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*) $\subseteq$ DB mit <u>mindestens</u> *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*) $\subseteq$ DB mit <u>exakt</u> *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) $\equiv$ 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* wird als Heap anstelle einer sortierten Liste implementiert
    » Grund: Die Liste *result* braucht nicht vollständig sortiert sein – es reicht sicherzustellen, dass das erste Element in *result* immer den größten Distanzwert hat. Ermöglicht Einfügen in O(log(k)) .

– 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 Anfragen

– Algorithmen mit Multi-Step Architektur
  - Alle drei Alternativen leicht erweiterbar
    - – Auswertung mit Bereichsanfrage
      - » *k*-NN Anfrage statt NN Anfrage 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)

      Bemerkung: Hier gilt die Verfeinerungsoptimalität nur unter der Annahme der Beschränkung auf einen LB-Filter (Grund siehe später)

– Verfeinerungsoptimale *k*-NN Anfrage

- Grundsätzlich:
  - Unterscheidung der Optimalität bzgl.
    - I/O Kosten (Seitenzugriffe im Index) = Kosten im Filterschritt
    - CPU 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

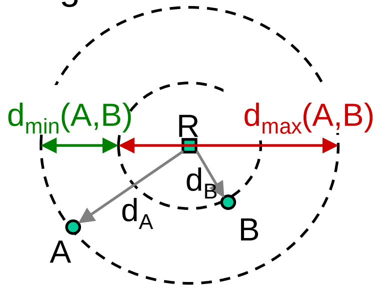
      $\Rightarrow$ höhere Selektivität des Filters

      $\Rightarrow$ weniger Seitenzugriffe, weniger Verfeinerungen
  - Ziel:
    - » Kandidatenmenge im Filterschritt durch Berücksichtigung weiterer Filterkriterien oder zusätzlicher Information weiter reduzieren
  - Motivation:
    - » Exakte Distanzberechnung sehr teuer im Vergleich zur Filterdistanzberechnung

      $\Rightarrow$ Oft lohnt es sich den Filter durch zusätzliche Filterinformationen zu verbessern
    - » In vielen Applikationen können Ähnlichkeitsdistanzen sowohl nach unten als auch nach oben hin effizient abgeschätzt werden

– Beispiele für die Ermittlung von unterer bzw. oberer Distanzabschätzung:

» Abschätzung basierend auf Referenzpunkten (z.B. M-tree)



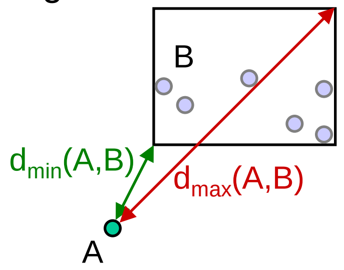
$d_{min}(A,B) = |d_A - d_B|$

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

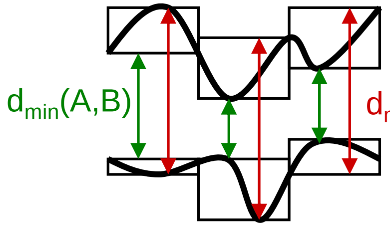
$d_{min}(A,B) \leq d(A,B) \leq d_{max}(A,B)$

LB exakt UB

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



» Individuelle Abschätzung (Applikations/Daten abhängig)

