Proseminar

Datenreduktionstechniken

Sommersemester 1999

Universität des Saarlandes

Ausarbeitung

zum Vortrag

Indexbäume als Suchstrukturen Nearest-Neighbor-Algorithmus

Stephan Wiehr

2. Juli 1999

Inhaltsverzeichnis

1 Einführung

In räumlichen Datenbanken ist es oft von Interesse, Datenobjekte nach ihrer Distanz zu einem Referenz-Objekt geordnet anzufragen. Vorstellbar wären Anfragen wie „Welche Stadt mit mehr als einer Million Einwohnern liegt Chicago am nächsten?“

Während bei einer eindimensionalen Datenmenge eine ähnliche Anfrage mithilfe einer geeigneten Indexstruktur (z.B. einem B*-Baum) mit Kosten von $O(\log n)$ für n Tupel beantwortet werden kann, ist dies im mehrdimensionalen Fall nicht so einfach.

Denkbar wären folgende drei Ansätze:

1. Schätze einen Radius um Chicago und überprüfe die Einwohnerzahlen der Städte in diesem Umkreis. Das Problem einer solchen Schätzung ist, daß ein zu großer oder zu kleiner Radius geschätzt werden könnte. Wäre die Suchregion zu groß, würden unnötig viele Städte überprüft, wäre sie zu klein würde eventuell keine Stadt gefunden, die das Einwohner-Kriterium erfüllt, und die Suchregion müßte erweitert werden.

2. Sortiere alle Städte nach ihrer Distanz zu Chicago wäre eine drastische Lösung. Das Problem hierbei ist, daß alle Städte bei Änderung des Referenzobjektes neu sortiert werden müßten, was wiederum hohe Kosten verursachen würde.

3. Suche die k nächsten Städte zu Chicago und überprüfe ihre Einwohnerzahlen. Hier liegt das Problem in der Wahl von k. Wie im 1. Ansatz, könnte eine zu großer oder zu kleiner Wert für k gewählt werden. Bei zu großem k würden zu viele Nachbarn berechnet, wäre k zu klein müßten $m > k$ nächste Nachbarn gefunden werden, was die Vorherige Berechnung der k nächsten Nachbarn beinhaltet.

Bevor auf die Lösung dieser Problemstellung eingegangen wird, jedoch erst einige Grundlagen.

2 Indexstrukturen

Indexstrukturen werden benötigt, um schnellere Zugriffe auf Datensätze einer Datenbank zu ermöglichen. Hierbei werden Strukturen benutzt, die leicht zu durchsuchen sind und somit eine bessere Performance bei Anfragen und Joins gewährleisten.
Im Laufe der Zeit wurden zahlreiche Indexstrukturen entworfen, die mitunter auf sehr spezielle Datenmengen zugeschnitten sind. Als Indexstrukturen werden insbesondere Indexbäume und Hashmethoden benutzt. Beispiele für eindimensionale Indexstrukturen sind der B-Baum, lineares Hashing und erweiterbares Hashing. Der R-Baum ist ein Beispiel für eine Indexstruktur, die speziell für mehrdimensionale Datenmengen entworfen wurde.

Suchstrategien in Bäumen orientieren sich an zwei verschiedenen Verfahren:

- Tiefensuche (depth-first-search): Der Baum wird erst in der Tiefe, danach in der Breite durchlaufen
- Breitensuche (breadth-first-search): Der Baum wird erst in die Breite, danach in die Tiefe durchlaufen

2.1 B-Bäume

Ein B-Baum [siehe Abb. 1] ist ein balancierter Baum für effizientes Suchen, Einfügen und Löschen. Ein Knoten des Baums enthält k Datensätze, Nichtblattknoten enthalten zusätzlich noch $k + 1$ Sohnzeiger.
Die Worst-Case-Komplexität für Suchen, Einfügen und Löschen in einem B-Baum mit n Datensätzen beläuft sich auf $O(\log_k n)$.

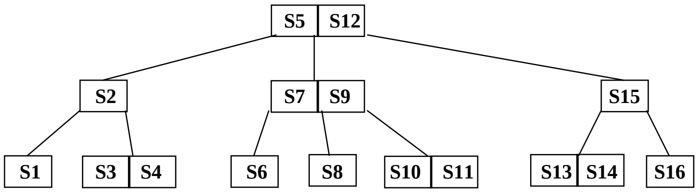

Abbildung 1: B-Baum

Formale Definition:

