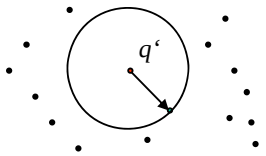
2.4 Nächste Nachbarn Anfragen

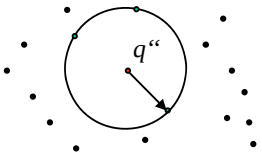
2.4.1 Nächste Nachbarn Anfrage

– Allgemeines
  - Eigenschaften
    – Benutzer gibt Anfrageobjekt *q* vor
    – Ergebnis enthält das Objekt, das die geringste Distanz zu *q* hat
    – Mehrdeutigkeiten müssen sinnvoll behandelt werden (mehrere nächste Nachbarn, oder nichtdeterministisch ein Objekt)
  - Formal

$$NN(q) = \{o \in DB \mid \forall x \in DB : dist(q,o) \leq dist(q,x)\}$$



Eindeutiges Ergebnis



Mehrdeutiges Ergebnis

– Basisalgorithmus (sequential scan): nichtdeterministisch

```
NN-SeqScan(DB, q)
    result = ∅;
    stopdist = +∞;
    FOR i=1 TO n DO
        IF dist(q, DB.getObject(i)) ≤ stopdist THEN
            result := getObject(i);
            stopdist = dist(q, DB.getObject(i));
    RETURN result;
```

– Algorithmus mit Index: Einfache Tiefensuche
  - Unterschied zur Range-Query
    – Nächste Nachbar kann beliebig weit vom Anfragepunkt weg liegen
    – Gestalt der Query zunächst unbekannt
    – Es kann zunächst nicht anhand der Seitenregion entschieden werden, ob eine Seite gebraucht wird
    – Ob eine Seite gebraucht wird, hängt auch von dem Inhalt der anderen Seiten ab

- Kennt man NN-Distanz, würde Range Query ausreichen
- Kennt man einen beliebigen Objekte, kann man dessen Abstand als obere Schranke für die NN-Distanz nutzen
- Kennt man mehrere Objekte, kann man den geringsten Abstand als obere Schranke für die NN-Distanz nutzen

• Umformulierung des RQ-Algorithmus:
- Verwende als ε die kleinste Distanz zu den bisher gefunden Nachbarn

```
Globale Variable: stopdist = +∞;

NN-Index-Simple-TS(pa, q)          // pa = Diskadress z.B. der Wurzel des Indexes
  result = ∅;
  p := pa.loadPage();
  IF p.isDataPage() THEN
    FOR i=0 TO p.size() DO
      IF dist(q, p.getObject(i)) ≤ stopdist THEN
        result := getObject(i);
        stopdist = dist(q, p.getObject(i));
  ELSE                             // p ist Directoryseite
    FOR i=0 TO p.size() DO
      IF MINDIST(q, p.getRegion(i)) ≤ stopdist THEN
        result := NN-Index-Simple-TS(p.childPage(i), q)
  RETURN result;
```

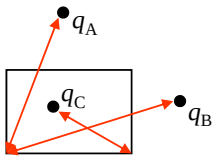

• Nachteil des einfachen Tiefensuch-Algorithmus
- Initialisierung: stopdist = +∞
- Dadurch: Start mit beliebigem Pfad
- Folge: die ersten gefundenen Objekte sind meist sehr weit vom Anfrageobjekt entfernt => stopdist ist wenig selektiv
- Verbesserung: beginne Pfad, der möglichst nah zum Anfrageobjekt liegt

– Algorithmus mit Index: Tiefensuche nach [RKV 95]

[Roussopoulos, Kelley, Vincent. Proc. ACM Int. Conf. Management of Data (SIGMOD), 1995]

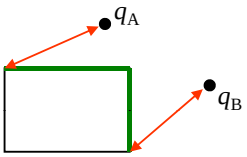
• Vermeidet langsame Einschränkung des Suchraums durch
- Verwendung der Seitenregionen zur Abschätzung der NN-Distanz
- Priorisierung der Tiefensuche nach Distanz der Seitenregion zur Query

