2.4.2 *k*-nächste Nachbarn (*k*-NN) Anfragen

– Allgemeines
  - Eigenschaften
    – Benutzer gibt Anfrageobjekt *q* und Anzahl *k* vor
    – Ergebnis enthält die *k* nächsten Nachbarn von *q*
    – Mehrdeutigkeiten müssen wiederum sinnvoll behandelt werden
  - Formal
    – Deterministisch
      kleinste Menge *NN*(*q*,*k*) ⊆ DB mit mindestens *k* Objekten, sodass
      $$\forall o \in NN(q,k), \forall o' \in DB - NN(q,k) : dist(q,o) < dist(q,o')$$
    – Nicht-deterministisch
      Menge *NN*(*q*,*k*) ⊆ DB mit exakt *k* Objekten, sodass
      $$\forall o \in NN(q,k), \forall o' \in DB - NN(q,k) : dist(q,o) \leq dist(q,o')$$
    – Klar: NN(*q*,1) ≡ NN(*q*)

– Basisalgorithmus (sequential scan): nichtdeterministisch

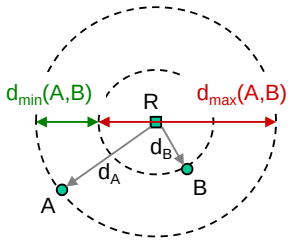
```
NN-SeqScan(DB, q, k)
    result = LIST OF (dist:REAL, p:OBJECT) ORDERED BY dist DESCENDING;
    result = [];
    FOR i=1 TO k DO
        result.insert(dist(q, getObject(i), getObject(i));
    FOR i=k+1 TO n DO
        IF dist(q, DB.getObject(i)) ≤ result.getFirst().dist THEN
            result.deleteFirst();
            result.insert(dist(q, getObject(i), getObject(i));
    RETURN result;
```

  - Bemerkung:
    – Liste result ist hier absteigend sortiert
      » Widerspricht möglicherweise der Anwendersicht (erstes Element = bestes Element = erster NN)
      » ABER: es gibt Datenstrukturen für geordnete Listen, deren Sortierung effizient ist und bei denen man sehr effizient auf das erste Objekt in der Liste zugreifen kann (z.B. Heapsort).

- Algorithmen mit Index
  - Grundsätzlich lassen sich alle Algorithmen zur NN-Suche auf *k*-NN-Suche erweitern, egal ob Index-basiert oder mehrstufig
    - Pruningdistanz ist entsprechend immer die Distanz zum aktuell gefundenen *k*-NN (erstes Element in result-Liste)
    - Beim Algorithmus nach [RKV] kann MINMAXDIST zu einer Seite nur wie Distanz zu einem Punkt gewertet werden und nicht als Gesamt-Pruningdistanz => MINMAXDIST lohnt sich i.A. nicht für *k*-NN-Query
- Algorithmen mit Multi-Step Architektur
  - Alle drei Alternativen leicht erweiterbar
    - Auswertung mit Bereichsanfrage
      - » *k*-NN-Query statt NN-Query im Filter und Refinement anpassen (siehe Übung)
    - Unmittelbare Verfeinerung
      - » erweitere *k*-NN-Algorithmus statt NN-Algorithmus um entsprechende Aufrufe
    - Auswertung nach Priorität
      - » benutze *k*-NN-Distanz als Abbruchkriterium (siehe Übung)