Ein Baum heißt B-Baum der Ordnung m $(m \geq 1)$, wenn gilt

1. $m \leq$ Anzahl Schlüssel pro Knoten (außer Wurzel) $\leq 2m$
2. Ein Nichtblattknoten mit k Schlüsseln $x_1, \ldots, x_k$ hat genau $k + 1$ Söhne $t_1, \ldots, t_{k+1}$, so daß
   - $\forall$ Schlüssel s im Teilbaum t_1 gilt $s < x_1$
   - $\forall$ Schlüssel s im Teilbaum t_i $(2 \leq i \leq k)$ gilt $x_{i-1} < s < x_i$
   - $\forall$ Schlüssel s im Teilbaum t_{k+1} gilt $x_k < s$
3. alle Blätter haben dasselbe Niveau

2.2 B*-Bäume

B*-Bäume sind eine häufig verwendete Indexstruktur für eindimensionale Daten. Der Hauptunterschied zu B-Bäumen besteht darin, daß die eigentlichen Daten nur in den Blattknoten gespeichert werden. Weiterhin führen B*-Bäume einen weiteren Parameter m^* ein, der den Füllgrad der Blattknoten bestimmt (analog zu m). Auf diese Weise wird der Ausgangsgrad in der Blatt-Ebene erhöht, so daß die Tiefe gegenüber B-Bäumen durch die Verlagerung der Datensätze in die Blattknoten nicht ansteigt. Desweiteren erfordert ein Scan in Sortierreihenfolge nur eine Traversierung der Blätter.

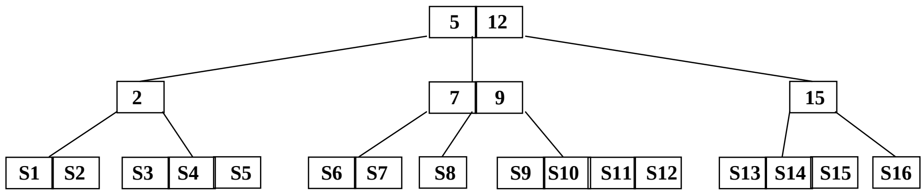

Abbildung 2: B*-Baum

Abbildung 2 zeigt den B*-Baum für die gleiche Datenmenge, die in Abbildung 1 benutzt wurde.

Formale Definition:

Ein Baum heißt B*-Baum der Ordnung (m, m^*) $(m \geq 1, m^* \geq 1)$, wenn gilt

1. $m \leq$ Anzahl Schlüssel pro Nichtblattknoten (außer Wurzel) $\leq 2m$
2. Ein Nichtblattknoten mit k Schlüsseln $x_1, \ldots, x_k$ hat genau $k+1$ Söhne $t_1, \ldots, t_{k+1}$, so daß
   - $\forall$ Schlüssel s im Teilbaum t_1 gilt $s \leq x_1$
   - $\forall$ Schlüssel s im Teilbaum t_i $(2 \leq i \leq k)$ gilt $x_{i-1} < s \leq x_i$
   - $\forall$ Schlüssel s im Teilbaum t_{k+1} gilt $x_k < s$
3. alle Blätter haben dasselbe Niveau
4. $m^* \leq$ Anzahl Schlüssel pro Blattknoten $\leq 2m^*$

2.3 R-Bäume

R-Bäume stellen eine Erweiterung von B*-Bäumen für räumliche Objekte dar.

Jeder Knoten eines R-Baums enthält ein Feld von Einträgen der Form (*Schlüssel*, *Zeiger*). *Schlüssel* ist hierbei ein Hyper-Rechteck, das die Datenobjekte des Unterbaums minimal begrenzt. *Zeiger* ist in Blattknoten eine Objektreferenz (z.B. eine TupelID), in inneren Knoten ein Sohnzeiger auf einen Knoten der nächsttieferen Ebene.

Beim R-Baum gibt es für die Anzahl der Schlüssel pro Knoten eine untere Schranke m und eine obere Schranke M $(M \geq 2m)$,so daß $m \leq$ Anzahl der Schlüssel pro Knoten $\leq M$. Die eigentlichen Daten werden wie beim B*-Baum in den Blättern gespeichert.

In Abbildung 3 findet sich eine räumliche Darstellung von 9 Liniensegmenten (a-i). Die begrenzenden Rechtecke der verschiedenen Ebenen des R-Baums sind durch verschieden gestrichelte Linien angezeigt.(R0 - gepunktet; R1 und R2 - gestrichelt; R3, R4, R5 und R6 - durchgezogen) Der Punkt q dient für die Beispiele in 4.3 (k-Nearest-neighbor-Algorithmus) und 5.3 (inkrementeller Nearest-Neighbor-Algorithmus) als Referenzobjekt für die Suche nach nächsten Nachbarn.

