

Studiengang: Informatik
Betreuer: Thomas Schwarz
Prüfer Prof. B. Mitschang

Beginn am: 17.02.2003
Beendet am: 16.08.2003

CR-Klassifikation: C.2.4, H.2.8, H.3.3

Studienarbeit Nr. 1889
Föderierte Nearest-Neighbor-Queries
Markus Iofcea

Fakultät Elektrotechnik, Informatik, Informationstechnik
Universität Stuttgart
Institut für Parallele und Verteilte Systeme (IPVS)
Abteilung Anwendersoftware (AS)
Universitätsstr. 38
70569 Stuttgart

Zusammenfassung

k nächste Nachbar (kNN) Anfragen werden in geographischen Informationssystemen häufig genutzt, um die räumlich nächstgelegenen Objekte zu einem Anfrageort zu finden. Im Rahmen dieser Studienarbeit wurde ein Algorithmus entworfen, der das Problem der k nächsten Nachbar Anfragen in geographischen Informationssystemen mit verteilter Datenhaltung löst. Die Daten sind in diesem Fall nur über eine Anfrageschnittstelle für den Algorithmus erreichbar. Es gibt keinen lokalen und direkten Zugriff auf die Indexstrukturen der räumlichen Datenbanken, was die meisten bisherigen kNN Algorithmen benötigen.

Der hier entworfene Algorithmus soll in die Nexus Plattform, einer offenen Plattform für ortsbezogene Dienste, die an der Universität Stuttgart entwickelt wird, eingebettet werden. Neben der ausführlichen Beschreibung des Algorithmus werden Anwendungsbereiche, die Schwierigkeiten, die bei dem Entwurf eines solchen Algorithmus auftreten, und Erweiterungsmöglichkeiten erläutert.

Inhaltsverzeichnis

1. Einleitung	9
2. Einführung in die Nearest Neighbor Queries	11
2.1. Nearest Neighbor Queries	11
2.1.1. Allgemein	11
2.1.2. Nächste Nachbar Suche mit Hilfe von R-Bäumen	11
2.2. Nearest Neighbor Queries für bewegliche Anfragepunkte	15
2.2.1. Allgemein	15
2.2.2. Idee	15
2.3. Constrained Nearest Neighbor Queries	21
2.3.1. Einführung	21
2.3.2. 2-Phasen Algorithmen für CNN Anfragen nach [6]	22
2.3.3. Optimierter 1-Phasen Ansatz nach [6]	24
2.4. Föderierte Nearest Neighbor Queries	27
3. Der Algorithmus	29
3.1. Nexus	29
3.2. Der Algorithmus	31
3.2.1. Es werden $n \geq k$ Ergebnisse gefunden	32
3.2.2. Es werden $0 < n < k$ Ergebnisse gefunden	34
3.2.3. Es werden keine Ergebnisse gefunden	35
3.2.4. Terminierung in den Fällen $0 < n < k$ und $n = 0$	36
3.3. Optimierungen	38
3.3.1. Bestimmung der Puffergröße für die erste Iteration	38
3.3.2. Mehrere Server gleichzeitig abfragen	39
3.4. Korrektheit	41
4. Die Implementierung	42
4.1. Benötigte Komponenten	42
4.2. Durchgeführte Anpassungen an der Nexus Föderation	42
4.3. Implementierung des Algorithmus	42
4.3.1. Testdatengenerator	43
4.3.2. ASR	44
4.3.3. SpaSe	45

4.3.4. FNNQ	46
4.4. Evaluierung	48
4.5. Optimierungsvorschläge für die Implementierung	51
5. Schlussbemerkungen	52
A. Begriffslexikon	54
B. Verwendete Abkürzungen	55

Abbildungsverzeichnis

1.	MRH Beispiel (Abb. nach [3])	12
2.	R-Baum zu Abbildung 1 (Abb. nach [3])	12
3.	<i>mindist</i> und <i>minmaxdist</i> im 2-dimensionalen Raum (Abb. nach [3])	13
4.	<i>mindist</i> führt zu teurer Traversierung (Abb. nach [3])	14
5.	ϵ_{t+1} muss nicht unendlich groß sein (Abb. in Anlehnung an [4])	16
6.	Pre-fetching Algorithmus (Abb. in Anlehnung an [4])	20
7.	CNNQ Beispiel mit einem Suchgebiet	22
8.	CNNQ mehrere Suchgebiete	23
9.	CNN Ranged Anfrage	24
10.	für einzuschließende MRH (Abb. nach [4])	24
11.	Modifizierte <i>mindist</i> (q, M) und <i>minmaxdist</i> (q, M)	25
12.	verteilte Datenhaltung mit Föderation	27
13.	Nexus Grobübersicht	29
14.	Initialer Schritt	31
15.	$n \geq k$	32
16.	$n \geq k$ Schleife	34
17.	$dist(x_{n_i}, q)$ ist größer als r_i	36
18.	<i>GenerateTestdata</i> - Testdatengenerator für den Algorithmus	44
19.	<i>ASR</i>	45
20.	<i>SpaSe</i>	46
21.	<i>FNNQ</i>	46
22.	Durchschnittliche Anzahl besuchter Server und zurückgelieferter Objekte bei 1000 AAs, 10 numNN und variabler Anzahl von Testobjekten	49
23.	Durchschnittliche Anzahl besuchter Server und zurückgelieferter Objekte bei 10.000 Testobjekten, 10 numNN und variabler Anzahl von AAs	50
24.	Durchschnittliche Anzahl besuchter Server und zurückgelieferter Objekte bei 5.000 Testobjekten, 1.000 AAs und variabler Anzahl von numNN	51