- Generalisierung der R-Optimalen *k*-NN Anfrage
  - Grundsätzlich:
    - » Unterscheidung der Optimalität bzgl.
      - I/O Kosten (Seitenzugriffe im Index) = Kosten im Filterschritt
      - Kosten für die Berechnung der exakten Distanz = Kosten im Verfeinerungsschritt
    - » Optimalität eines mehrstufigen Anfragealgorithmus hängt von der im Filterschritt zur Verfügung stehenden Distanzinformation ab, d.h. mehr Information im Filterschritt
      - $\Rightarrow$ höhere Selektivität des Filters
      - $\Rightarrow$ geringere Verfeinerungskosten
  - Ziel:
    - » Kandidatenmenge im Filterschritt mehr reduzieren unter Berücksichtigung weiterer Filterkriterien
  - Motivation:
    - » exakte Distanzberechnung sehr teuer im Vergleich zur Filterdistanzberechnung
      - $\Rightarrow$ Filter weiter verfeinern durch zusätzliche Filterinformationen
    - » In vielen Applikationen können Ähnlichkeitsdistanzen sowohl nach unten als auch nach oben hin effizient abgeschätzt werden
  - Idee:
    - » zusätzlich zur unteren Distanzabschätzung (= Lower-Bound (LB)) wird obere Distanzabschätzung (= Upper-Bound (UB)) im Filterschritt mit einbezogen
      - $\Rightarrow$ optimale Auswertung mit unterer und oberer Distanzabschätzung

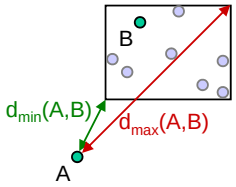
– Beispiele für die Ermittlung von unterer bzw. oberer Distanzabschätzung:
  » Abschätzung basierend auf Referenzpunkten (z.B. M-tree)



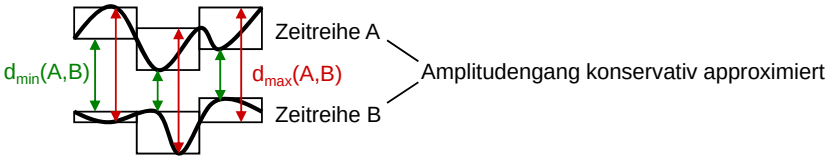
$d_{min}(A,B) = |d_A - d_B|$
$d_{max}(A,B) = d_A + d_B$

$d_{min}(A,B) \leq d(A,B) \leq d_{max}(A,B)$
LB exakt UB

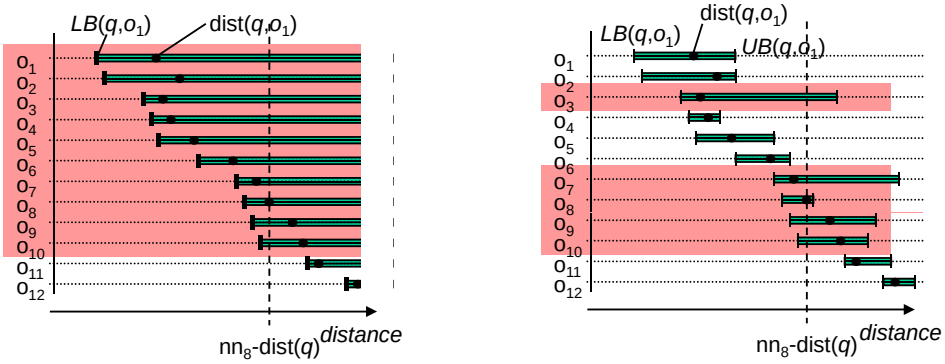
  » Abschätzung basierend auf Regionen (z.B. R-tree)



  » Individuelle Abschätzung (Applikations/Daten abhängig)



• LB-basierte *k*-NN-Suche vs. (LB+UB)-basierte *k*-NN-Suche



– nn_k-dist(q) bezeichne die Distanz des k-nächsten Nachbarn von dem Anfrageobjekt q
– Alle Objekte o deren Filterdistanzen die Eigenschaft
  $LB(q,o) \leq nn_k\text{-}dist(q) \leq UB(q,o)$ erfüllen, müssen verfeinert werden.
– Bei der LB-basierten *k*-NN-Suche müssen mehr Objekte verfeinert werden als bei der (LB+UB)-basierten *k*-NN-Suche
  » (siehe Beispiel oben) LB-basierte *k*-NN-Suche muss 10 Objekte verfeinern, während die (LB+UB)-basierte *k*-NN-Suche nur 5 Objekte verfeinern muss
  » Voraussetzung: Die Berechnung der exakten Distanz für die Resultatmenge ist nicht erforderlich

