wedge d(o_1, o_2) \leq \epsilon\}$$

- Selbstverbund: dieselbe Menge zweimal

Anfragearten

Ähnlichkeitsverbund graphisch



Baumverfahren

- Ausgangspunkt: punktförmige hochdimensionale Feature-Objekte
- B-Baum ist eindimensional
- Abbildung mehrdimensionaler Raum auf eine Dimension im Allgemeinen nicht distanzerhaltend möglich
  (siehe etwa Simplex mit $n + 1$ Punkten im n-dimensionalen Raum)
- Fazit: mehrdimensionale Indexverfahren sind erforderlich

Baumverfahren

Grundidee hierarchischer Indexierungsverfahren:

- Beschreiben von Punktmengen durch geometrische, umschreibende Regionen (Cluster)
- bei der Suche Test und evtl. Ausschluss von Regionen
- Regionen können sich gegenseitig enthalten
  $\longrightarrow$ Halbordnung und daher Hierarchie (Hasse-Diagramm) oder Baum

Unterscheidungskriterien von Baumverfahren

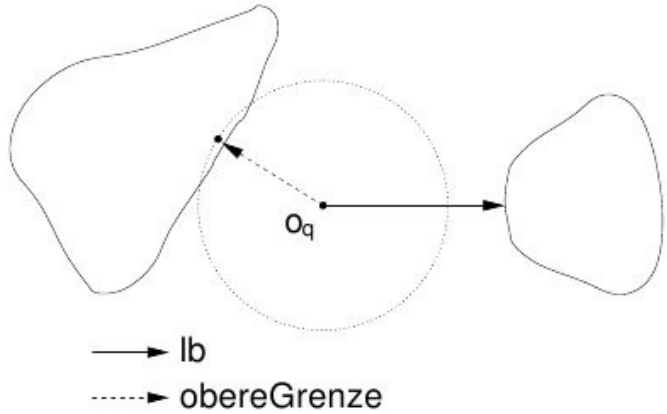
Merkmal	**Unterscheidung**
Cluster-Bildung	global zerlegend (space partitioning) lokal gruppierend (data partitioning)
Cluster-Überlappung	überlappend disjunkt
Balance	balanciert unbalanciert
Objektspeicherung	Blätter und Knoten Blätter
Geometrie	Hyperkugel Hyperquader Hyperellipsoid

Suchalgorithmen in Bäumen

- Algorithmen zur Berechnung des nächsten Nachbars
- Anfragepunkt q
- Existenz zweier Distanzfunktionen
  - Distanz zwischen zwei Punkten
  - minimale Distanz zwischen q und potenziellem Clusterpunkt eines Clusters

Branch-und-Bound-Algorithmus

- 1988 publiziert von Niemann und Goppert
- geht von Objektspeicherung in Blättern aus
- realisiert Tiefensuche
- verwendet dynamisch angepasste Distanz `obereGrenze` zu NN-Kandidat



Branch-und-Bound-Algorithmus


```
[1]  real obereGrenze = ∞
[2]  punkt naechsterNachbar = nil
[3]
[4]  procedure BranchAndBound(punkt q,knoten T)
[5]
[6]    sortiere Subknoten von T aufsteigend nach lb-Distanz zu q
[7]    for each Subknoten k von T do
[8]      if k ist Blatt then
[9]        for each Punkt p in k do
[10]         distanz = d(p,q) //Distanz zwischen p und q
[11]         if distanz < obereGrenze then do
[12]           obereGrenze = distanz
[13]           naechsterNachbar = p // NN-Kandidat
[14]         end if
[15]       end for
[16]     else do
[17]       lb=lb(q,k) //kleinstmögliche Distanz von q zu k
[18]       if lb > obereGrenze then
[19]           schließe k von allen weiteren Betrachtungen aus
[20]       else BranchAndBound(q,k)
[21]     end else
[22]   end for
[23] end procedure
```

RKV-Algorithmus

Roussopoulos/Kelly/Vincent 1995

- spezieller Branch-and-Bound-Algorithmus zur Nächste-Nachbarsuche
- Feature-Objekte als hochdimensionale Objekte
- für Baum mit lokal gruppierenden Hyperquadern und Feature-Objekten in den Blättern
- Hyperquader als MBR (engl. minimum bounding rectangle):
  - jede Hyperfläche berührt mind. ein Feature-Objekt von außen
  - MBR wird durch zwei Punkte (s, t) (innerste und äußerste Ecke) identifiziert