1. Einleitung

Hintergrund

Das Nexus Projekt [12] ist eine offene Plattform für ortsbezogene Dienste, die an der Universität Stuttgart entwickelt wird. Die Plattform kann eine Vielzahl von verschiedenen ortsbezogenen Informationen halten, die den Anwendungen einheitlich präsentiert werden. Nexus sorgt durch einheitliche Schnittstellen dafür, dass der Datenzugriff für den Anwendungsprogrammierer transparent bleibt und die Heterogenität der Daten nicht zu einer zusätzlichen Belastung für den Programmierer wird.

Eine zentrale Komponente von Nexus ist die Föderation. Die Nexus Föderation ist die Middleware-Schicht zwischen Client-Anwendungen und den Servern der Dienstleister, die ortsbezogenen Informationen zur Verfügung stellen. Die Daten sind auf verschiedene Anbieter verteilt, die wiederum nur die von Ihnen erhobenen Daten zur Verfügung stellen. Die Föderation leitet die Anfragen von den Client-Anwendungen zu den Servern der Dienstleister, kombiniert die Daten der verschiedenen Anbieter und sendet deren Antwort gebündelt zurück an die Client-Anwendung. So wird die Verteilung der Daten für die Client-Anwendungen transparent.

Motivation

Ein häufig angetroffener Anfragetyp in geographischen Informationssystemen, wie zum Beispiel der Nexus Plattform, ist die k -Nearest Neighbor (kNN) Query [3]. Bei einer kNN Anfrage werden die k nächsten Nachbarn zu einem Anfrageort q gesucht. kNN Anfragen sehen beispielsweise folgendermaßen aus und können beliebig komplex sein:

- Welche sind die nächstgelegenen 5 Tankstellen?
- Welche sind die nächstgelegenen 3 italienischen Restaurants, die einen Festsaal haben und deren Speisekarte mindestens 30 Gerichte umfasst?

Die meisten Algorithmen, die die k nächsten Nachbarn berechnen (z.B. [1, 3]) benötigen lokalen Zugriff auf die räumliche Datenbank und deren räumliche Indices (oft als R-Bäume realisiert, Vgl. [10]). Bei einem verteilten System, wie der Nexus Plattform, ist dies nicht möglich da die Daten auf mehrere räumliche Server verteilt sind, die jeweils eine vollkommen autonome Datenhaltung haben.

Der naive Ansatz wäre, die Daten aller Server auf einen einzigen Server zu spiegeln, welcher ohne Netzwerkzugriff erreicht werden kann und auf dessen Daten ein Algorithmus direkt zugreifen kann. Durch die redundante Datenhaltung muss jedoch mit Geschwindigkeitseinbußen während der Spiegelungsphase und hohem Speicherverbrauch gerechnet werden, die das System ineffizient machen und dessen Skalierbarkeit einschränken. Zusätzlich geht dabei die Autonomie der Server verloren.

Ziel dieser Arbeit

Ziel dieser Arbeit ist es, im Rahmen des Nexus Projekts einen Algorithmus zu entwerfen, der das Problem der kNN Anfragen für eine verteilte Datenhaltung auf entfernten Servern löst. Der Algorithmus soll später in die Nexus-Föderation eingebettet werden und durch die Nexus-Anfrageschnittstelle den Anwendungsprogrammierern zur Verfügung gestellt werden.

Der hier vorgestellte Algorithmus, basiert auf einer oder mehreren Gebietsanfragen an den Nexus Verzeichnisdienst (Area Service register, ASR) um die k nächsten Nachbarn zu einem gegebenen Punkt zu finden. Die Gebietsanfragen an das ASR sollen so formulieren werden, dass die Anzahl der Iterationen und überflüssigen Ergebnisse gering gehalten wird. Die Schwierigkeit dabei ist, dass das Zielgebiet, in dem sich die k nächsten Nachbarn befinden, nicht bekannt ist.

Gliederung der Ausarbeitung

Die verbleibenden Kapitel dieser Ausarbeitung sind folgendermaßen gegliedert. In Kapitel 2 werden Grundlagen zu kNN Anfragen erläutert. In Kapitel 3 wird ein Algorithmus vorgestellt, der das Problem der föderierten nächsten Nachbarn Anfragen (FN-NQ) löst, dessen Randbedingungen besprochen und die Korrektheit gezeigt. In Kapitel 4 wird eine Implementierung des Algorithmus vorgestellt, eine Evaluierung durchgeführt und weitere Optimierungen vorgeschlagen. Schließlich werden in Kapitel 5 weitere Möglichkeiten der kNN Anfragen in Nexus diskutiert.

2. Einführung in die Nearest Neighbor Queries

2.1. Nearest Neighbor Queries

2.1.1. Allgemein

Es gibt eine Menge S von räumlichen Objekten und eine Abstandsfunktion d . Eine einfache kNN Anfrage soll nun zu einem gegebenen Anfrageort q die k nächsten Nachbarn finden, also die k Objekte mit dem kleinsten $d(q, o)$ mit $o \in S$.