– Optimale (LB+UB)-basierte *k*-NN-Suche

[Kriegel, Kröger, Kunath, Renz. 10th Int. Symp. on Spatial and Temporal Databases (SSTD'07), 2007]

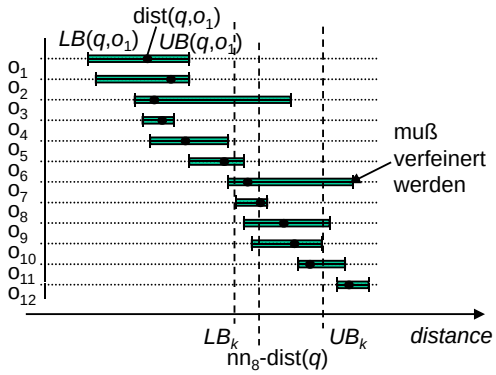
- Optimalität:

  Beweisbar: Algorithmus ist optimal bzgl.
  - » Anzahl der Seitenzugriffe (Filterschritt) und
  - » Anzahl der Verfeinerungen (Verfeinerungsschritt)

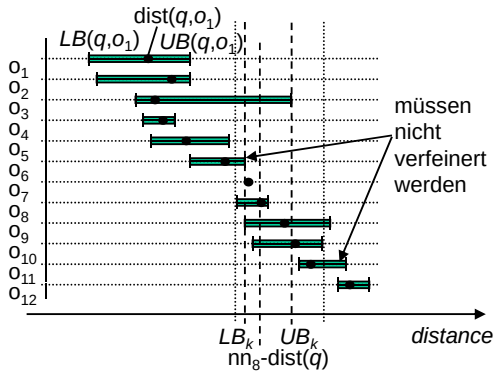
  – Beruht auf dem Prinzip der iterativen Verfeinerung (analog zu „Auswertung nach Priorität“ s. Folie 77):
  - » auf Filterebene läuft „Ranking Query“ ab (siehe Kapitel 2.4.3)
  - » Filtern aufgrund von unterer und oberer Schranke
  - » nach jedem Verfeinern wird erneut gefiltert
  - » Objekt nur dann anfordern, wenn unbedingt notwendig
  - » Objekt nur dann verfeinern, wenn unbedingt notwendig
- Vorteil
  - Einsparungen gegenüber dem Algorithmus „Auswertung nach Priorität“ durch zusätzliche Verwendung der oberen Distanzabschätzung UB(q,o)
- Nachteil
  - Komplexität des Ranking-Algorithmus (Speicher und/oder Zeit) bleibt
  - Kein exaktes „Ranking“ auf den k Ergebnisobjekten

- Prinzip:

vor Verfeinerungsschritt



nach Verfeinerungsschritt



  – konservative Approximation der k-NN-Distanz nn_k-dist(q) durch
  - » LB_k = k kleinste LB-Filterdistanz
  - » Ub_k = k kleinste UB-Filterdistanz
  – Ziel: Nur diejenigen Objekte verfeinern, deren untere und obere Distanzabschätzung die k-NN-Distanz nn_k-dist(q) überdeckt
  – Beweisbar: Es existiert immer mind. ein Kandidat, dessen untere und obere Distanzabschätzung sowohl LB_k als auch UB_k und somit auch die k-NN-Distanz nn_k-dist(q) überdeckt