RKV-Algorithmus

MIN-Distanz

minimal mögliche Distanz zwischen Anfragepunkt q und MBR (s, t)

$$MINDIST(q, (s, t)) = \sum_{i=1}^{n} |q[i] - r[i]|^2$$

mit

$$r[i] = \begin{cases} s[i] & \text{wenn } q[i] < s[i] \\ t[i] & \text{wenn } q[i] > t[i] \\ q[i] & \text{sonst.} \end{cases}$$

RKV-Algorithmus

MINMAX-Distanz

maximal möglicher Abstand zum nächsten Nachbarn in (s, t)
(Ausnutzen der Minimaleigenschaft eines MBR)

$$MINMAXDIST(q,(s,t)) = \min_{1 \le k \le n} \left(|q[k] - rm[k]|^2 + \sum_{\substack{i \neq k \\ 1 \le i \le n}} |q[i] - rM[i]|^2 \right)$$

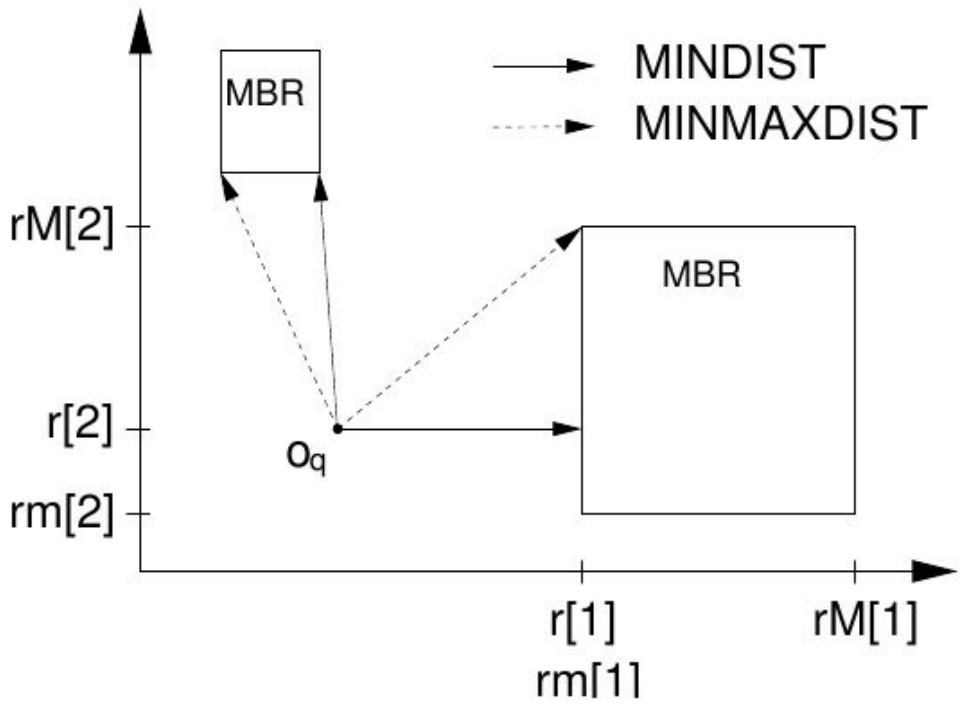
mit

$$rm[k] = \begin{cases} s[k] & \text{wenn } q[k] \le \frac{(s[k]+t[k])}{2} \\ t[k] & \text{sonst} \end{cases} \quad \text{und}$$

$$rM[i] = \begin{cases} s[i] & \text{wenn } q[i] \ge \frac{(s[i]+t[i])}{2} \\ t[i] & \text{sonst} \end{cases}$$