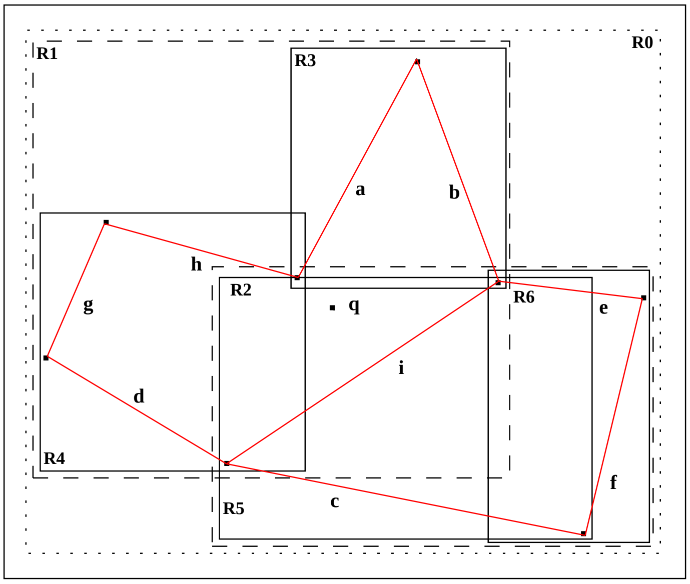

Abbildung 3: Räumliche Darstellung von 9 Liniensegmenten

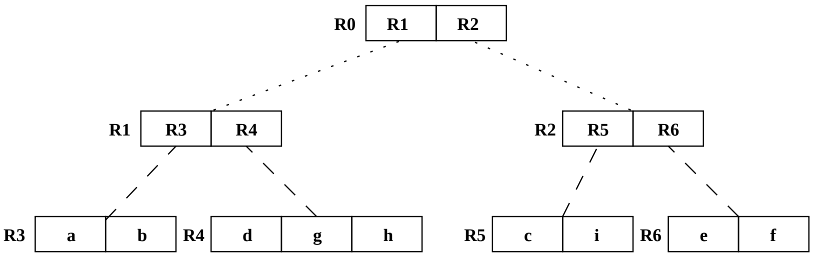

Abbildung 4: R-Baum für Abb. 3

In Abbildung 4 findet sich der zu Abbildung 3 gehörige R-Baum.

3 Grundlagen

3.1 (k-)Nearest-Neighbor-Query

Daten einer räumlichen Datenbank können auf verschiedene Arten abgefragt werden. Mögliche Anfragen können sich zum Beispiel auf Beinhaltung eines Punktes, Überschneidungen mit einem gegebenem Objekt, Beinhaltung eines gegebenen Objektes oder Enthaltensein in einem Objekt beziehen. Beim Nearest-Neighbor-Algorithmus ist die minimale Distanz zu einem gegebenen Referenzobjekt von Interesse.

Formal läßt sich eine Nearest-Neighbor-Query wie folgt definieren [1] :

$$NNQ(q) \quad = \quad \{x | \forall o : dist(q.G, x.G) \leq dist(q.G, o.G)\}$$

d.h. $NNQ(q)$ ist die Menge aller Objekte x, deren Distanz zum gegebenen Objekt q minimal ist.

Analog läßt sich eine k-Nearest Neighbor Query definieren:

$$\begin{aligned} k - NNQ(q) \quad = \quad & \{x_i | i \in \{1, \ldots, k\}, \\ & \forall i : dist(q.G, x_i.G) \leq dist(q.G, x_{i+1}.G) \\ & \wedge \forall o, i : dist(q.G, x_i.G) \leq dist(q.G, o.G)\} \end{aligned}$$

d.h. $k - NNQ(q)$ ist die Menge der k Objekte $x_1, \ldots, x_k$, deren Distanz zum gegebenen Objekt q minimal ist und die bezüglich ihrer Distanz zu q sortiert sind.

3.2 Distanzfunktionen und Konsistenz

Die Distanz zweier Objekte wird oft definiert als die Distanz ihrer nächsten Punkte,jedoch sind auch andere Definitionen denkbar. Hierbei ist die Distanzfunktion abhängig von der Dimension des Datenraumes und von der gewählten Metrik.

Im zweidimensionalen Raum ließe sich die Distanz zweier Punkte $a = (a_x, a_y)$ und $b = (b_x, b_y)$ beispielsweise durch die euklidische Metrik beschreiben:

$$dist(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$$

Denkbar wäre jedoch durchaus auch die Verwendung der Manhattan-Metrik:

$$dist(a, b) = |a_x - b_x| + |a_y - b_y|$$

Desweiteren müssen d_o (Abstand von Objekten zum Referenzobjekt) und d_n (Abstand von inneren Knoten zum Referenzobjekt) nicht auf der gleichen Metrik basieren. Einzig wichtig ist die Konsistenz von d_o und d_n, die sich formal wie folgt definieren läßt:

Definition: Die Funktionen d_o und d_n sind konsistent

$$\Longleftrightarrow \forall q, e\, \exists n \in N, e \sqsubseteq N : d(q, n) \leq d(q, e)$$

(Dabei bezeichne $e \sqsubseteq N$ die Tatsache, daß e genau in N enthalten ist.)