Formal: Gesucht wird eine Menge T , der Größe k , von Objekten die sich am nächsten zu q befinden.

$$T(q, k, S) = \{o_1 \dots o_k \mid d(q, o_i) \leq d(q, o') \forall i, 1 \leq i \leq k, o_i \in S \text{ und } \forall o' \in S \setminus \{o_1 \dots o_k\}\}$$

In Worten: Die Distanz $d(q, o_i)$ zwischen dem Anfragepunkt q und den k Objekten in der Menge T muss kleiner sein als die Distanz $d(q, o')$ zwischen dem Anfragepunkt q und allen Objekten o' aus der Menge S ohne die k nächsten Nachbarn aus der Menge T .

2.1.2. Nächste Nachbar Suche mit Hilfe von R-Bäumen

Bei dem hier vorgestellten Ansatz zur Auffindung der nächsten Nachbarn (nach [3]) werden R-Bäume [9, 10] als Indexstruktur für die Objektzugriffe verwendet. R-Bäume bleiben balanciert wie B-Bäume und passen sich dynamisch den Gegebenheiten an. B-Bäume werden als bekannt vorausgesetzt.

Ein R-Baum besteht aus einer Menge von minimalen rechteckigen Hüllen (MRH), welche wiederum eine Menge von Objekten oder weiteren MRHs komplett umhüllt.

Eine MRH heißt minimal, weil jede Kante der MRH mindestens ein Objekt enthalten muss. Wäre dies nicht der Fall wäre das Rechteck nicht minimal und könnte noch weiter verkleinert werden (vgl. Abbildung 1).

Jeder interne Knoten im R-Baum hält nur Referenzen auf die Tochterknoten. Die eigentlichen Datenobjekte sind in den Blattknoten gespeichert (vgl. Abbildung 2). Die Objekte in den Blattknoten werden als atomar angesehen, auch wenn sie aus mehreren geometrischen Primitiven (Rechtecke, Kreise, Dreiecke, Linien) zusammengesetzt sein können.

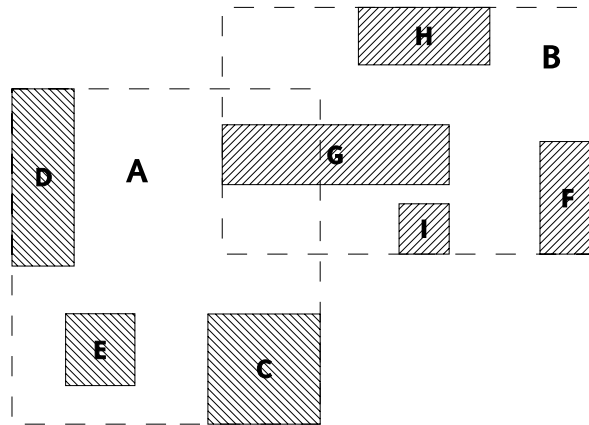


Abbildung 1: MRH Beispiel (Abb. nach [3])

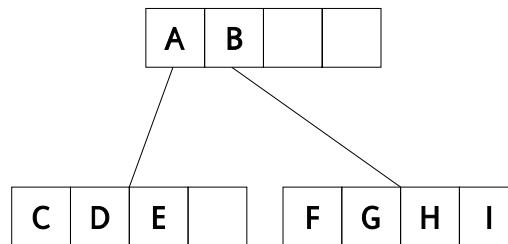


Abbildung 2: R-Baum zu Abbildung 1 (Abb. nach [3])

Metriken für die 1-NN Suche

Die erste Metrik ist *mindist*. Diese Metrik beschreibt die minimale Distanz zwischen dem Anfrageort q und einem Objekt o in einer MRH M . Befindet sich der Anfrageort q innerhalb von M , so ist $mindist(q, M) = 0$. Wenn q außerhalb von M ist, so ist $mindist(q, M)$ der euklidischer Abstand zwischen q und der nächstgelegenen Kante von M . Bei dieser Metrik wird davon ausgegangen, dass sich der nächste Nachbar nicht viel weiter entfernt ist als *mindist*.

Beim traversieren des R-Baumes muss an jedem inneren Knoten entschieden werden welche MRH als nächstes durchsucht werden soll. Hier kann *mindist* als erste Annäherung benutzt werden. Die Traversierung mittels der *mindist* Metrik darf aber nicht überbewertet werden, denn der nächste Nachbar kann durchaus viel weiter entfernt sein als *mindist*. Durch leeren Raum kann es oft zum Verwerfen von Ergebnissen kommen, wenn immer die MRH mit dem kleinsten *mindist* als nächstes durchsucht wird.

Die zweite Metrik ist $minmaxdist$. $minmaxdist(q, M)$ ist das Minimum, über n Dimensionen, aller maximalen Distanzen zwischen der Anfrageposition q und Punkten auf den Kanten der MRH (vgl. Abbildung 3). Die maximale Distanz zwischen q und einer Kante auf einer der n Dimensionen ist der Eckpunkt auf der Kante, der am weitesten von q entfernt ist. Es werden alle maximalen Distanzen zwischen q und den Kanten berechnet und die kleinste dieser maximalen Distanzen als $minmaxdist$ ausgewählt. $minmaxdist$ garantiert, dass es mindestens ein Objekt innerhalb eines MHRs gibt, das höchstens $minmaxdist$ weit entfernt von q ist. $minmaxdist$ kann dazu beitragen, dass unnötige Zugriffe auf MRH vermieden werden. Dadurch können die Suchpfade verkürzt werden. So können zum Beispiel alle MRHs weggelassen werden, deren $mindist$ größer ist als $minmaxdist$ eines anderen MRH.