RKV-Algorithmus

MIN- und MINMAX-Distanzen graphisch



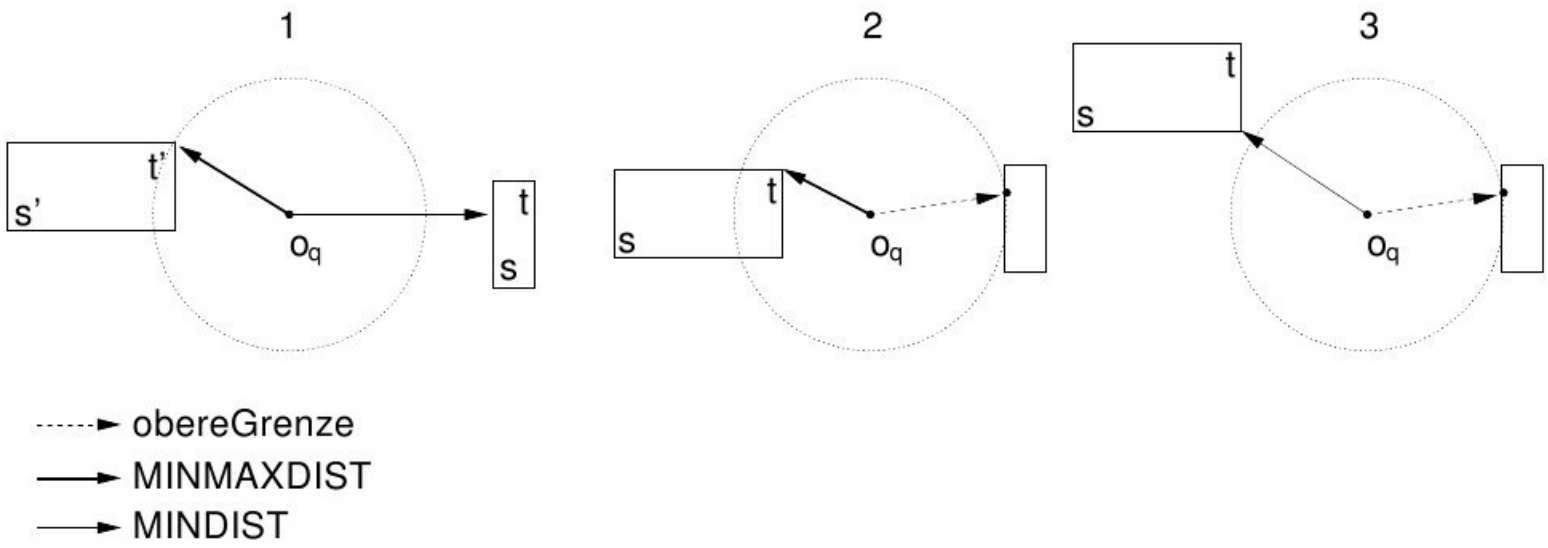
RKV-Algorithmus

Reduzierung des Suchaufwands

- Sortierung der Kindsknoten anhand MIN-Distanz (optimistisch) oder MINMAX-Distanz (pessimistisch)
- 3 Strategien zur Suchaufwandreduzierung (*obereGrenze* ist Distanz zum NN-Kandidaten)
  1. $MINDIST(q,(s,t)) > MINMAXDIST(q,(s',t'))$:
     $\longrightarrow$ MBR (s,t) braucht nicht aufgesucht zu werden
  2. $obereGrenze > MINMAXDIST(q,(s,t))$:
     $\longrightarrow$ $obereGrenze = MINMAXDIST(q,(s,t))$
  3. $obereGrenze < MINDIST(q,(s,t))$:
     $\longrightarrow$ MBR (s,t) braucht nicht aufgesucht zu werden

RKV-Algorithmus

Reduzierung des Suchaufwands graphisch



RKV-Algorithmus


```
[1]  procedure RKV(punkt q,knoten T,real obereGrenze,
[2]                objekt naechsterNachbar)
[3]       knoten neuerKnoten
[4]       brancharray branchList
[5]       real distanz
[6]       objekt o
[7]       if T ist Blattknoten then
[8]         for each o in T do
[9]           distanz ist Distanz zwischen q und o
[10]          if distanz < obereGrenze then do
[11]            obereGrenze = distanz
[12]            naechsterNachbar = o
[13]          end if
[14]        end for
[15]      else do
[16]        branchList sei Liste von Kindknoten aus T
[17]        branchList nach MINDIST oder MINMAXDIST sortieren
[18]        branchList nach Strategie 1 kürzen
[19]        obereGrenze nach Strategie 2 reduzieren
[20]        branchList nach Strategie 3 kürzen
[21]        for each neuerKnoten in branchList do
[22]          RKV(q,neuerKnoten,obereGrenze,naechsterNachbar)
[23]          branchList nach Strategie 3 kürzen
[24]        end for
[25]      end do
[26] end procedure
```