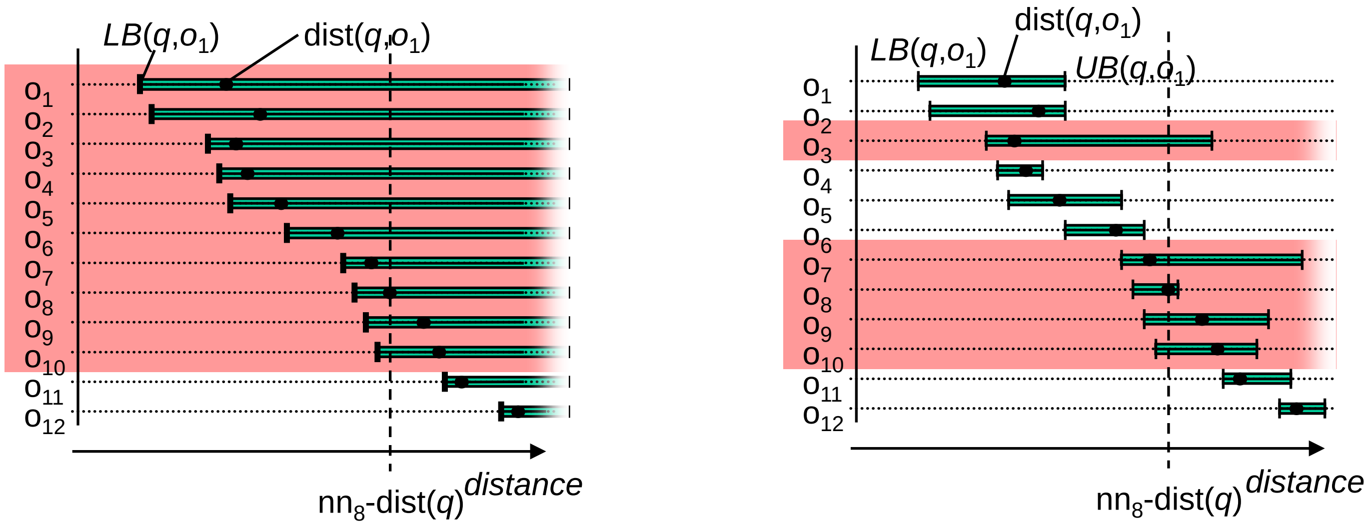


– Idee:

» zusätzlich zur unteren Distanzabschätzung (LB) wird auch obere Distanzabschätzung (UB) im Filterschritt miteinbezogen

$\Rightarrow$ optimale Auswertung mit unterer und oberer Distanzabschätzung

- 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) ≤ nn_k-dist(q) ≤ 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]

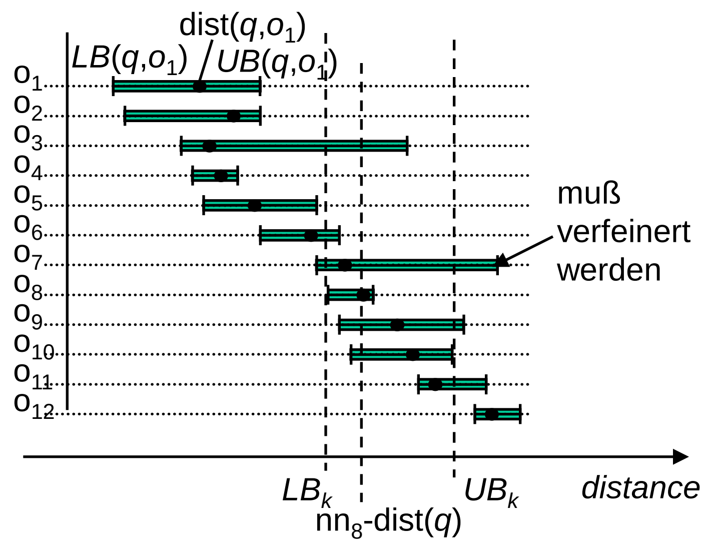
- Optimalität:

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

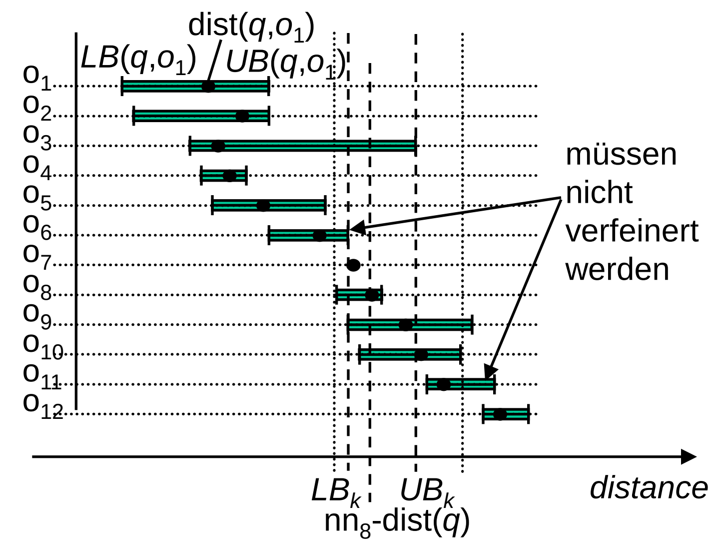
  – Beruht auf dem Prinzip der iterativen Verfeinerung (analog zu „Auswertung nach Priorität“ s. Folie 77):
  - » auf Filterebene läuft „Ranking Query“ ab (siehe Kapitel 2.4.3)
  - » Filtern aufgrund von unterer und oberer Schranke
  - » nach jedem Verfeinern wird erneut gefiltert
  - » Objekt nur dann 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 $c \in$ *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 $c \in$ *candidates* zu *result* hinzu; // Abbruchbedingung

else

verfeinere alle $c \in$ 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*;