• Neben MINDIST weitere Abschätzungen der NN-Distanz durch:
- MAXDIST
  - » Maximale Distanz zwischen Query und allen Punkten der Seitenregion
  - » NN-Distanz kann nicht schlechter als MAXDIST werden



$$\mathrm{MAXDIST}(region, q) = \sqrt{\sum_{0<i\le d} \max\{(q_i - region.UB_i)^2, (q_i - region.LB_i)^2\}}$$

– MINMAXDIST
  » MBRs als Seitenregionen: maximale NN-Distanz noch besser abzuschätzen
  » Auf jeder Kante des MBR muss ein Punkt liegen (sonst ist MBR nicht minimal)
  » Intuition: „nächstliegende Kante, weitester Punkt“



$$MINMAXDIST(region, q) = \sqrt{\min_{0<k\le d}(|q_i - rm_i|^2 + \sum_{\substack{i\neq k \\ 0<i\le d}} |q_i - rM_i|^2)}$$

wobei $$rm_i = \begin{cases} region.LB_i & \text{if } q_i \le \frac{region.LB_i + region.UB_i}{2} \\ regionUB_i & \text{else} \end{cases}$$

$$rM_i = \begin{cases} region.LB_i & \text{if } q_i \ge \frac{region.LB_i + region.UB_i}{2} \\ regionUB_i & \text{else} \end{cases}$$

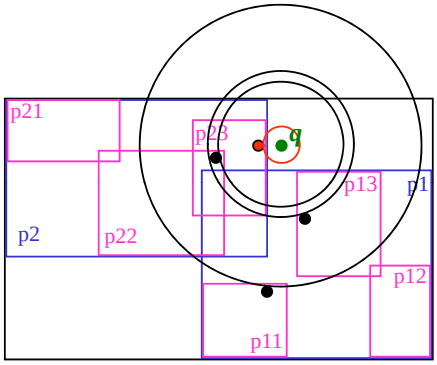
– Für andere Geometrien (nicht MBRs) sind MINDIST und MAXDIST analog definierbar; MINMAXDIST allerdings nicht
– Abschätzung von stopdist durch Minimum aus stopdist und MINMAXDIST (bzw. MAXDIST) aller bisher bekannten Seitenregionen (pruningdist)
– Vor dem rekursiven Abstieg: sortieren der Kindseiten nach MINDIST (experimentell als bestes Prioritätsmaß ermittelt)

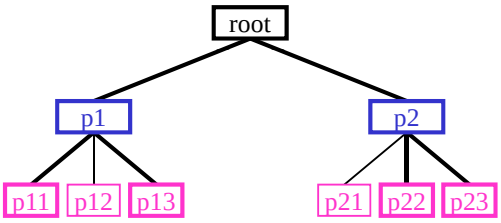
– Algorithmus:

Globale Variablen: stopdist = +∞; pruningdist = +∞;

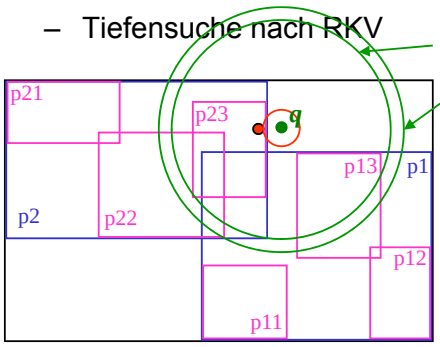
```
NN-Index-RKV-TS(pa, q)          // pa = Diskadress z.B. der Wurzel des Indexes
  result = ∅;
  p := pa.loadPage();
  IF p.isDataPage() THEN
    FOR i=0 TO p.size() DO
      IF dist(q, p.getObject(i)) ≤ stopdist THEN
        result := getObject(i);
        stopdist = dist(q, p.getObject(i));
      IF stopdist < pruningdist THEN
        pruningdist = stopdist;
  ELSE                          // p ist Directoryseite
    FOR i=0 TO p.size() DO
      IF MINMAXDIST(q, p.getRegion(i)) < pruningdist THEN
        pruningdist = MINMAXDIST(q, p.getRegion(i));
    quicksort(p.getObjectArray(), MINDIST);
    FOR i=0 TO p.size() DO
      IF MINDIST(q, p.getRegion(i)) ≤ pruningdist THEN
        result := NN-Index-RKV-TS(p.childPage(i), q);
  RETURN result;
```