Diese Konsistenzbedingung garantiert uns, daß für jedes Referenzobjekt q die begrenzenden Rechtecke eines beliebigen Objektes e mindestens genauso nahe an q sind wie e selbst..

[1] $q.G \subseteq E^d$ bezeichnet die räumliche Ausdehnung des Objekts q, E^d bezeichnet den d-dimensionalen euklidischen Raum

4 k-Nearest-Neighbor-Algorithmus

4.1 Pseudocode

```
KNEAREST(k, QueryObject, SpatialIndex)
NearestList ← NEWLIST(k)
KNEARESTTRAVERSAL(NearestList, k, QueryObject, SpatialIndex.
RootNode)
return NearestList
```

```
KNEARESTTRAVERSAL(NearestList, k, QueryObject, Node)
if Node is a leaf node then
  for each Object in Node do
    if DIST(QueryObject, Object) < NearestList.MaxDist then
      INSERT(NearestList, DIST(QueryObject, Object), Object)
    end if
  end for
else
  ActiveBranchList ← entries in Node
  SORTBRANCHLIST(QueryObject, ActiveBranchList)
  for each Child node in ActiveBranchList do
    if DIST(QueryObject, Child) < NearestList.MaxDist then
      KNEARESTTRAVERSAL(NearestList, k, QueryObject, Child)
    else
      exit loop
    end if
  end for
end if
```

4.2 Erläuterungen

Im k-Nearest Neighbor Algorithmus wird der R-Baum ähnlich einer rekursiven Tiefensuche durchlaufen. *NearestList* ist die Liste der k nächsten Nachbarn und enthält außerdem die maximale Distanz (*NearestList.MaxDist*) ihrer Einträge (zu Beginn: ∞). Falls der momentane Knoten ein innerer Knoten ist, werden seine Kinder in die *ActivebranchList* gespeichert und diese sortiert. Für jeden dieser Kindknoten, dessen Distanz zum Referenzobjekt kleiner ist als die in *NearestList* gespeicherte bisherige maximale Distanz wird der Algorithmus rekursiv aufgerufen. Auf diese Weise wird die *ActivebranchList* als "Wegweiser" durch den Baum benutzt. Ist der momentane Knoten ein Blattknoten, so werden die Objekte des Blattes in *NearestList* eingefügt, sofern sie eine niedrigere Distanz zum Referenzobjekt haben als *NearestList.MaxDist*. Die Rekursion endet, wenn keine Kindknoten zu einem Knoten gefunden werdne können, die die Distanzbedingung erfüllen.

Segment	Distanz des Objekts	Distanz des MBR
a	17	13
b	48	27
c	57	53
d	59	30
e	48	45
f	86	74
g	81	74
h	17	17
i	21	0

BR	Distanz
R0	0
R1	0
R2	0
R3	13
R4	11
R5	0
R6	44

Tabelle 1: Distanzen der Objekte und begrenzenden Rechtecke zu q

4.3 Beispiel

Eine Anfrage nach den drei nächsten Nachbarn (d.h. $k = 3$) von q (zu Distanzen siehe Tabelle 1) würde wie folgt ablaufen. [2]

k-NN(R0): *ActiveBranchList* für R0 ist (R1, R2)
    k-NN(R1): *ActiveBranchList* für R1 ist (R4, R3)
        k-NN(R4): füge d,g,h in *NearestList* ein:
        {(h,17),(d,59),(g,81): 81}
        k-NN(R3): füge a,b in *NearestList* ein: {(h,17),(a,17),(b,48): 48}
    k-NN(R2): *ActiveBranchList* für R2 ist (R5, R6)
        k-NN(R5): i ersetzt b, c ist zu distant: {(h,17),(a,17),(i,21): 21}
        k-NN(R6): wird nicht aufgerufen, da DIST(q,R6) $\geq$ 21

Da keine weiteren Verzweigungen zu näheren Nachbarn als den gefundenen führt ist der endgültige Inhalt von *NearestList*: {(h,17),(a,17),(i,21)} und wird zurückgegeben als die drei nächsten Nachbarn von q.

4.4 Optimierung

Zur Optimierung des k-Nearest-Neighbor-Algorithmus, benutzt man verschiedene Dsitanzmaße, um *ActiveBranchList* zu sortieren. Auf diese Weise können Knoten vor Eintritt in die Rekursion aus *ActiveBranchList* (downward pruning) bzw. nach ihrer Abarbeitung bei Auflösung der Rekursion (upward pruning) „beschnitten“ werden, was den Baumdurchlauf erheblich beschleunigen kann, da Knoten eventuell nicht berücksichtigt werden müssen.