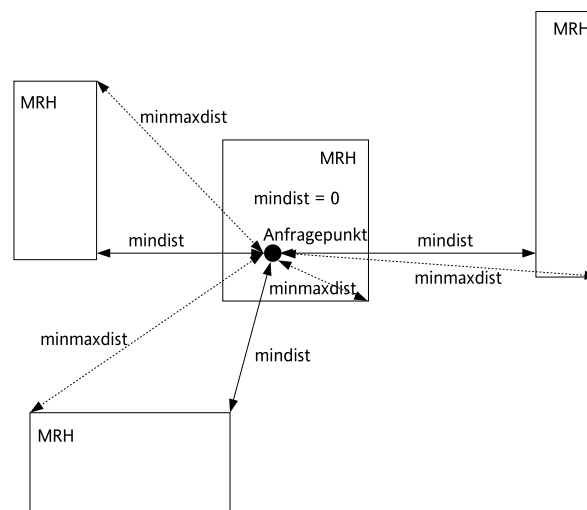


Abbildung 3: $mindist$ und $minmaxdist$ im 2-dimensionalen Raum (Abb. nach [3])

$mindist$ und $minmaxdist$ stellen die untere bzw. die obere Grenze für die Distanz zwischen q und o dar und werden dazu benötigt die Suchpfade effizient zu verkürzen und die Suchreihenfolge bei der Traversierung zu ordnen.

1-NN Algorithmus für R-Bäume

Als Heuristik zur Bestimmung der Suchreihenfolge, bei der Suche nach dem nächsten Nachbarn, dienen bei diesem Ansatz die beiden oben vorgestellten Metriken $mindist$ und $minmaxdist$. $mindist$ führt zu einer optimistischen Suchreihenfolge und $minmaxdist$

zu einer pessimistischen, jedoch nicht dem Worst Case. Da aber nicht allein der Abstand zwischen Anfrageort und MRH entscheidend für die Bestimmung der Suchreihenfolge ist, sondern auch die Größe und der Aufbau der MRHs, ist hier zu beachten, dass man Fälle konstruieren kann bei denen die *mindist* Metrik zu einer sehr ineffizienten Baum-Traversierung führen kann (vgl. Abbildung 4).

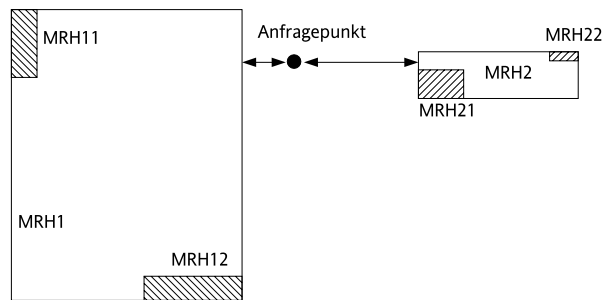


Abbildung 4: *mindist* führt zu teurer Traversierung (Abb. nach [3])

Suchreihenfolge (Tiefensuche):

- *mindist* würde bei dem Beispiel in Abbildung 4 zu folgender Suchreihenfolge führen: Zuerst wird MRH1 besucht, dann MRH11, MRH12, MRH2 und MRH21 bevor der nächste Nachbar gefunden wird.
- Die *minmaxdist* Metrik würde zu folgender Suchreihenfolge führen: MRH2 wird zuerst besucht und dann MRH21. Wenn MRH1 noch besucht wird, können MRH11 und MRH12 gleich verworfen werden.

Verkürzung der Suchpfade: weiterhin werden in [3] drei Strategien vorgeschlagen um die Suchpfade zu verkürzen.

1. eine MRH M mit $mindist(q, M') \geq minmaxdist(q, M)$ mit $M' \in (\text{noch nicht besuchte MRH})$ kann verworfen werden. M' kann den nächsten Nachbarn nicht enthalten.
2. ein Objekt o dessen Distanz zu q größer ist als $minmaxdist(q, M)$ kann verworfen werden, da M ein Objekt o' enthält welches näher an q ist als o .
3. jede MRH M kann verworfen werden deren $mindist(q, M)$ größer ist als $dist(q, o)$ des schon gefundenen nächsten Nachbarn.

2.2. Nearest Neighbor Queries für bewegliche Anfragepunkte

2.2.1. Allgemein

Hier soll eine Erweiterung für die Suche nach den nächsten Nachbarn vorgestellt werden.

In der Praxis kommt es oft vor, dass der Anfrageort q sich bewegt (z.B. ein fahrendes Auto). Die nächste Nachbar Anfrage für bewegliche Anfragepunkte wird im Folgenden k -NNBA genannt.

Die Aufgabe ist nun diese beweglichen Objekte zu verfolgen und deren Position in einer Datenbank zu speichern. Der konventionelle Ansatz wäre es die Position aller beweglichen Objekte in gewissen Zeitintervallen zu speichern. Dieser Ansatz ist aber aus folgenden Gründen nicht besonders effizient:

- zu häufige Aktualisierungen überfordern die Datenbank. Es würde zu Verzögerungen kommen.
- bei zu wenigen Aktualisierungen würde die Aktualität leiden.