RKV-Algorithmus

knn-Anfragen mit RKV-Algorithmus

Modifikation des Algorithmus:

- sortierte Warteschlange zur Verwaltung der k Nächste-Nachbarnkandidaten
- *obereGrenze* ist Distanz zum letzten Kandidat

RKV-Algorithmus

Bewertung RKV-Algorithmus

- `getNext`-Anfragen werden nicht unterstützt (Problem aufgrund der Tiefensuche)
- ursprünglich für euklidsche Distanz entwickelt; funktioniert auch auf anderen Distanzfunktionen, so lange MIN- und MINMAX-Distanzen berechnet werden könnnen
- Einschränkung auf MBRs als Clustergeometrien
  $\longrightarrow$ ansonsten statt MINMAX- die MAX-Distanz verwenden

HS-Algorithmus

Henrich/Hjaltason/Samet

- Algorithmus für `getNext`-Anfragen
- statt Tiefensuche Verwendung einer global sortierten Warteschlange:
  - enthält Knoten, Blätter und Feature-Objekte mit minimalen Distanzen `lb` zum Anfragepunkt
  - legt Abarbeitungsreihenfolge aufgrund aufsteigender Distanz fest
    $\longrightarrow$ nur Kopfelemente werden entnommen

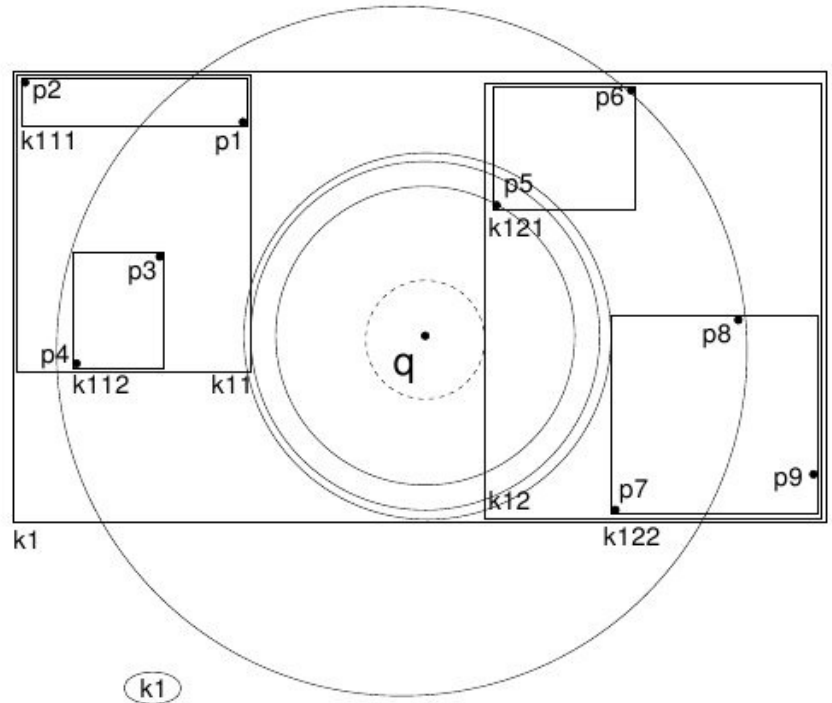
HS-Algorithmus

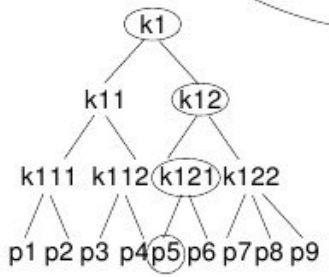
- Initialisierung mit Wurzelelement
- wenn entnommenes Kopfelement Knoten, dann werden dessen Kinder eingefügt
- wenn entnommenes Kopfelement Blatt, dann werden dessen Feature-Objekte eingefügt
- wenn entnommenes Kopfelement Feature-Objekt $\longrightarrow$ Ausgabe