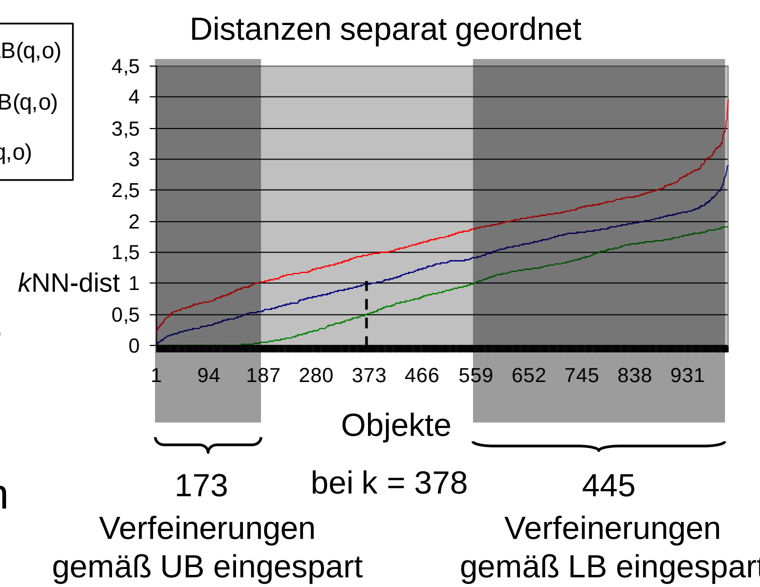
- **Optimalität gemäß der Seitenzugriffe:**

  In *Schritt 1*: die Bedingung „$LB_{next} \leq LB_k$“ garantiert, dass nur die notwendigen Objekte angefordert werden $\Rightarrow$ minimale Seitenzugriffe

- **Optimalität gemäß der Anzahl der Verfeinerungen:**

  In *Schritt 3* die Bedingung „$\text{LB}(q,c) \leq \text{LB}_k \leq \text{UB}_k \leq \text{UB}(q,c)$“ garantiert, dass nur diejenigen Objekte verfeinert werden, die verfeinert werden müssen, d.h. für die gilt: $\text{LB}(q,c) \leq \text{nn}_k\text{-dist}(q) \leq \text{UB}(q,c) \Rightarrow$ minimale Verfeinerungen

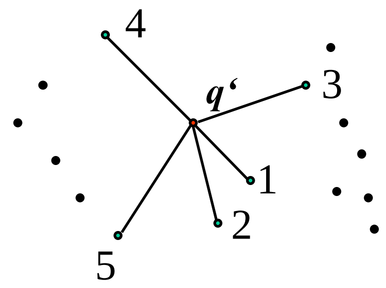
- Experimente auf Realdaten (NN-Suche auf Zeitreihen mit DTW-Distanz)



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!!!

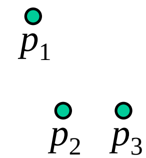
2.5 Reverse Nächste Nachbarn Anfragen

2.5.1 Allgemeines

- Eigenschaften
  - Benutzer gibt Anfrageobjekt *q* vor
  - Ergebnis enthält alle Objekte, die *q* als nächsten Nachbarn haben
  - Analog: Reverse *k*-nächste Nachbarn
  - Mehrdeutigkeiten (bei NN) entsprechend behandeln
- Formal
  - Reverse nächste Nachbarn $RNN(q) = \{o \in DB \mid q \in NN(o)\}$
  - Reverse *k*-nächste Nachbarn $RNN(q,k) = \{o \in DB \mid q \in NN(o,k)\}$
- Anwendungsbeispiel:
  - Standortsuche für neue Filiale (welche Kunden haben die neue Filiale als „nächsten Nachbarn“)

– Zusammenhang zwischen NN und RNN

- NN ist keine symmetrische Relation
  - $y \in NN(x) \cancel{\Leftrightarrow} x \in NN(y)$
  - $y \in NN(x) \cancel{\Leftrightarrow} y \in RNN(x)$
- RNN ist ein „eigenständiges Problem



	NN	RNN
p_1	$\{p_2\}$	$\{\}$
p_2	$\{p_3\}$	$\{p_3, p_1\}$
p_3	$\{p_2\}$	$\{p_2\}$

– Basisalgorithmus (sequential scan): nichtdeterministisch

```
RNN-SeqScan(DB, q, k)
    resultSet = ∅;
    FOR i=1 TO n DO
        neighbors = NN-SequScan(DB, DB.getObject(i), k);
        IF q ∈ neighbors THEN
            resultSet.add(DB.getObject(i));
    RETURN resultSet;
```

- Offensichtliche Verbesserung
  - Statt NN-SequScan einen besseren NN-Algorithmus verwenden