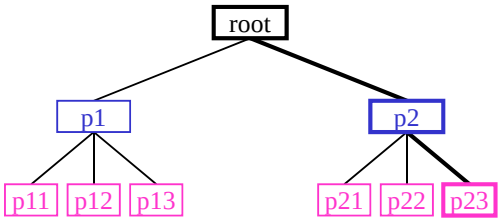
- Ablaufbeispiel
  - Einfache Tiefensuche



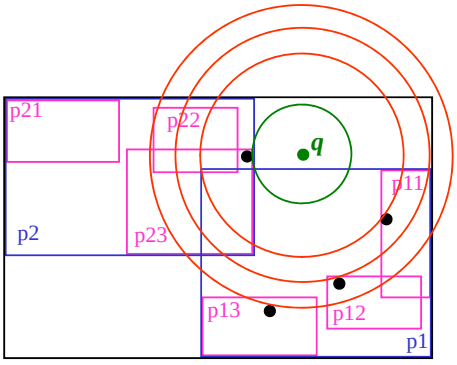


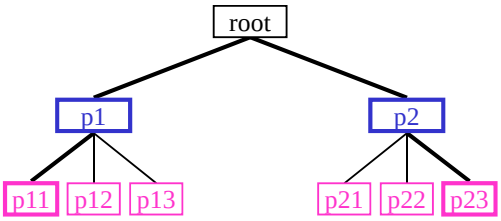
  - Tiefensuche nach RKV





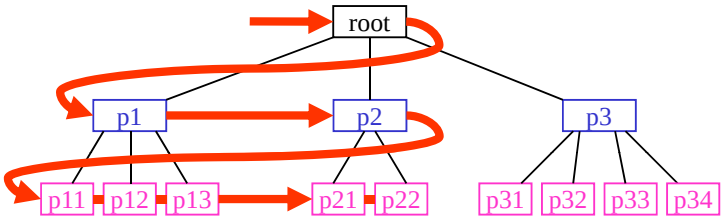
- Fazit:
  - Priorisierung mit MINDIST bewirkt Reduktion der Seitenzugriffe von 7 auf 3
  - MINMAXDIST verbessert Pruning-Distanz, verhindert hier aber keine Seitenzugriffe
  - Trotz Priorisierung: Tiefendurchlauf kann prinzipiell stark fehlgeleitet werden wenn z.B. eine Seite auf dem ersten Level sehr nah am Queryobjekt liegt, ihre Kindseiten aber relativ weit weg





  - Bei Start mit p_2 hätte keine der Kindseiten von p_1 geladen werden müssen

– Algorithmus mit Index: Breitensuche



- Abgesehen von MINMAXDIST-Abschätzung stehen Punktdistanzen erst auf Blatt-Ebene zur Verfügung
  - Viele Zugriffe auf Directoryseiten
  - Die erste Punktdistanz hätte viele dieser Zugriffe schon verhindern können
- Speicherintensiv
  - Worst-case: gesamte letzte Directory-Ebene muss im RAM gehalten werden

– Algorithmus mit Index: Prioritätssuche nach [HS 95]

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

- Statt rekursivem Durchlauf: Liste der aktiven Seiten (active page list APL)
  - Seite p ist aktiv genau dann wenn folgende Bedingungen erfüllt sind:
    - » p wurde noch nicht geladen
    - » Elternseite von p wurde bereits geladen
    - » MINDIST(q, p.getRegion()) ≤ pruningdist
  - APL wird mit Wurzel des Indexes initialisiert
  - Seiten in APL nach MINDIST zum Anfrageobjekt aufsteigend sortiert
  - Algorithmus entnimmt immer die erste Seite aus APL (mit kleinster MINDIST)
  - Entnommene Seite wird geladen und verarbeitet: („verfeinert“)
    - » Datenseiten werden wie bisher verarbeitet
    - » Directoryseiten: Kindseiten mit MINDIST ≤ pruningdist in APL einfügen
  - Ändert sich pruningdist werden Seiten mit MINDIST > pruningdist alternativ:
    - » aus APL entfernt
    - » als gelöscht markiert
    - » ohne explizite Markierung später ignoriert