Um die Anzahl der Updates zu reduzieren und trotzdem einigermaßen aktuell zu bleiben wird in [2] vorgeschlagen, die beweglichen Objekte als Funktionen der Zeit zu modellieren. Updates sind nur dann nötig, wenn sich die Parameter der Funktion signifikant ändern.

2.2.2. Idee

Bei diesem Ansatz, nach Z. Song und N. Roussopoulos [4], wird davon ausgegangen, dass die Ergebnismengen T_1 und T_2 von zwei Anfrageorten q_1 und q_2 aus einer k -NNBA voneinander abhängen, wenn die zwei Anfrageorte nahe beieinander liegen. Wenn wir die Ergebnismenge T_1 für den Anfrageort q_1 haben, soll mit wenig Mehraufwand T_2 , für den Anfrageort q_2 , berechnet werden. Auch hier werden die Objekte in R-Bäumen organisiert.

Grundsätzlich soll hier also keine neue effiziente räumliche Indexstruktur eingeführt werden sondern es wird vielmehr versucht sich die Information aus der alten Ergebnismenge, von der letzten Anfrage, zu nutzen und diese weiterhin zu verwenden. Es kann sogar vorkommen, dass man die Ergebnismenge für den neuen Punkt komplett aus der Historie berechnet werden kann, ohne neue Berechnungen zu starten. Die Nebenabrede

hier ist natürlich, dass die schon vorhandene Ergebnismenge nicht völlig veraltet ist. Wenn aber doch, so lässt sich eine komplett neue Suche nicht vermeiden. Wenn die nächste Position präzise vorhergesagt werden kann, kann dieser Ansatz noch weiter optimiert werden indem er um einen Puffer erweitert wird. Doch dazu später mehr.

q ist der sich bewegende Punkt und q_t ist seine Position zu der Zeit t . Die Objekte werden aufsteigend, nach der Distanz zu q_t , sortiert in T gespeichert. ϵ_t ist der kleinste Radius um q_t , in dessen Kreisfläche garantiert k Objekte gefunden werden können.

Feste obere Grenze Algorithmus (Fixed Upper Bound)

Die initiale Suche wird mit einem statischen kNN Algorithmus (wie der oben beschriebene Algorithmus nach [3]) mit unendlich großem Suchraum durchgeführt. In Abbildung 5 betrachten wir zunächst q zum Zeitpunkt t . Innerhalb des durchgezogenen Kreises befinden sich k Objekte. Zur Zeit $t + 1$ befindet sich q an der Position q_{t+1} . ϵ_{t+1} ist gültig wenn in dessen Kreisfläche k Objekte sind. Diese Bedingung ist leicht zu erfüllen, wenn der durchgezogene Kreis eingeschlossen wird.

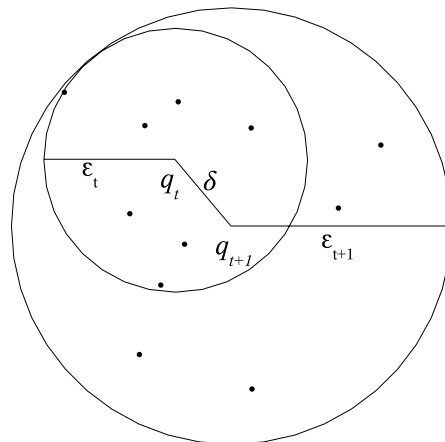


Abbildung 5: ϵ_{t+1} muss nicht unendlich groß sein (Abb. in Anlehnung an [4])

Theorem 1. Angenommen q_t sei der Anfrageort zur Zeit t und die k nächsten Nachbarn seien $\{p_1, p_2, \dots, p_k\}$ und $D_t(k)$ sei die Distanz zu dem am weitesten von q_t entfernten nächsten Nachbar. δ ist der Abstand zwischen q_t und q_{t+1} . Zum Zeitpunkt $t + 1$ gilt $\epsilon_{t+1} = D_t(k) + \delta$.

Beweis. Per Definition haben wir:

$$\text{dist}(p_i, q_t) \leq D_t(k) \quad (\forall i \leq k)$$

ferner gilt durch die Dreiecksungleichung:

$$\text{dist}(p_i, q_{t+1}) \leq D_t(k) + \delta \quad (\forall i \leq k)$$

somit ist gezeigt, dass sich mindestens k Objekte in einem Umkreis von $\epsilon_{t+1} = D_t(k) + \delta$ um q_{t+1} befinden.

■

Der Algorithmus arbeitet folgendermaßen. Er führt zum Zeitpunkt $t = 1$ eine initiale, statische Suche durch. Dabei wird die Position q_1 gespeichert, sowie $D_1(k)$. Zu späteren Zeitpunkten $t > 1$ wird jeweils mit Hilfe von $D_{t-1}(k)$ das nächste Suchgebiet bestimmt und wieder eine statische Suche durchgeführt. Dabei wird $D_t(k)$ und q_t für die nächste Suche gespeichert.

Lazy Suchalgorithmus

Beim vorherigen Algorithmus wird an jeder neuen Anfrageposition eine neue Suche initiiert. Das ist in manchen Fällen, wie zum Beispiel wenn sich die Position q_{t+1} nur wenig verändert hat im Vergleich zur Position q_t , sehr ineffizient. Es soll nun gezeigt werden, dass in diesen Fällen eine neue Suche unnötig ist.