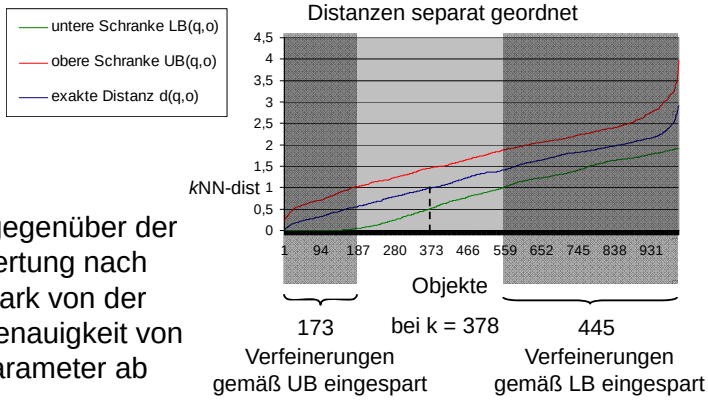
– Algorithmus

k-NN-MultiStep-Optimal(DB, *q*)
Ranking = initialisiere Ranking bzgl. *q* auf LB-Filterdistanz; // Kapitel 2.4.3
result = ∅; *candidates* = ersten *k* Objekte aus dem Ranking; intitialisiere UB_k, LB_k aus *candidates*;
REPEAT
// Schritt 1: hole nächsten Kandidaten
if $LB_{next} \leq LB_k$ then // LB_{next} = LB(*q*,o_{next}), wobei o_{next} = nächstes Obj. im Ranking
p = Ranking.getNext();
füge *p* zu *candidates* hinzu, falls LB(*q*,*p*) ≤ UB_k; // LB_{next} evtl. Distanz zu MBR
endif;
aktualisiere LB_k, UB_k, LB_{next};
// Schritt 2: Filtere true hits und true drops aus (Filter-Schritt)
for each *c* ∈ *candidates* do
if UB(*q*,*c*) ≤ LB_k then nehme *c* aus *candidates* und füge es zu *result* hinzu; // true hit
if LB(q,c) > UB_k then entferne *c* von *candidates*; // true drop
end for;
// Schritt 3: Verfeinere Kandidaten (Verfeinerungs-Schritt)
if |*result*|+|*candidates*| ≤ *k* und $LB_{next} > UB_k$ then
füge alle *c* ∈ *candidates* zu *result* hinzu; // Abbruchbedingung
else
verfeinere alle *c* ∈ candidates, die LB(q,c)≤LB_k≤UB_k≤UB(q,c) erfüllen, d.h.
berechne $d_{exakt}(q,c)$ und setze LB(*q*,*c*) = UB(*q*,*c*) = $d_{exakt}(q,c)$;
end if;
UNTIL(|*candidates*|=0 und $LB_{next} > UB_k$**)**;
RETURN *result*;

– **Optimalität gemäß der Seitenzugriffe:**
In *Schritt 1*: die Bedingung „$LB_{next} \leq LB_k$“ garantiert, dass nur die notwendigen Objekte angefordert werden ⇒ minimaler Seitenzugriff

– **Optimalität gemäß der Anzahl der Verfeinerungen:**
In *Schritt 3* die Bedingung „LB(*q*,*c*) ≤ LB_k ≤ UB_k ≤ UB(*q*,*c*)“ garantiert, dass nur diejenigen Objekte verfeinert werden, die verfeinert werden müssen, d.h. für die gilt: LB(*q*,*c*) ≤ nn_k-dist(*q*) ≤ UB(*q*,*c*) ⇒ minimale Verfeinerungen

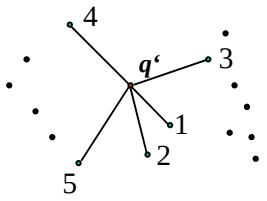
– Experimente auf Realdaten (NN-Suche auf Zeitreihen mit DTW-Distanz)



Bemerkung:
Effizienzgewinn gegenüber der einfachen „Auswertung nach Priorität“ hängt stark von der Approximationsgenauigkeit von UB und dem *k* Parameter ab

2.4.3 Nächste Nachbarn Ranking

– Allgemeines
  - Eigenschaften
    – Benutzer gibt Anfrageobjekt *q* vor und initialisiert damit das Ranking
    – Benutzer kann mehrfach Funktion getNext() aufrufen, die ihm jeweils den 1., 2., usw. Nachbarn von *q* zurück gibt.
    – Mehrdeutigkeiten müssen wiederum sinnvoll behandelt werden
      » Typischerweise nicht-deterministisch: der *k*-te Aufruf ergibt einen der *k*-NN



– Basisalgorithmen (siehe Übung!!!)