– Algorithmus:

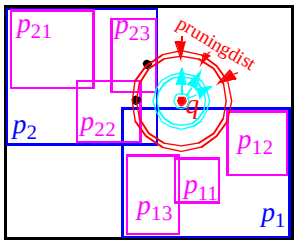
Globale Variablen: stopdist = +∞; pruningdist = +∞;

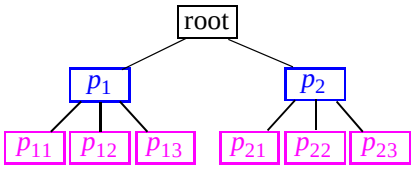
```
NN-Index-HS(pa, q)              // pa = Diskadress z.B. der Wurzel des Indexes
    result = ∅;
    apl = LIST OF (dist:Real, da:DiskAdress) ORDERED BY dist ASCENDING
    apl = [(0.0, pa)]
    WHILE NOT apl.isEmpty() AND apl.first().dist ≤ pruningdist DO
        p := apl.getFirst().da.loadPage();
        apl.deleteFirst();
        IF p.isDataPage() THEN

            (* wie bisher *)

        ELSE                        // p ist Directoryseite
            FOR i=0 TO p.size() DO
                IF MINDIST(q, p.getRegion(i)) ≤ pruningdist THEN
                    apl.insert(MINDIST(q, p.getRegion(i)), p.childPage(i));
    RETURN result;
```


- Beispiel





APL:

1. root
2. p_1 p_2
3. p_2 p_{12} p_{13} p_{11}
4. p_{23} p_{22} p_{12} p_{13} p_{11} p_{21}
5. p_{22} p_{12} p_{13}

- Eigenschaften
  - Allgemein
    - » Seiten werden nach aufsteigendem Abstand geordnet zugegriffen (blaue Kreise)
    - » pruningdist wird kleiner, sobald nähergelegenes Objekt gefunden (rote Kreise)
    - » Anfragebearbeitung stoppt, wenn beide Kreise sich treffen

- Speicherbedarf
  - » Wie bei Breitensuche kann gesamter unterste Directorylevel in APL stehen
  - » Dieser Fall is allerdings unwahrscheinlicher als bei Breitensuche
  - » Speicherkomplexität O(n) (Tiefensuche O(log n)

- Optimalität des Verfahrens

  [Berchtold, Böhm, Keim, Kriegel. ACM Smp. Principles of database Systems (PODS), 1997]
  - Prioritätssuche nach [HS 95] ist optimal bzgl. der Anzahl der Seitenzugriffe
  - Beweis (Überblick):
    - » Lemma 1: jeder korrekte Algorithmus muss mind. die Seiten laden, die von der NN-Kugel um q berührt werden
    - » Lemma 2: das Verfahren greift auf Seiten in aufsteigendem Abstand von q zu
    - » Lemma 3: keine Seite s wird zugegriffen, mit MINDIST(q, s) > NN-Distanz(q)

  - **Lemma 1:** Ein korrekter NN-Algorithmus muss mind. die Seiten s laden, die MINDIST(q, s) ≤ NN-Distanz(q) erfüllen.

    Beweis: Angenommen eine Seite s mit MINDIST(q, s) ≤ NN-Distanz(q) wird nicht geladen. Dann kann diese Seite Punkte enthalten (als Datenseite; Directoryseiten können im entspr. Teilbaum Punkte speichern), die näher am Anfragepunkt liegen als der nächste Nachbar. Der nächste Nachbar ist also nicht als solcher validiert, da über Punkte in einem Teilbaum keine Infos bekannt sind, außer dass sie in der entsprechenden Region liegen.

    □

  - **Lemma 2:** Das Verfahren greift auf die Seiten des Index aufsteigend sortiert nach MINDIST zu.