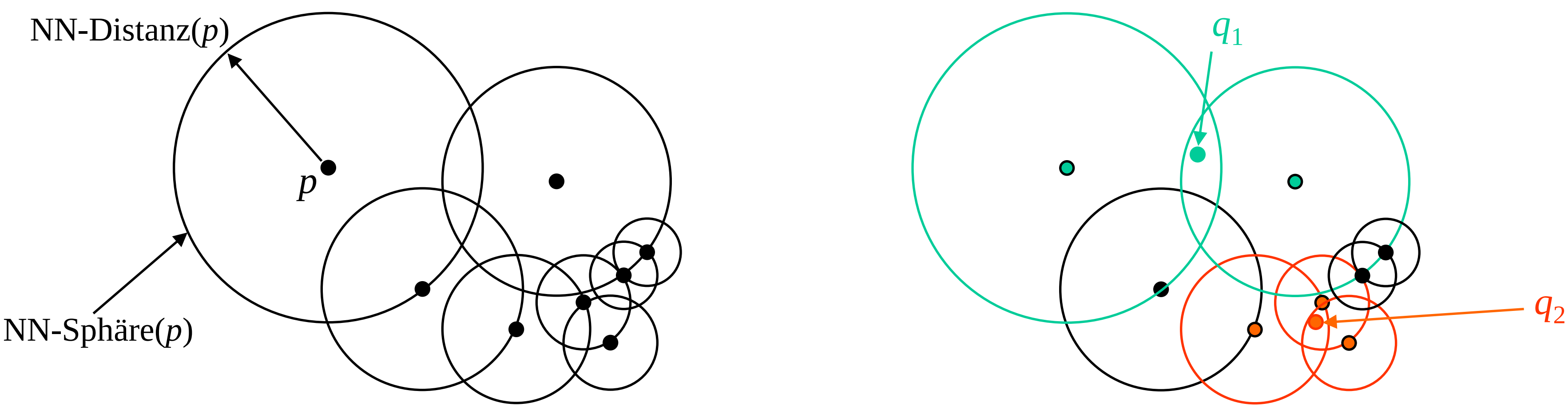
– Index-basierte Methoden für die RkNN-Suche

- Annahme:
  - Daten/Objekte in einer Baumartigen Indexstruktur, z.B. R-tree, organisiert
  - Suche erfordert hierarchischen Durchlauf der Directory-Seiten
- Ziel
  - Bei der Suche möglichst früh Seiten auf höheren Index-Level ausschließen (d.h. effektive Pruning-Strategien)

    ==> möglichst starke Einschränkung des Suchraums
- Pruning-Strategien
  - Generell gibt es zwei Index-basierte Pruning-Strategien für die RkNN-Suche
  - Self-Pruning Strategien:
    - » Punkte/Seiten schließen sich selbst aus
    - » Basieren auf k-NN-dist-Abschätzungen angewandt auf Punkte/Seitenregionen
    - » Punkte/Seiten, deren k-NN-dist-Bereich den Anfragepunkt nicht enthalten können ausgeschlossen werden
  - Mutual-Pruning Strategien:
    - » Punkte/Seiten schließen sich gegenseitig aus
    - » Basieren auf Voronoi-Hyperebenen

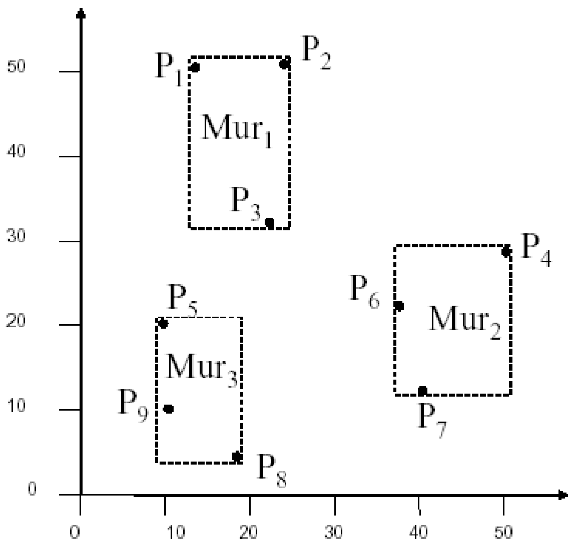
2.5.2 Self-Pruning Strategien

- $p \in \mathrm{RNN}(q) \Leftrightarrow \mathrm{dist}(p, q) \leq \text{NN-Distanz}(p)$
- Materialisiere für alle Objekte die NN-Distanz
- Prüfe, ob $\mathrm{dist}(p, q) \leq \text{NN-Distanz}(p)$ statt während der Anfrage eine NN-Query für alle Objekte zu berechnen