HS-Algorithmus


```
[1]  procedure HS(punkt q,knoten T)
[2]      pqueue queue // Priority Queue
[3]      queueEintrag element // Priority-Queue-Eintrag
[4]      objekt fo // Feature-Objekt
[5]      knoten k
[6]      enqueue(queue, T, lb(q,T))
[7]      while not isEmpty(queue) do
[8]        element = dequeue(queue)
[9]        if element ist Feature-Objekt then do
[10]         while element = first(queue) do
[11]           deleteFirst(queue) // Duplikate entfernen
[12]         end do
[13]         ausgabe(element) // getNext-Resultat ausgeben
[14]       end do
[15]       else if element ist Blattknoten then
[16]         for each fo in element do
[17]           enqueue(queue, fo, lb(q,fo))
[18]         end do
[19]       else // innerer Knoten
[20]         for each Kind k in element do
[21]           enqueue(queue, k, lb(q,k))
[22]         end do
[23]       end if
[24]     end do
[25] end procedure
```

HS-Algorithmus





Queue: k1
k12 k11
k121 k11 k122
p5 k11 k122 p6

HS-Algorithmus

Bewertung

- gut geeignet für `getNext`-Anfragen
- unabhängig von Cluster-Geometrie
- Problem: Warteschlange kann zu groß für Hauptspeicher werden
  $\longrightarrow$ aufwändige Auslagerung auf Festplatte notwendig
- Navigation „springt“ aufgrund Warteschlangensortierung keine Tiefen/Breitensuche
  $\longrightarrow$ teure Festplattenzugriffe

Overview

Aggregation - Einführendes Beispiel

- Gesucht sind alle Bilder, die zu einem vorgegebenen Photo bezüglich Farbverteilung und Textur ähnlich sind. Für jedes Ergebnisbild müssen also zwei Ähnlichkeitswerte anhand einer geeigneten Aggregatfunktion kombiniert werden.

- Gesucht sind alle Bilder, die zu mehreren Anfragebildern ähnlich sind. In diesem Fall müssen pro Ergebnisbild mehrere Ähnlichkeitswerte aggregiert werden.

Anforderungen

An eine Aggregation agg, die Ähnlichkeitswerte für ein Objekt aggregiert, werden bestimmte Forderungen gestellt:

1. *Ähnlichkeitswerte*. Die Funktion muss mehrere Ähnlichkeitswerte aus dem Intervall $[0, 1]$ auf einen Wert aus dem Intervall $[0, 1]$ abbilden:

$$\mathrm{agg} : [0, 1]^n \rightarrow [0, 1]$$

2. *Monotonie.* Wenn die Eingangswerte nicht sinken, dann sinkt auch der aggregierte Ähnlichkeitswert nicht:

$$x_1 \leq y_1 \wedge \ldots \wedge x_n \leq y_n \Rightarrow \mathrm{agg}(x_1, \ldots, x_n) \leq \mathrm{agg}(y_1, \ldots, y_n)$$

Anforderungen

3. *Strikte Monotonie*. Wenn alle Eingangswerte wachsen, dann muss auch der entsprechende, aggregierte Ähnlichkeitswert wachsen:

   $$x_1 < y_1 \wedge \ldots \wedge x_n < y_n \Rightarrow \mathrm{agg}(x_1, \ldots, x_n) < \mathrm{agg}(y_1, \ldots, y_n)$$

4. *Stetigkeit*. Die Aggregatfunktion soll bezüglich der Eingangswerte stetig sein, also keine abrupten Sprünge aufweisen.

Anforderungen

5. *Idempotenz*. Eine Aggregation derselben Werte muss diesen Wert selbst ergeben:

$$\mathrm{agg}(a, \ldots, a) = a$$

6. *Unabhängigkeit von der Reihenfolge*. Das Resultat einer Aggregation ist unabhängig von der Reihenfolge der zu aggregierenden Ähnlichkeitswerte:

$$\mathrm{agg}(x_1, x_2, \ldots, x_n) = \mathrm{agg}(x_{p_1}, x_{p_2}, \ldots, x_{p_n})$$

wobei $[p_i]$ eine beliebige Permutation der Werte $[i]$ darstellt.

Generalisiertes Mittel

- Das generalisierte Mittel ist definiert wie folgt:

$$\mathrm{agg}^{\alpha}_{gm}(x_1, \ldots, x_n) = \left(\frac{x_1^{\alpha} + \ldots + x_n^{\alpha}}{n} \right)^{\frac{1}{\alpha}}$$

- Der Parameterwert α muss ungleich 0 sein.