    Beweis: Die Seiten werden in aufsteigender Reihenfolge aus der APL entnommen. Es muss also nur sichergestellt werden, dass nach Entnahme von Seite s keine Seiten s' mehr in APL eingefügt werden, mit MINDIST(q,s') < r := MINDIST(q,s). Alle Seiten, die nach Entnahme von s in APL eingefügt werden, sind entweder Kindseiten von s oder Kindseiten von Seiten s'' mit MINDIST(q,s'') ≥ r. Da die Region einer Kindseite in der Region der Elternseite vollständig eingeschlossen ist, ist die MINDIST einer Kindseite nie kleiner als die der Elternseite. Daher haben alle später eingefügten Seiten eine MINDIST ≥ r.

    □

  - **Lemma 3:** Das Verfahren greift auf keine Seite s zu, mit MINDIST(q,s) > NN-Distanz(q).

    Beweis: Nach Lemma 2 können nach Zugriff auf Seite s nur Punkte p gefunden werden, mit dist(q, p) > MINDIST(q, s). Wäre vor Zugriff auf s ein Punkt p mit dist(q, p) < MINDIST(q, s) gefunden worden, dann wäre s aus der APL gelöscht worden bzw. der Algorithmus hätte vor der Bearbeitung von p angehalten.

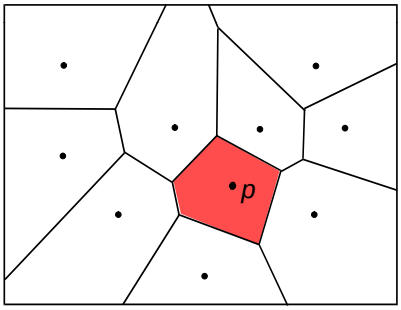
    □

  - Aus Lemma 1-3 ergibt sich, dass der Algorithmus nach [HS 95] optimal bzgl. der Anzahl der Seitenzugriffe ist.

– Hybrider Algorithmus: Voronoi-Diagramme

[Berchtold, Ertl, Keim, Kriegel, Seidl. Proc. Int. Conf. Data Engineeering (ICDE), 1998]

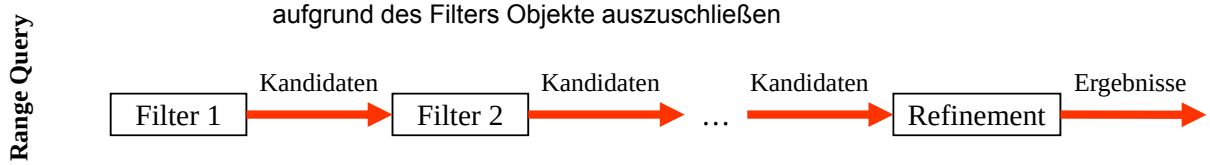
- Nur für Vektordaten!!!
- Idee:
  - Berechne für jeden Punkt *p* den Teil des Datenraumes in dem *p* der nächste Nachbar ist (Voronoi-Zellen)
  - Speichere Voronoi-Zellen in DB
  - NN-Anfrage entspricht Punktanfrage mit *q* auf den Voronoi-Zellen
    => Punkt *p*, in dessen Voronoi-Zelle *q* liegt, ist der NN von *q*



- Problem:
  - Voronoi-Zellen sind konvexe Polygone
  - Ab $d > 2$ sehr komplex (große Anzahl Eckpunkte)
- Lösung: Approximation der Voronoi-Zellen (z.B. mit MBR)
  - Nur Filterschritt, da MBRs sich überlappen können, und q in mehrere dieser MBRs liegen kann
    => Verfeinerung der Kandidaten mit exakten Punktdistanzen

– Mehrstufiger Algorithmus: Filter/Refinement

- Allgemeines
  - Algorithmen verwenden meist nur LB-Filter
  - Bei mehreren Filterschritten: $dist_{Filter1} \leq dist_{Filter2} \leq \ldots$
  - Unterschied zu Bereichsanfragen:
    - » RQs können durch einfache Hintereinanderschaltung der Filterschritte und der Verfeinerung ausgewertet werden
    - » Bei NN-Queries nicht so leicht möglich, da der NN-Kandidat des (ersten) Filters nicht notwendigerweise der exakte NN sein muss
    - » Bei geeigneter Filterdistanz ist es aber wahrscheinlich, dass exakter NN unter den ersten NN-Kandidaten des Filterschritts ist
    - » Rückmeldung der im Refinement ermittelten Distanzen an den Filterschritt um aufgrund des Filters Objekte auszuschließen