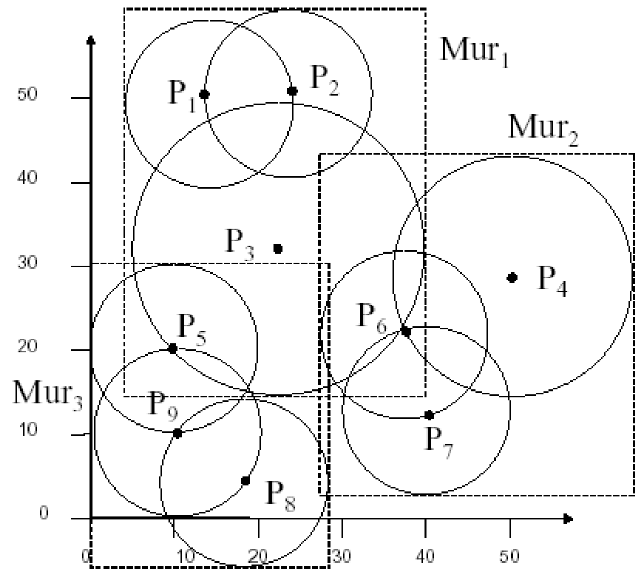


- Für Vektordaten
  - RNN-Query ist Punktanfrage bzgl. der NN-Sphären der Punkte (vgl. Voronoi-Ansatz zur NN-Query)
  - Speichere NN-Sphären in einem Index für ausgedehnte Objekte (z.B. R-Baum)

- RNN-Baum [Korn, Muthukrishnan. ACM Int. Conf. Management of Data (SIGMOD), 2000]
  - RNN-Queries für Vektordaten
  - Berechne NN-Distanz für alle DB-Punkte
  - Speichere statt Punkte alle NN-Sphären der Punkte in R-Baum



Normaler R-Baum



RNN-Baum

- Algorithmus zur NN-Suche
  - Datenseiten enthalten Kreise, d.h. Objekte der Form (Punkt, Radius)
    - » Punkt = DB-Objekt (Mittelpunkt)
    - » Radius = NN-Distanz(Punkt)

```
RNN-Tree-Search(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).Point) ≤ p.getObject(i).Radius THEN
                result := result ∪ getObject(i).Point;

    ELSE                           // p ist Directoryseite
        FOR i=0 TO p.size() DO
            IF MINDIST(q, p.getRegion(i)) = 0 THEN
                result := result ∪ RNN-Tree-Search(p.childPage(i), q);
    RETURN result;
```

- Vorteil
  - » Sehr gute Performanz (Pruning-Power) bei RNN-Anfragen
- Nachteile
  - » *k* muss fest vorgegeben sein
  - » Nur für Vektordaten
  - » Hohe Überlappungen der Seitenregionen im Directory führt zu schlechter Performanz bei normalen NN-Anfragen

    Lösung: speichere einen RNN-Baum und einen konventionellen R-Baum
  - » Schlechte Performanz bei Einfügungen und Löschungen

    Beispiel: *Einfügen* von *p*

„Normaler" Index:
- Bestimme NN(*p*) und füge Kreis (*p*, NN-Distanz(*p*)) in Index ein

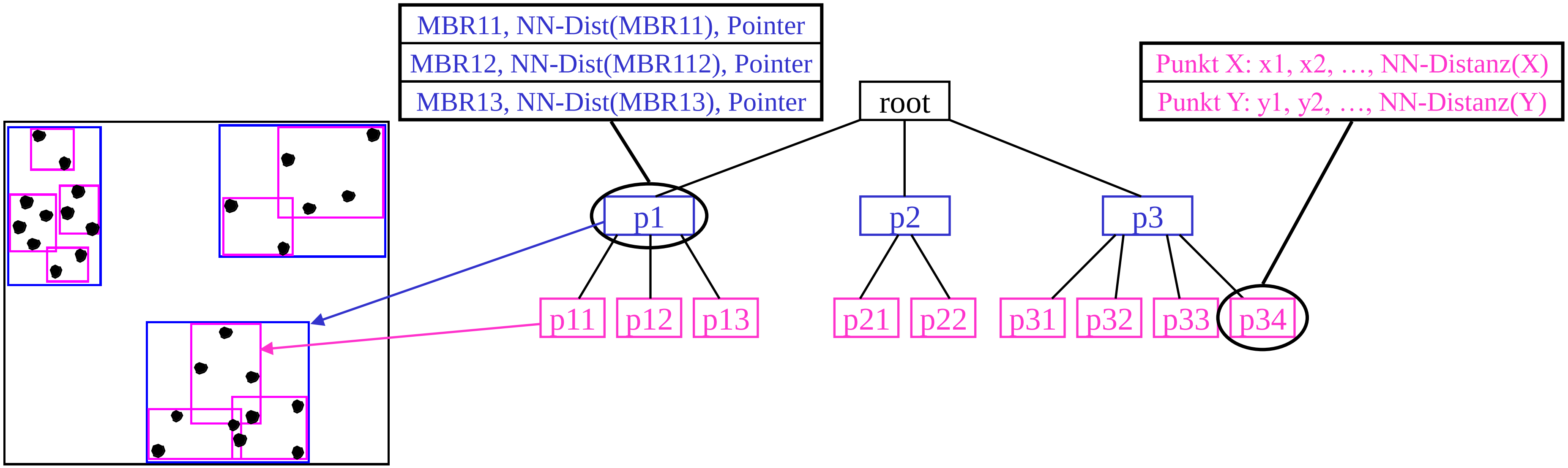
Zusätzlich für RNN-Baum:
- Bestimme RNN(*p*)
- Erneuere NN-Sphären aller $o \in$ RNN(*p*)
- Erneuere Seitenregionen (von *p* und allen *o*) betroffener Datenseiten
- Erneuere rekursiv Seitenregionen der Vaterseiten