Als Metrik werden vorgeschlagen (siehe auch Abb. 5):

- MINDIST: MINDIST(q, r) ist die minimale Distanz zwischen q und einem Objekt in r
- MINMAXDIST: MINMAXDIST(q, r) ist die maximale Distanz zu q bei der ein Objekt in r mit Sicherheit gefunden werden kann.

[2] $k - NN(x)$ bezeichne den Aufruf von KNEARESTTRAVERSAL mit Knoten x

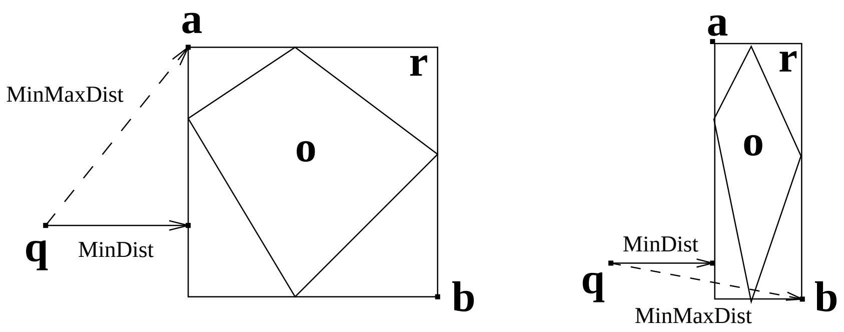

Abbildung 5: MinDist und MinMaxDist

Zur Knotenbeschneidung können dann folgende drei Strategien verwendet werden:

1. (downward pruning) Lösche Eintrag mit begrenzendem Rechteck r_1 aus *ActiveBranchList*, falls $MINDIST(q, r_1) > MINMAXDIST(q, r_2)$, wobbei r_2 ein anderes begrenzendes Rechteck in der *ActiveBranchList* ist.

2. (upward pruning) Lösche Eintrag o, falls

$$DIST(q, o) > MINMAXDIST(q, r)$$

   Dabei ist r ein begrenzendes Rechteck.

3. (up-/downward pruning) Lösche jeden Knoten aus der *ActiveBranchList* für dessen begrenzendes Rechteck r gilt:

$$MINDIST(q, r) > NearestList.MaxDist$$

Da MinMaxDist lediglich eine obere Grenze für die Distanz des q nächsten Objekts ist, sind die Strategien 1 und 2 nur für $k = 1$ von Wert. Strategie 3 hingegen ist in der **if**-Anweisung in Zeile 11 von k-NN implementiert.

5 Inkrementeller Nearest-Neighbor-Algorithmus

5.1 Pseudocode

INCNEAREST($QueryObject$, $R - tree$)

$Queue \leftarrow$ NewPriorityQueue()
Enqueue($Queue$, $R - Tree.RootNode$, 0)
while not IsEmpty($Queue$) **do**
  $Element \leftarrow$ Dequeue($Queue$)
  if $Element$ is an object or its BR **then**
    if $Element$ is the BR of $Object$ **and not** IsEmpty($Queue$) **and** Dist($QueryObject$, $Object$) $>$ First($Queue$).Key **then**

```
      ENQUEUE(Queue, Object, DIST(QueryObject, Object))
    else
      report Element (or if BR, the associated object) as the next nearest
      object
    end if
  else if Element is a leaf node then
    for each entry (Object, Rect) in leaf node Element do
      ENQUEUE(Queue, [Object], DIST(QueryObject, Rect))
    end for
  else
    for each entry (Node, Rect) in node Element do
      ENQUEUE(Queue, Node, DIST(QueryObject, Rect))
    end for
  end if
end while
```

5.2 Erläuterungen

Anstelle eines Rekursionskellers, wie er üblicherweise für eine Tiefensuche benutzt wird, verwendet der inkrementelle Nearest-Neighbor-Algorithmus eine Prioritätswarteschlange, in die sowohl Knoten (d.h. begrenzende Rechtecke) als auch Objekte eingefügt werden. Um zwischen Elementen mit gleicher Distanz unterscheiden zu können, werden Knoten vor Objekten eingefügt, während Objekte nach einer zufälligen aber eindeutigen Regel eingefügt werden. Auf diese Weise wird für bestimmte Indexstrukturen sichergestellt, daß ein Objekt nicht mehrmals als nächster Nachbar zurückgegeben wird, wenn Objekte mit mehr als einem Knoten assoziiert sind (z.B. in R^+-Bäumen). Wenn ein R-Baum als Indexstruktur benutzt wird, ist dieser Zusatz jedoch überflüssig.