- Im folgenden:
  - Verschiedene Auswertungsstrategien
  - Ein Filterschritt (leicht generalisierbar auf mehrere Filterschritte)

- Auswertung mit Bereichsanfrage

  [Korn, Sidiropoulos, Faloutsos, Siegel, Protopapas. Proc. Int. Conf. Very Large Databases (VLDB), 1996]
  [Korn, Sidiropoulos, Faloutsos, Siegel, Protopapas. TKDE 10(6), 1998]

  - Idee
    - » Verfeinerungsdistanz ε eines beliebigen Punktes ist obere Schranke für die NN-Distanz
    - » Folge: ist p der NN von q, so gilt $dist(p, q) \leq \varepsilon$ und $LB_{Filter}(p, q) \leq \varepsilon$
    - » Also: $p \in RQ(q, \varepsilon)$
    - » Gutes ε ist z.B. der NN von q bzgl. der Filterdistanz
  - Prinzip
    - » Auf Filterebene NNQ ausführen
    - » Anschließend eine RQ ausführen (mit Index oder mehrstufig)
    - » Auf dem (hoffentlich kleinen) Ergebnis der RQ den exakten Test (Refinement) durchführen

- Algorithmus

```
NN-MultiStep-RQ(DB, q)
    r = NN-Query auf der Filterdistanz;        // beliebig implementierbar
    ε = dist(q, r);
    candidates = RQ-MultiStep(DB, q, ε);
    result = r ;
    stopdist = ε;

    // Refinement
    FOR EACH p∈candidates DO
        IF dist(p,q) ≤ stopdist THEN
            stopdist = dist(q, p)
            result = p;
    RETURN result;
```

  - Vorteil
    - » Einfacher Algorithmus
  - Nachteil
    - » Leistung stark von Filterselektivität abhängig: schlechter Filter => großes ε => große Ergebnismenge der RQ => hohe Kosten für Verfeinerung

- Auswertung mit unmittelbarer Verfeinerung
  - Idee
    - » Jedes Objekt, das nicht aufgrund des Filters ausgeschlossen werden kann, wird sofort verfeinert
    - » Einbau in einen beliebigen NN-Algorithmus, z.B. in NN-Index-Simple-TS (S. 60)
  - Algorithmus

```
NN-MultiStep-Simple(pa, q)        // pa = Diskadress z.B. der Wurzel des Indexes
  result = ∅;
  p := pa.loadPage();
  IF p.isDataPage() THEN
    FOR i=0 TO p.size() DO
      IF dist_Filter(q, p.getObject(i)) ≤ stopdist THEN
        IF dist(q, p.getObject(i)) ≤ stopdist THEN
          result := getObject(i);
          stopdist = dist(q, p.getObject(i));
  ELSE                            // p ist Directoryseite
    FOR i=0 TO p.size() DO
      IF MINDIST(q, p.getRegion(i)) ≤ stopdist THEN
        result := NN-MultiStep-Simple(p.childPage(i), q)
  RETURN result;
```


  - Vorteil
    - » Gute Speicherplatzkomplexität (je nach NN-Algorithmus!!!), da keine Kandidaten zwischen gespeichert werden müssen
    - » Einfache Erweiterung eines beliebigen NN-Algorithmus
  - Nachteil
    - » Hohe Verfeinerungskosten (fast alle Punkte), wenn Filter wenig selektiv ist oder NN-Algorithmus langsam konvergiert