- Varianten:
  - » Voronoi-Zellen statt NN-Sphären
  - » Andere Verfeinerungsreihenfolge (siehe NN-Algorithmen)

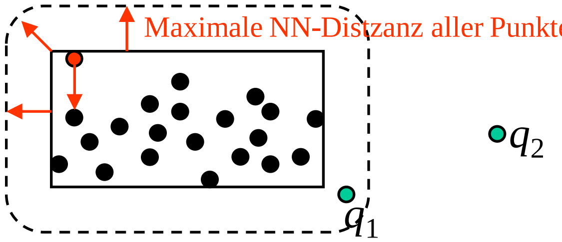
– RdNN-Baum [Yang, Lin. IEEE Int. Conf. Data Engineering (ICDE), 2001]
  - Prinzip
    – Idee des RNN-Baum, ABER
    – Speichere die DB-Objekte statt NN-Sphären
    – Speichere zu jedem Punkt in der DB seine NN-Distanz
      » Zu jedem Punkt zusätzlich einen Wert abspeichern
      » Zu jeder Seitenregion *s* zusätzlich noch den Wert

$$\max_{child \in s} child.getNNDist()$$

abspeichern (Maximum aller NN-Distanzen in den Kinderseiten)



– Ausschluss von Directory-Seiten, wenn MINDIST(q,Seitenregion) größer ist, als die aggregierte NN-Distanz der Seitenregion



Query q_2: Seite kann ausgeschlossen werden; kein Punkt der Seite kann q_2 als NN haben

Query q_1: Seite kann nicht ausgeschlossen werden

- Algorithmus zur RNN-Suche

RdNN-Tree-Search(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*)) ≤ p.getNNDist(*i*) **THEN**
                result := result ∪ getObject(*i*);

    ELSE // p ist Directoryseite
        FOR *i*=0 **TO** p.size() **DO**
            IF MINDIST(*q*, p.getRegion(*i*)) ≤ p.getNNDist(*i*) **THEN**
                result := result ∪ RdNN-Tree-Search(p.childPage(*i*), *q*);
    RETURN result;

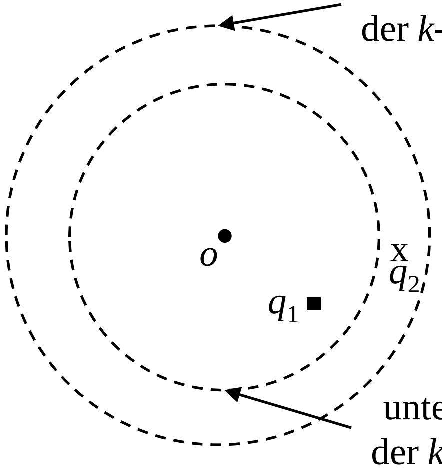
- Auch hier wieder alle möglichen anderen algorithmischen Lösungen zur NN-Suche anwendbar (Prioritätssuche, etc.)
- Vorteil
  - » Sehr gute Selektivität, damit gute Performanz bei Anfragen
  - » Seitenüberlappung wie bei „normalem“ R-Baum, daher kein extra Index für NN-/RQ-Anfragen nötig
- Nachteil
  - » *k* muss fest vorgegeben sein
  - » Nur für Vektordaten (aber: Konzept sehr leicht für M-tree erweiterbar)
  - » Weiterhin schlechte Performanz bei Einfügungen und Löschungen

– MRkNNCoP-Baum

[Achtert, Böhm, Kröger, Kunath, Pryakhin, Renz. ACM Int. Conf. Management of Data (SIGMOD), 2006]