Der Algorithmus geht wie folgt vor: Als erstes wird die Wurzel des Indexbaumes in die Warteschlange eingefügt. Danach wird die while-Schleife ausgeführt, die solange läuft, bis alle nächsten Nachbarn ausgegeben wurden (oder der Algorithmus abgebrochen wird). In der Schleife wird das vorderste Objekt aus der Warteschlange genommen und abhängig von der Art dieses Objektes folgt eine Fallunterscheidung:

1. Objekt ist ein innerer Knoten $\Rightarrow$ alle Kindknoten werden mit ihrer Distanz zum Referenzobjekt in die Warteschlange eingefügt

2. Objekt ist ein Blattknoten $\Rightarrow$ alle begrenzenden Rechtecke der im Knoten gespeicherten Objekte werden in die Warteschlange eingefügt

3. Objekt ist das begrenzende Rechteck eines Objekts $\Rightarrow$ das Objekt wird in die Warteschlange eingefügt, falls diese nicht leer und die Distanz des Objekts zum Referenzobjekt größer ist als die des ersten Objekts der Warteschlange

4. Objekt ist ein Datenobjekt $\Rightarrow$ Objekt wird ausgegeben

5.3 Beispiel

Mit den Distanzen aus Tabelle 1 ergibt sich für eine Query nach den drei nächsten Nachbarn folgendes (begrenzende Rechtecke von Objekten werden durch das korrespondierende Objekt in eckigen Klammern dargestellt):

1. R0 entfernen, R1 und R2 einfügen
   Queue: {(R1,0),(R2,0)}
2. R1 entfernen, R3 und R4 einfügen
   Queue: {(R2,0),(R4,11),(R3,13)}
3. R2 entfernen, R5 und R6 einfügen
   Queue: {(R5,0),(R4,11),(R3,13),(R6,44)}
4. R5 entfernen, [c] und [i] einfügen
   Queue: {([i],0),(R4,11),(R3,13),(R6,44),([c],53)}
5. [i] entfernen, DIST(q,i)>DIST(q,R4)
   ⇒ i einfügen
   Queue: {(R4,11),(R3,13),(i,21),(R6,44),([c],53)}
6. R4 entfernen, [d], [g] und [h] einfügen
   Queue: {(R3,13),([h],17),(i,21),([d],30),(R6,44),([c],53),([g],74)}
7. R3 entfernen, [a] und [b] einfügen
   Queue: {([a],13),([h],17),[i,21),[[b]27),([d],30),(R6,44),([c],53),([g],74)}
8. [a] entfernen, DIST(q,a)≤DIST(q,[h])
   ⇒ a ist nächster Nachbar
   Queue: {([h],17),(i,21),([b],27),([d],30),(R6,44),([c],53),([g],74)}
9. [h] entfernen, DIST(q,h)≤DIST(q,i)
   ⇒ h ist zweit-nächster Nachbar
   Queue: {(i,21),([b],27),([d],30),(R6,44),([c],53),([g],74)}
10. i entfernen und als dritt-nächsten Nachbar ausgeben

5.4 Varianten

Durch wenige Modifikationen läßt sich der inkrementelle Nearest-Neighbor-Algorithmus in seiner Funktionalität dahingehend ändern, daß das am weitesten entfernte Objekt gefunden wird. In diesem Fall würden die Elemente der Prioritätswarteschlange in abfallender Reihenfolge ihrer Distanz sortiert. Darüberhinaus ist es notwendig, eine zusätzliche Distanzfunktion $d_{max}(q, n)$ (maximale Distanz eines Punktes in n zu q) einzuführen. Diese Funktion muß eine Konsistenzbedingung - wie in 3.2 für d_n beschrieben - erfüllen, wobei lediglich das $\leq$ in ein $\geq$ geändert wird. Auf diese Weise wird sichergestellt, daß jeder Knoten n einen mindestens genauso weit entfernten Punkt von q hat wie alle in ihm enthaltenen Objekte.

Natürlich kann der inkrementelle Nearest-Neighbor-Algorithmus auch modifiziert werden, so daß er das k-Nearest-Neighbor-Problem löst. Dafür wird der Algorithmus lediglich dahingehend verändert, daß er terminiert sobald die k nächsten Nachbarn ausgegeben wurden.