- Auswertung nach Priorität

  [Seidl, Kriegel. Proc. ACM Int. Conf. Management of Data (SIGMOD),1998]
  - Auf Filterebene läuft „Ranking Query“ ab (siehe Kapitel 2.4.3)
    - » Funktion getNext(): liefert beim ersten Aufruf den 1. Nachbarn, beim zweiten Aufruf den 2. Nachbarn, usw.
    - » Rufe solange getNext() auf, bis das erhaltene Objekt die aktuelle stopdist überschreitet
    - » Verfeinere das erhaltene Objekt und passe ggf. die stopdist an
  - Vorteil
    - » Beweisbar: Algorithmus optimal, d.h. eine minimale Anzahl von Kandidaten werden verfeinert
  - Nachteil
    - » Komplexität des Ranking-Algorithmus (Speicher und/oder Zeit)

– Algorithmus

```
NN-MultiStep-Optimal(DB, q)
    Ranking = initialisiere Ranking bzgl. q auf Filterdistanz      // Kapitel 2.4.3
    result = ∅;
    stopdist = +∞;
    REPEAT
        p = Ranking.getNext();
        filterdist = dist_Filter(p, q);
        IF dist(q, p) ≤ stopdist THEN
            result = p;
            stopdist = dist(q, p);
    UNTIL filterdist > stopdist;
    RETURN result;
```


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

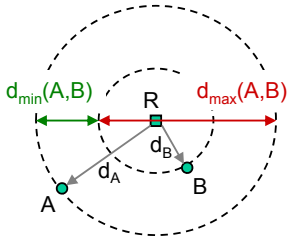
– Grundsätzlich:
  » Unterscheidung der Optimalität bzgl.
    - I/O Kosten (Seitenzugriffe im Index) = Kosten im Filterschritt
    - Kosten für die Berechnung der exakten Distanz = Kosten im Verfeinerungsschritt
  » Optimalität eines mehrstufigen Anfragealgorithmus hängt von der im Filterschritt zur Verfügung stehenden Distanzinformation ab, d.h. mehr Information im Filterschritt
    ⇒ höhere Selektivität des Filters
    ⇒ geringere Verfeinerungskosten

– Ziel:
  » Kandidatenmenge im Filterschritt mehr reduzieren unter Berücksichtigung weiterer Filterkriterien

– Motivation:
  » exakte Distanzberechnung sehr teuer im Vergleich zur Filterdistanzberechnung
    ⇒ 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
    ⇒ optimale Auswertung mit unterer und oberer Distanzabschätzung

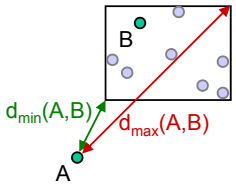
– Beispiele für die Ermittlung von unterer bzw. oberer Distanzabschätzung:
  » Abschätzung basierend auf Referenzpunkten (z.B. M-tree)



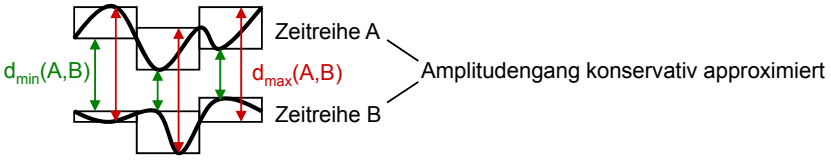
$d_{min}(A,B) = |d_A - d_B|$
$d_{max}(A,B) = d_A + d_B$

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

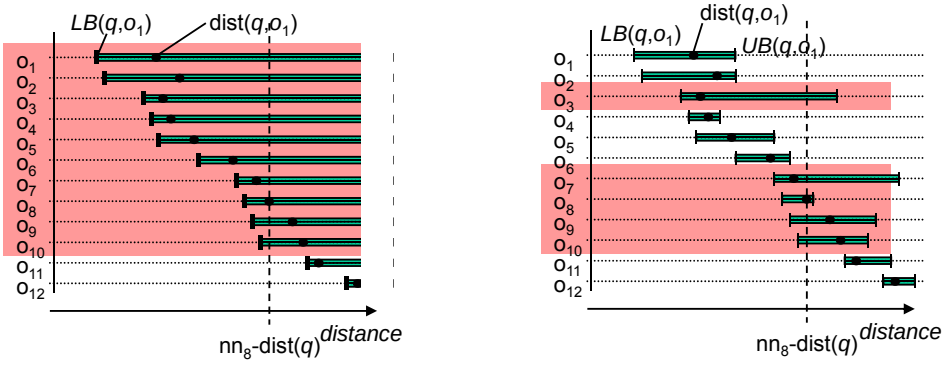
  » Abschätzung basierend auf Regionen (z.B. R-tree)