- Vergleich bisheriger Verfahren
  - RNN-Tree/RdNN-Tree: Vorberechnung der NN-Distanz
    - » Wert für *k* ist fix und vorher bekannt
    - » Update-Problematik
    - » Dafür: erweiterbar auf metrische Daten (z.B. M-Tree)
  - Geometrische Suche
    - » Nur für Vektordaten
    - » Teurer Verfeinerungsschritt, schlechtere Selektivität
    - » Dafür: beliebiges *k*, keine Update-Problematik
- Idee:
  - Benutze die gute Selektivität der vorberechneten NN-Distanzen
  - Berechne für mehrere (am besten alle) Werte für *k* die *k*-NN-Distanzen vor
  - Problem: Speicherung aller Distanzen zu aufwendig
    - » Pro Objekt alle *k*-NN-Distanzen => Index wäre sehr hoch => hohe Kosten
  - Lösung: Approximiere die *k*-NN-Distanzen
    - » Approximation sollte untere Schranke (LB) der *k*-NN-Distanz sein => true drops, wir können Objekte (Seiten) frühzeitig ausschließen
    - » Zusätzliche Approximation als obere Schranke (UB) => true hits

- Bewertung von Objekt *o* (analog: Seiten) mit UB- und LB-Approximationen
  - $\text{dist}(o,q) \leq LB_{k\text{-NN-Dist}}(o)$
    => *o* true hit, d.h. $o \in \text{RNN}(q, k)$
    » Beispiel: $q = q_1$
  - $\text{dist}(o,q) \geq UB_{k\text{-NN-Dist}}(o)$
    => *o* true drop, d.h. $o \notin \text{RNN}(q, k)$
    » Beispiel: $q = q_3$
  - $UB_{k\text{-NN-Dist}}(o) \leq \text{dist}(o,q) \leq LB_{k\text{-NN-Dist}}(o)$
    => *o* Kandidat
    » Beispiel: $q = q_2$



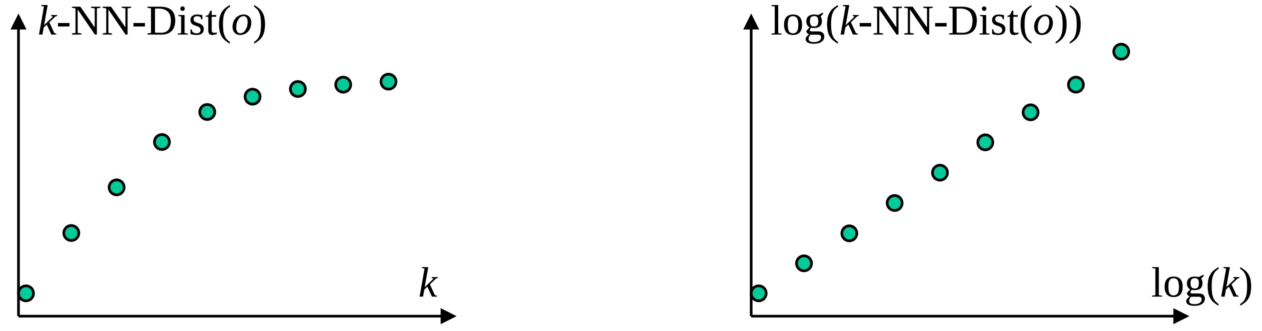
- Gegeben: für jedes Objekt *o* eine Sequenz der *k*-NN-Distanzen, <1-NN-Dist(*o*), 2-NN-Dist(*o*), …, k_{max}-NN-Dist(*o*)> für ein hinreichend großes k_{max}
- Frage: wie kann ich UB- und LB-Approximation dieser *k*-NN-Distanzen berechnen und kompakt speichern?

- Lösung aus der Theorie der Selbstähnlichkeit
  - Potentzgesetz gilt für Verhältnis zwischen
    - » dem Radius einer Hyperkugel
    - » der Anzahl an Objekten innerhalb der Hyperkugel

$$encl(\varepsilon) \propto \varepsilon^{d_f}$$

wobei $encl(\varepsilon)$ = #Objekte innerhalb der Kugel
d_f = „Fraktale Dimension“

  - Übertragung auf k-NN-Sphäre:
    - » ε = k-NN-Distanz
    - » $encl(\varepsilon) = k$
  - Im log-log-Raum: $\log(k-\mathrm{NN}-\mathrm{Dist}(o)) \propto \frac{\log(k)}{d_f}$