5.5 Laufzeit

Die Laufzeit des inkrementellen Nearest-Neighbor-Algorithmus läßt sich über zwei Parameter abschätzen:

1. die erwartete Fläche der Suchregion
2. die erwartete Anzahl der besuchten Blattknoten bzw. der Objekte in der Prioritätswarteschlange (basierend auf 1. und einer angenommenen Verteilung und Größe der Blattknoten)

Für die Suche nach k nächsten Nachbarn mit einer angenommenen Anzahl von c Punkten pro Blattknoten ist die zu erwartende Überschneidung der Blattknotenregion mit der Suchregion begrenzt durch $4(2\sqrt{\frac{k}{\pi c}}-1)$. Unter der Annahme, daß sich die Hälfte der c Punkte innerhalb der Suchregion befinden ist die Anzahl der Objekte, die in der Prioritätswarteschlange verbleiben höchstens $2.26\sqrt{ck} - 2c$. Da sich innerhalb der Suchregion k Punkte befinden ist also die Anzahl der Punkte in Blattknoten, die die Suchregion schneiden durch $k + 2.26\sqrt{ck} - 2c$ begrenzt. Da nun jeder Blattknoten c Punkte enthält ergibt sich eine obere Schranke für die Anzahl der Blattknotenzugriffe von $\frac{k}{c} + 2.26\sqrt{\frac{k}{c}} - 2$

Zusammenfassend ergibt sich:

- erwarte Anzahl von Blattknotenzugriffen: $O(k + \sqrt{k})$
- erwarte Anzahl der Objekte in der Prioritätswarteschlange: $O(\sqrt{k})$

Da sich die Kosten für Einfüge- und Löschoperationen der Prioritätswarteschlange auf $O(\log m)$ bei einer Größe der Prioritätswarteschlange von m belaufen, folgt die Kostenabschätzung von $O(k + \sqrt{k}) \cdot O(log\sqrt{k}) = O(k \log k)$

5.6 Korrektheit

Es sei:

$A(o)$	-	Menge der Vorgänger von o inkl. o
R	-	Menge der ausgegebenen Objekte
Q	-	Menge der Objekte in der Warteschlange

Folgende Annahme garantiert, daß jedes Objekt zu jedem Zeitpunkt einen Repräsentanten in der Prioritätswarteschlange hat. Dies impliziert, daß jedes Objekt irgendwann ausgegeben wird.

Invariante für die äußerste while-Schleife:

Für jedes Objekt o in $SpatialIndex$ ist $A(o) \cap (Q \cup R) \neq \emptyset$

Beweis: (vollständige Induktion)

Die Invariante gilt zu Anfang, da der Wurzelknoten ein Repräsentant für alle Elemente ist.

Für $o \in R$ ist die Bedingung trivialerweise erfüllt.

Ansonsten existiert unter der Induktionsannahme $a \in A(o)$, so daß $a \in Q$. Falls a nicht als nächstes Element aus der Queue entfernt wird, bleibt die Invariante unberührt. Für den entgegengesetzten Fall ergibt sich folgende Unterscheidung:

1. Wenn $a = o$, so wird a ausgegeben und die Invariante ist erfüllt.
2. Wenn a ein Blattknoten ist, wird o in die Queue eingefügt, da $d_o(q, o) \geq d_n(q, a)$ und die Invariante ist erfüllt.
3. Wenn a ein innerer Knoten ist, werden alle Söhne in die Queue eingefügt. Da $a \in A(o)$ ist mindestens ein Sohn von a auch in $A(o)$ und die Invariante ist erfüllt.

5.7 Transformation des k-NN

Der k-Nearest-Neighbor-Algorithmus kann über einige Modifikationen in einen inkrementellen Algorithmus verwandelt werden.

Dazu verwendet man eine Liste B, die alle Instanzen der *ActiveBranchList* vereinigt, als "Karte" für den Baumdurchlauf. Auf diese Weise wird die rekursive Tiefensuche umgangen. Über eine Kombination dieser Liste mit *NearestList* erreicht man eine Funktionalität ähnlich der *PriorityQueue* des INN, wobei jedoch immer noch alle Einträge in der Liste gehalten werden müssen.

6 Experimente

Zum Vergleich der vorgestellten Algorithmen wurde als Index anstelle eines R-Baums ein R*-Baum - der eine Erweiterung des R-Baums ist - verwendet. Die Performance-Tests wurden unter anderem für zwei Karten durchgeführt, eine reale Karte von Prince George's County mit 59551 Liniensegmenten und eine zufällig erzeugte Karte mit 64000 Liniensegmenten.

Die durchgeführten Experimente ergaben:

- INN hat geringere Ausführungszeiten, Disk I/Os und Distanzkalkulationen als k-NN.
- Die Optimierung des k-NN (siehe 4.4) verbesserte zwar dessen Performance, änderte aber nichts an der Tatsache, daß der INN bessere Performance zeigt.
- Sogar für das k-Nearest-Neighbor-Problem zeigte der INN bessere Performance als der k-NN.