  » 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

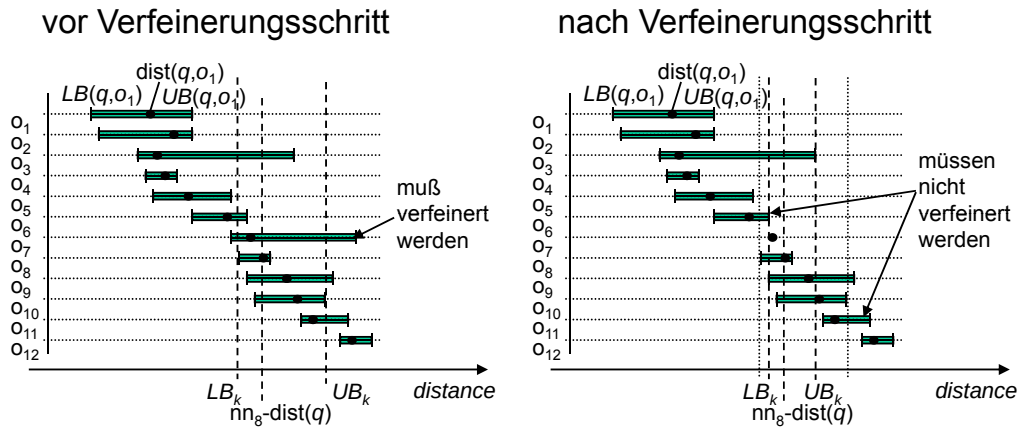
– Optimale (LB+UB)-basierte *k*-NN-Suche

[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:

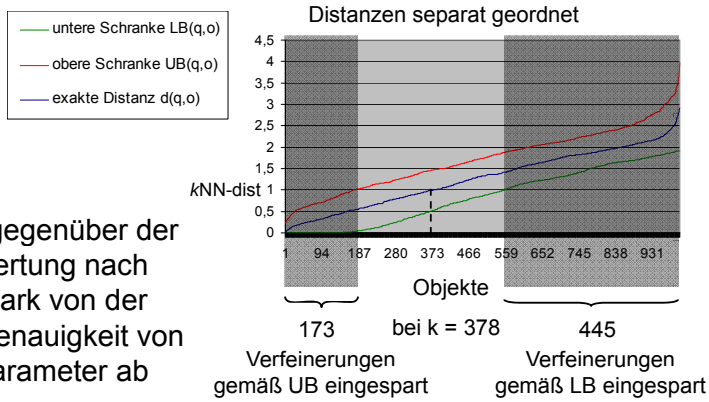


– konservative Approximation der *k*-NN-Distanz nn_k-dist(*q*) durch
  » LB_k = *k* kleinste LB-Filterdistanz
  » Ub_k = *k* kleinste UB-Filterdistanz
– Ziel: Nur diejenigen Objekte verfeinern, deren untere und obere Distanzabschätzung die *k*-NN-Distanz nn_k-dist(*q*) überdeckt
– Beweisbar: Es existiert 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 „$LB(q,c) \leq LB_k \leq UB_k \leq UB(q,c)$“ garantiert, dass nur diejenigen Objekte verfeinert werden, die verfeinert werden müssen, d.h. für die gilt: $LB(q,c) \leq nn_k\text{-dist}(q) \leq UB(q,c)$ ⇒ minimale Verfeinerungen
- Experimente auf Realdaten (NN-Suche auf Zeitreihen mit DTW-Distanz)

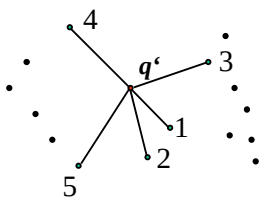


Bemerkung:
Effizienzgewinn gegenüber der einfachen „Auswertung nach Priorität“ hängt stark von der Approximationsgenauigkeit von UB und dem *k* Parameter ab

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!!!