Ausgangspunkt für die neue Überlegung beim Lazy Algorithmus ist die Tatsache, dass $D_t(k+1)$ mit wenig mehr Aufwand berechnet werden kann. Es ist sogar so, dass bei vielen NN Algorithmen mehr als k Objekte gefunden werden, die später verworfen werden, weil nur die k -nächsten gefragt sind. Also muss einfach nur der $k+1$ -te nächste Nachbar zusätzlich gespeichert werden. Nachdem $D_t(k+1)$ als bekannt vorausgesetzt wird, wird folgende These aufgestellt:

Theorem 2. Angenommen zum Zeitpunkt t seien die k -nächsten Nachbarn $\{p_1, p_2, \dots, p_k\}$ und $D_t(k)$ bekannt. Zusätzlich ist $D_t(k+1)$ bekannt. Es wird nun angenommen, dass sich die Ergebnismenge bei q_{t+1} im Vergleich zu q_t nicht verändert, wenn folgendes gilt:

$$\delta \leq \frac{D_t(k+1) - D_t(k)}{2}$$

wobei δ der Abstand zwischen q_t und q_{t+1} ist.

Beweis. Per Definition gilt:

$$\text{dist}(p_i, q_t) \leq D_t(k) \quad (\forall i \leq k)$$

Wenn der Anfragepunkt sich nach q_{t+1} bewegt gilt:

$$\text{dist}(p_i, q_{t+1}) \leq D_t(k) + \delta \quad (\forall i \leq k)$$

Für alle Objekte p die nicht in der Ergebnismenge enthalten sind gilt:

$$D_t(k+1) \leq \text{dist}(p, q_t)$$

$$\text{dist}(p, q_t) \leq \text{dist}(p, q_{t+1}) + \delta$$

$$D_t(k+1) - \delta \leq \text{dist}(p, q_{t+1})$$

Wenn $\delta \leq \frac{D_t(k+1) - D_t(k)}{2}$ für alle Objekte p welche nicht in der Ergebnismenge sind gilt, dann gilt folgendes:

$$\text{dist}(p_i, q_{t+1}) \leq D_t(k) + \delta \leq D_t(k+1) - \delta \leq \text{dist}(p, q_{t+1}) \quad (\forall i \leq k)$$

Das bedeutet, dass die Ergebnismenge $\{p_1, p_2, \dots, p_k\}$ immer noch korrekt ist. ■

Oben wurde ein sehr intuitive Tatsache dargestellt. Nämlich, dass wenn sich die Anfrageposition nur gering ändert, die Ergebnismenge sich nicht verändert.

Wenn sich die Anfrageposition signifikant verändert hat und eine neue Suche nach den k nächsten Nachbarn gestartet werden muss, ist es von Vorteil die Ergebnisse aus vorhergehenden Suchen wiederzuverwenden. Dazu wird folgende These aufgestellt:

Theorem 3. Die Ergebnismenge zum Zeitpunkt $t+1$ enthält die Objekte p_i aus der Ergebnismenge zum Zeitpunkt t , welche folgende Ungleichung erfüllen:

$$D_t(i) \leq D_t(k+1) - 2\delta$$

Beweis. $\text{dist}(p_i, q_{t+1}) \leq D_t(i) + \delta \leq D_t(k+1) - \delta \leq D_{t+1}(k+1)$ ■

Der Algorithmus geht folgendermaßen vor.

1. Zu Beginn wird die Anfrageposition, die Ergebnismenge, $D_1(k)$ und $D_1(k+1)$ (der Abstand zum nächsten Punkt außerhalb der Ergebnismenge der NN Anfrage) gespeichert.
2. Zu jedem späteren Zeitpunkt wird zuerst geprüft ob die alte Ergebnismenge immer noch korrekt ist (Theorem 2). Wenn ja kann der Algorithmus stoppen.
3. Wenn δ zu groß ist muss eine neue Suche initiiert werden. Es muss aber zuerst geprüft werden, welche Objekte aus der vorangehenden Anfrage übernommen werden können (Theorem 3). Dann wird ϵ_t (Theorem 1) für die Suche neu berechnet und eine statische NN Anfrage gestartet.
4. Nachdem die Anfrage durchgeführt wurde und eine neue Ergebnismenge berechnet wurde, muss diese nun in einem Puffer samt $D_t(k)$ und $D_t(k+1)$ für die nächste Anfrage gespeichert werden.

Pre-fetching Suchalgorithmus

Die Grundüberlegung hier ist, dass zu Beginn eine m -nächste Nachbarn Anfrage mit $m > k$ durchgeführt wird. Die Ergebnisse werden in einem Puffer im Speicher abgelegt. Zu späteren Zeitpunkten soll dann nur noch die Pufferinformation ausreichen um die k -nächsten Nachbarn zu berechnen, ohne eine neue Berechnungen zu starten. Es muss nun überlegt werden, wie man den Puffer für zukünftige Anfragen aktuell halten kann, sprich wann der Puffer aktualisiert werden muss.

Theorem 4. Der Puffer, welcher die m -nächsten Nachbarn hält, muss erst aktualisiert werden, wenn,

$$\delta \leq \frac{D_t(m) - D_t(k)}{2}$$

wobei δ der Abstand zwischen q_t und der neuen Position q' ist.