– Algorithmus mit Index

[Hjaltason, Samet. Int. Symp. Large Spatial Databases (SSD), 1995]

- Alle *k*-NN-Algorithmen können entsprechend erweitert werden
- Problem der rekursiven Algorithmen
  – Nachdem der i-te Nachbar gefunden ist, wird das Ergebnis an die Ranking-Ausgabe übergeben
  – Weitere getNext()-Aufrufe erfordern erneutes rekursives Suchen
- Vorteil der Prioritätssuche
  – Kompletter Zustand des Algorithmus ist in apl und result gespeichert
- Unterschied zum *k*-NN-Algorithmus
  – Unbeschränkte Ergebnisliste *result* in die jeder Punkt einer geladenen Datenseite eingefügt wird (***aufsteigend*** nach Distanz zu *q* sortiert).
  – Keine Pruningdistanz => Kindseiten verfeinerter Seiten in APL einfügen
  – Algorithmus stoppt (für den aktuellen getNext()-Aufruf) sobald erste Seite in APL größere MINDIST zu *q* hat als bestes Element in *result*
  – Dieses Element wird aus result gelöscht und ausgegeben
  – Nächster getNext()-Aufruf arbeitet mit aktuellen APL und *result* weiter
- Hoher Speicherplatzbedarf: im worst case gesamte DB in *result*

- Algorithmus

```
Globale Variablen:
result = LIST OF (dist:Real, obj:Object) ORDERED BY dist ASCENDING;
apl = LIST OF (dist:Real, da:DiskAdress) ORDERED BY dist ASCENDING

NN-Ranking(pa, q)
    result = [(+∞, dummy)];
    apl = [(0.0, pa)];
    WHILE NOT apl.isEmpty() AND apl.getFirst().dist ≤ result.getFirst().dist DO
        p = apl.getFirst().da.loadPage();
        apl.deleteFirst();
        IF p.isDataPage() THEN
            FOR i=0 TO p.size() DO  // Jedes Objekt einfügen
                result.insert( (dist(q, p.getObject(i)),p.getObject(i)) );

        ELSE                                        // p ist Directoryseite
            FOR i=0 TO p.size() DO  // Jede Seite einfügen
                apl.insert( (MINDIST(q, p.getRegion(i)),p.getChildPage(i)) );
    resultObject = result.getFirst().obj;
    result.deleteFirst();
    RETURN resultObject;
```


– Algorithmen mit Multi-Step Architektur

- Nicht alle Varianten der NN-Algorithmen mit Multi-Step Architektur lassen sich zu Ranking Algorithmen erweitern
  - Auswertung mit Bereichsanfrage
    - » Erweiterung nicht möglich, da kein ε ermittelbar
  - Unmittelbare Verfeinerung
    - » Erweitere einen Ranking Algorithmus
  - Auswertung nach Priorität
    - » Verwalte unbegrenzte Liste von Objekten statt result-Variable

- Ranking-Algorithmus für Multi-Step *k*-NN-Anfragen wichtig, da die Resultate für weitere Filterschritte benötigt werden (bei Auswertung nach Priorität)
  - Ranking im Filterschritt notwendig, da die Ergebnismenge des Filters zunächst unbekannt. Ergebnismenge des Filters wird durch Ergebnise der Verfeinerungen bestimmt!!!