6.1 Spezielle Datenmengen

Außer den bisher erwähnten durchgeführten Tests wurden die vorgestellten Algorithmen auch noch für spezielle Datenmengen getestet. Nachfolgende Testergebnisse zeigen exemplarisch den Vergleich der Algorithmen für große Datenmengen (siehe 6.1.2), wobei NN die Anzahl der berechneten nächsten Nachbarn bezeichnet.

log(NN)	INN	k-NN
1	80	160
2	100	300
3	190	930
4	900	5000
5	8000	30000
6	90000	260000
7	750000	N/A

log(NN)	INN	k-NN
1	20	170
2	80	310
3	170	1000
4	900	3500
5	7000	20000
6	60000	110000
7	500000	N/A

Tabelle 2: Ausführungszeit (links) und R-Baum Disk I/Os (rechts)

log(NN)	INN	k-nn
1	200	1900
2	900	4300
3	2200	14000
4	13000	60000
5	120000	330000
6	1e+06	2e+06
7	9e+06	N/A

Tabelle 3: Distanzkalkulationen

6.1.1 Variierende Datenmengen

Für variierende Datenmengen wurden vier reale Karten und neun zufällig erzeugte Karten zwischen 1000 und 256000 Liniensegmenten verwendet.

Auch hier zeigte sich das Leistungsverhältnis, das obige Experimente ergaben. Der k-NN war zwischen 10% und 19% langsamer als der INN.

6.1.2 Große Datenmengen

Um die beiden Nearest-Neighbor-Algorithmen an einer großen Datenmenge zu testen, wurde eine Karte mit 8 Millionen Linien zufällig generiert.

Der k-Nearest-Neighbor-Algorithmus konnte für $k = 8$ Millionen nicht mehr getestet werden, da nicht genügend Speicher zur Verfügung stand, um die *Nearest-List* mit 8 Millionen Einträgen im Speicher zu halten. Im Gegensatz dazu wuchs die *PriorityQueue* des INN auf maximal 83000 Elemente, also etwa 1% der Anzahl von nächsten Nachbarn insgesamt (auch das am weitesten entfernte Objekt ist irgendwann ein nächster Nachbar).

6.1.3 Hochdimensionaler Raum

Der INN ist unabhängig von der Dimension des Datenraums. Hier sind lediglich durch die verschiedenen Indexstrukturen Grenzen gesetzt. Beispielsweise degenerieren R-Bäume für Dimensionen größer als 7. Auch andere Indexstrukturen

(X-Baum, LSD^h-Baum) ermöglichen oberhalb von Dimensionen von 20 kaum Geschwindigkeitsgewinne gegennüber einer sequentiellen Suche.

Ein weiteres Problem ist die Wahl der Suchregion, die durch einen Radius begrenzt ist und je nach Dimension ein Kreis, eine Kugel oder eine Hyper-Sphäre beschreibt. Dem entgegen stehen die üblichen räumlichen Indexstrukturen, deren Knotenabdeckung sich meist auf Hyper-Rechtecke bezieht. So liegt das Verhältnis eines Kreises zu einem Quadrat bei ca. 79%, während es bei einer Kugel und einem Kubus nur noch bei 52% liegt. In vier Dimensionen ist das Verhältnis zwischen einer Hyper-Sphäre und einem Hyper-Kubus nur noch bei 10%. Das hat zur Folge, daß Punkte innerhalb des Hyper-Kubus aber außerhalb der Suchregion (der Hyper-Sphäre) in die Nearest-Neighbor-Suche unnötigerweise mit einbezogen werden.

Zur Lösung dieses Problesm wurde versucht, näherungsweise nächste Nachbarn zu berechnen, was sich in Experimenten als durchaus praktikabel erwies. Der durchschnittliche relative Fehler belief sich auf 10%.

7 Zusammenfassung

Zusammenfassend läßt sich feststellen, daß der inkrementelle Nearest-Neighbor-Algorithmus in jedem Fall dem k-Nearest-Neighbor-Algorithmus vorzuziehen ist. Auch für spezielle Datenmengen eignet sich der INN sehr gut, insbesondere durch seine flexible Anpassungsmöglichkeit an verschiedene Indexstrukturen. Jedoch sind dem Algorithmus im hochdimensionalen Raum durch die gegebenen Indexstrukturen Grenzen gesetzt, die nur durch neue (bessere) Indexstrukturen zu überschreiten sind.

Literatur

[1] P.M.G. Apers, H.M. Blanken, M.A.W. Houtsma. Multimedia databases in perspective, Springer Verlag London Ltd. 1997

[2] V. Gaede, O. Günther. Multidimensional Access Methods

[3] A. Guttmann. R-trees: A dynamic index structure for spatial searching. Proceedings of the ACM Sigmod Conference, S. 47-57, Boston, MA, Juni 1984

[4] G.R. Hjaltson, H. Samet. Distance Browsing in spatial Databases

[5] G. Weikum. Skript zur Vorlesung Datenbanksysteme, 1998