Beweis. Wie in Theorem 1 gezeigt ist $\epsilon(q')$ gültig, wenn $\epsilon(q') \geq D_t(k) + \delta$. Wie in Abbildung 6 zu sehen, muss der Puffer nicht aktualisiert werden, wenn der gepunktete Kreis sich vollständig im gestrichelten Kreis befindet, der alle Objekte im Puffer enthält. Dies ist der Fall, wenn $\epsilon(q') \leq D_t(m) - \delta$.

Das Theorem 4 sagt uns also wann der Puffer aktualisiert werden muss. Bei genauer Betrachtung sehen wir, dass der Lazy Algorithmus ein Spezialfall des Pre-fetching Algorithmus ist, nämlich mit $m = k + 1$. Der Algorithmus arbeitet folgendermaßen:

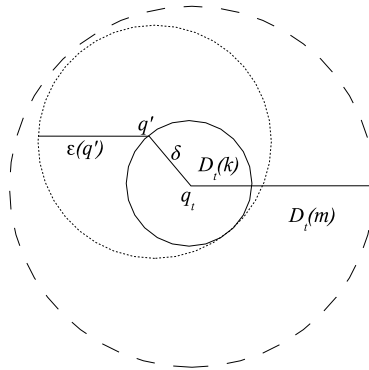


Abbildung 6: Pre-fetching Algorithmus (Abb. in Anlehnung an [4])

1. Zu einer gegebenen Position 1 werden die m nächsten Nachbarn mit Hilfe eines NN Algorithmus bestimmt und in einem Puffer zwischengespeichert. Darunter befinden sich auch die k nächsten Nachbarn. Außerdem wird noch $D_1(k)$ und $D_1(m)$ gespeichert.
2. Zu einem späteren Zeitpunkt $t > 1$ wird zuerst geprüft ob die gesuchten Ergebnisse in dem Puffer sind (Theorem 4).
3. Wenn die Ergebnisse in dem Puffer sind müssen sie nur noch aus dem Puffer berechnet werden. Wenn nicht, wird eine neue Berechnung von m nächsten Nachbarn angestoßen und $D_t(k)$ und $D_t(m)$ gespeichert.

Der Tradeoff hier ist die Puffergröße.

2.3. Constrained Nearest Neighbor Queries

2.3.1. Einführung

Eine weitere Form der Nearest Neighbor Queries in GeoInformationssystemen (GIS) sind die Constrained Nearest Neighbor Queries (CNNQ). Diese Art der Nearest Neighbor Queries werden in [6] im Detail beschrieben. Hier soll nun ein grober Überblick gegeben werden.

Bei den CNNQ ist der Suchraum, in dem nach Kandidaten gesucht wird, auf einen bestimmten Teil des geographischen Gebietes beschränkt, das bestimmte vom Benutzer vorgegebenen Kriterien erfüllt. Es muss also nicht der gesamte Datenraum nach den k -nächsten Nachbarn durchsucht werden.

Auch bei den CNNQ wird großen Wert auf möglichst wenig Ein-/Ausgabeoperationen während der Abarbeitung der Anfrage gelegt. Zusätzlich sind Strategien notwendig um den Suchraum zu verkleinern (Vgl. [3]) und um unnötige Ergebnisse zu vermeiden.

CNN Anfragen können folgendermaßen lauten:

- Welche Werkstatt ist die nächstgelegene im Nordosten zu der aktuellen Position p ?
- Welche Tankstelle ist die nächstgelegene im südlichen Stadtteil von Stuttgart?

Das Ergebnis einer solchen Anfrage ist das nächstgelegene Objekt aus dem eingeschränkten Suchgebiet, welches die Anfragekriterien erfüllt.

Der Ort, an dem die Anfrage gestellt wird, befindet sich bei CNN Anfragen typischerweise außerhalb des Suchgebietes. Dies ist der wesentliche Unterschied zu den gewöhnlichen NN Anfragen, denn dort wird der Suchraum nicht geographisch beschränkt werden und der Ausgangspunkt für die Anfrage befindet sich immer im Suchgebiet.

In Abbildung 7 sehen wir eine typische Anfrage, wie die oben erwähnte Anfrage nach der nächstgelegenen Tankstelle im Nordosten der aktuellen Position.

Zurückgeliefert werden dürfen nur Objekte deren x - und y -Werte größer sind als die der aktuellen Position q . Die Ergebnismenge ist definiert als:

$$\{r \mid (r_x > q_x \wedge r_y > q_y) \wedge (\forall o : o_x > q_x \wedge o_y > q_y \rightarrow d(o, q) \geq d(r, q))\}$$

wobei $r_{x,y}$ und $q_{x,y}$ die jeweiligen Koordinatenwerte von r und q sind und d die Abstandsfunktion.

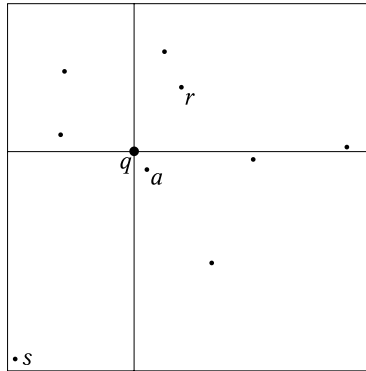


Abbildung 7: CNNQ Beispiel mit einem Suchgebiet

Wir sehen, dass bei einer normalen NN Anfrage das Objekt a zurückgeliefert wird und bei der CNNQ das Objekt r . Wenn das Szenario aus Abbildung 7 beibehalten wird, die Anfrage jedoch so geändert wird, dass jetzt nach der nächstgelegenen Tankstelle im Südwesten gefragt wird, so sehen wir ein Extrembeispiel. Das Objekt s wird in diesem Fall zurückgeliefert, obwohl es das am weitesten entfernte Objekt von der aktuellen Position q ist.