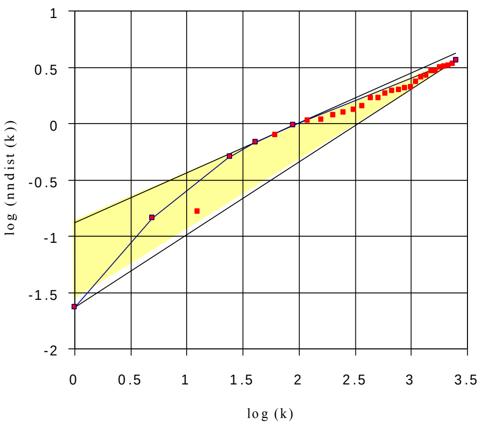
– In der Realität verhalten sich die Distanzen nicht wie perfekte Linien im log-log-Raum
– Trotzdem: im log-log-Raum können die *k*-NN-Distanzen mit einer Linie approximiert werden
– Das ist erheblich billiger als alle *k*-NN-Distanzen zu speichern, oder andere Funktionen höherer Ordnung zu verwenden, um die Distanzen im normalen *k* / *k*-NN-Dist – Raum zu approximieren

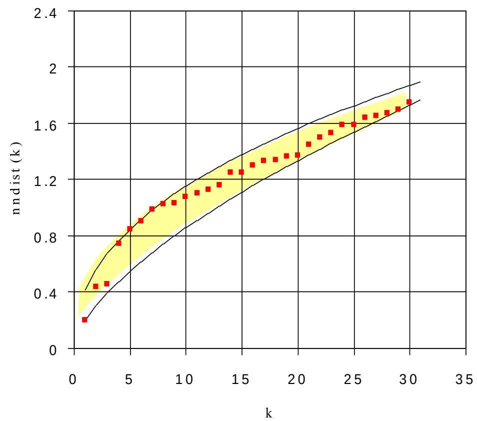
- LB- und UB-Approximationen
  – UB-Approximation ist eine Linie im log-log-Space, sodass

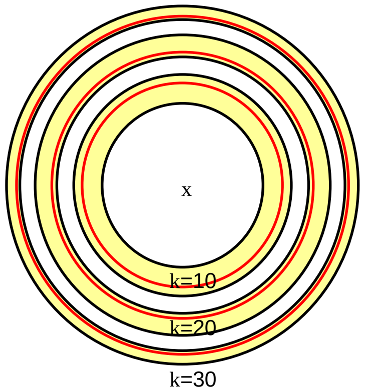
$$\forall k \le k_{\max} : k\text{-NN-Dist}(o) \le UB_{k\text{-NN-Dist}}(o)$$

  – LB-Approximation ist eine Linie im log-log-Space, sodass

$$\forall k \le k_{\max} : k\text{-NN-Dist}(o) \ge LB_{k\text{-NN-Dist}}(o)$$







- Jedem Objekt wird zugeordnet
  - Eine LB-Approximation der *k*-NN-Distanzen
  - Eine UB-Approximation der *k*-NN-Distanzen
- Jeder Seite im Index wird zugeordnet
  - Eine LB-Approximation der LB-Approximationen der Kindseiten
  - UB-Approximation wird nicht gespeichert, da zu wenig selektiv

- Vorteil:
  - Beliebiges *k*
  - Für allgemein metrische Daten (M-Tree) oder Vektordaten (z.B. X-Tree)
  - Durch UB- und LB-Approximationen höhere Filterselektivität als Geometrisches Verfahren => weniger Kandidaten die verfeinert werden müssen

- Nachteil
  - Updateproblematik
  - k_{max} muss bekannt sein (ABER i.d.R. kein Problem)
  - Teure Verfeinerung nötig (ABER i.d.R. deutlich weniger Kandidaten)

- ***Filter***-Algorithmus für allgemein metrische Daten (M-tree)

Knoten Node = (RoutingObj, CovRadius)

MINDIST(q,Node) = max{dist(q,Node.RoutingObj) – CovRadius, 0}

```
MRkNNCoP-Tree-Search(DB, q)      // DB als MRkNNCoP-Tree organisiert
  result = ∅;
  candidates = ∅;
  queue = LIST OF (dist:Real, obj:Object) ORDERED BY dist ASCENDING;
  queue = [(0.0, DB.root)];
  WHILE NOT queue.isEmpty() DO
    p = queue.first().Object;
    IF p.isDataPage() THEN
      FOR i=0 TO p.size() DO
        IF dist(q, p.getObject(i)) ≤ LB_{k-NN-Dist}(p.getObject(i)) THEN
          result := result ∪ getObject(i);
        ELSE IF dist(q, p.getObject(i)) ≤ UB_{k-NN-Dist}(p.getObject(i)) THEN
          candidates := candidates ∪ getObject(i);

    ELSE                    // p ist Directoryseite
      FOR i=0 TO p.size() DO
        IF MINDIST(q, p.getRegion(i)) ≤ UB_{k-NN-Dist}(p.getRegion(i)) THEN
          queue.insert((MINDIST(q, p.getRegion(i)), p.childPage(i)));
```