- Folgende Spezialfälle ergeben sich:
  - $\alpha = 1$: arithmetisches Mittel
  - $\alpha = \infty$: maximaler Ähnlichkeitswert
  - $\alpha = -\infty$: minimaler Ähnlichkeitswert

Klassifikation

1. *Combiner-Algorithmen*: Ausgangspunkt dieser Algorithmen sind mehrere, nach Ähnlichkeitswerten absteigend sortierte Objektlisten, die anhand einer Aggregatfunktion zu einer solchen sortierten Liste vereinigt werden sollen.

2. *Kondensator-Algorithmen*: Ausgangspunkt für diese Algorithmen ist eine Liste von Objekten, bei der mehrere Listenobjekte zu jeweils einem neuen Listenobjekt aggregiert werden.

3. *Indexaggregation*: Bei diesen Algorithmen existieren keine Eingangslisten. Statt dessen wird innerhalb einer Indexstruktur die Aggregation ausgeführt, bevor eine sortierte Liste erstellt wird.

Combiner-Algorithmen

- Jeder Listeneintrag `e` enthält einen eindeutigen Identifikator `e.id` eines Medienobjekts zusammen mit einem Ähnlichkeitswerte `e.grade`.

- Wir beschränken uns der Einfachheit halber auf zwei Eingangslisten: `links` und `rechts`. Ein Objekt `o` hat einen Eintrag `o.lgrade` für den Ähnlichkeitswert der linken Liste und `o.rgrade` der rechten Liste.

- Der aggregierte Ähnlichkeitswert wird mit `o.grade` und der Identifikator mit `o.id` notiert. Einträge, denen noch kein Wert zugewiesen wurde, besitzen den Default-Wert `NULL`.

Combiner-Algorithmen

- Auf die Listenelemente kann in zwei verschiedenen Modi zugegriffen werden. Zum einen ist ein sequentieller Zugriff, also jeweils ein Zugriff auf den Beginn der Liste, möglich. Zusätzlich kann auf ein Medienobjekt über den dazugehörenden Identifikator direkt zugegriffen werden.

- Im Folgenden werden die beiden Zugriffsmodi mit *sequentiellem Zugriff* und mit *randomisiertem Zugriff* bezeichnet. Für die Zugriffe werden die Funktionen `getNext()` und `random()` verwendet.

Combiner-Algorithmen

- Die Ähnlichkeitswerte eines Medienobjekts aus den verschiedenen Listen sollen anhand einer monotonen Aggregatfunktion agg zu einem neuen Ähnlichkeitswert zusammengefasst werden.
- Von allen Ergebnisobjekten soll auf `k` Objekte mit den größten, aggregierten Ähnlichkeitswerten effizient zugegriffen werden können.
- Alternativ wird of auch ein sequentieller Zugriff auf die Objekte in der absteigenden Reihenfolge der aggregierten Ähnlichkeitswerte verlangt.
- Die beiden Arten der Ergebniszugriffe werden mit `top-k`- und mit `ranking`-Zugriff bezeichnet.

Combiner-Algorithmen

- TA-Algorithmus
- NRA-Algorithmus
- Stream-Combine-Algorithmus

Combiner-Algorithmen: TA-Algorithmus


```
01 procedure TA(liste links,liste rechts,funktion agg,liste top-k)
02   eintrag ol, or
03   real tau
04   repeat
05     ol = getNext(links)
06     or = getNext(rechts)
07     ol.rgrade = random(rechts, ol.id)
08     or.lgrade = random(links, or.id)
09     ol.grade = agg(ol.lgrade, ol.rgrade)
10     or.grade = agg(or.lgrade, or.rgrade)
11     aktualisiere top − k bzgl.  ol und or
12     tau = agg(ol.lgrade, or.rgrade)
13   until |top − k| = k and ∀o ∈ top − k : o.grade ≥ tau
14   sortieretop − k − Elementenacho.grade
15 end procedure
```

Combiner-Algorithmen: TA-Algorithmus

Beispiel

links		**rechts**		**tau**	**top-k**	
id	**lgrade**	**id**	**rgrade**	**tau**	**id**	**lgrade+rgrade**
o3	0,9	o4	0,8	1,7	o4	1,4
o1	0,7	o2	0,7	1,4	o1	1,3
o4	0,6	o1	0,6	**1,2**	–	–
o2	0,2	o5	0,4	0,6	–	–
o5	0,1	o3	0,1	0,2	–	–