Bei diesen beiden Fällen war der Anfrageort räumlich mit dem Suchgebiet verbunden bzw. es gab nur ein Suchgebiet. In manchen Fällen ist es jedoch sinnvoll mehrere Suchgebiete anzugeben in denen gesucht werden soll. So zum Beispiel wenn man folgende Anfrage stellen will:

- Befindet sich das nächste italienische Restaurant in dem Ort westlich oder östlich von der Autobahn (siehe Abbildung 8)?

Eine reguläre NN Anfrage würde Objekt a zurückliefern. Die CNN Anfrage liefert jedoch Objekt r zurück.

2.3.2. 2-Phasen Algorithmen für CNN Anfragen nach [6]

Inkrementelles Suchen

In der ersten Phase liefert ein inkrementeller NN-Algorithmus (in [7] wird ein solcher inkrementeller Algorithmus vorgestellt der später in [8] an R-Bäume angepasst wurde), der nicht auf ein bestimmtes Suchgebiet R beschränkt ist, die nächsten Nachbarn zu einer gegebenen Position p aufsteigend nach der Entfernung zu p sortiert. In der zweiten

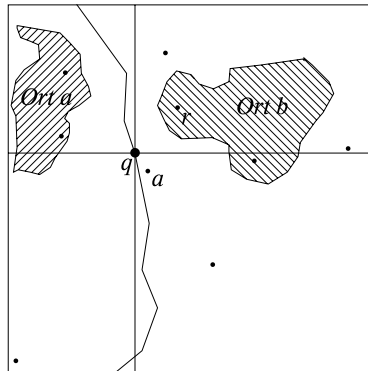


Abbildung 8: CNNQ mehrere Suchgebiete

Phase prüft der Algorithmus in jeder Iteration ob die gefundenen Objekte mit R überlappen. Der Algorithmus stoppt sobald sich das zurückgelieferte Objekt in R befindet. Man kann sich vorstellen, dass dabei einige nächste Nachbarn gefunden werden, die wieder verworfen werden müssen, weil sie sich nicht innerhalb des Suchgebiets befinden.

Der Sonderfall, dass R keine nächsten Nachbarn hat, muss noch abgefangen werden. Das geschieht dadurch, dass der Algorithmus stoppt sobald ein gefundenes Objekt weiter entfernt ist als das Suchgebiet R . Hierzu wird die Funktion $maxdist(p, r)$ benötigt.

NN Suche mit Ranged Anfrage

In einigen Fällen kann das inkrementelle Suchen jedoch zu unnötigen Zugriffen auf Blattknoten führen. Ein anderer Ansatz ist es die zwei Phasen von vorher umzudrehen.

Abhängig von der Größe und der Position des Suchgebietes ist eine Ranged Query effizienter um die Objekte aus dem Suchgebiet zu erhalten und auf die Nächste-Nachbar-Bedingung zu überprüfen.

Diese Methode ist ein guter Kompromiss, wenn das Suchgebiet nicht groß ist bzw. nicht viele Objekte enthält. Ansonsten ist das Überprüfen aller Punkte auf die Nächste-Nachbar-Bedingung sehr ineffizient.

Wenn das Suchgebiet jedoch nur wenige Objekte enthält die überprüft werden müssen und dazu noch weit entfernt von dem Anfragepunkt ist, dann ist dieser Ansatz effizienter als der Erste. Wie in Abbildung 9 zu sehen ist, würde der erste Algorithmus wesentlich mehr unnötige Objekte verwerfen müssen als Algorithmus zwei.

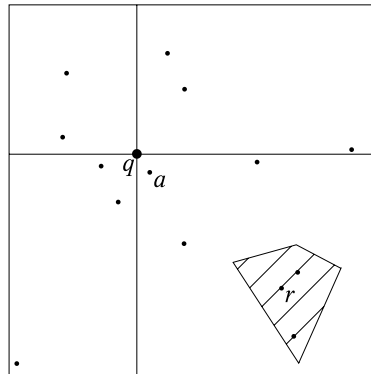


Abbildung 9: CNN Ranged Anfrage

2.3.3. Optimierter 1-Phasen Ansatz nach [6]

Dieser Ansatz ist wesentlich effizienter als die im vorherigen Abschnitt beschriebenen Ansätze. Die zwei oben beschriebenen Phasen werden dabei zusammengeführt. Welcher Algorithmus für die notwendigen NN Anfragen verwendet wird, ist nur sekundär wichtig. Interessant sind die Ansätze von [3, 7, 8]. Was auf jeden Fall benötigt wird ist ein NN Algorithmus der so modifiziert ist, dass er mit beschränkten Suchgebieten arbeiten kann. Dass neue Bedingungen für die Auswahl von in Betracht kommenden MRHs bei der nächste Nachbar Suche notwendig sind, macht Abbildung 10 deutlich.

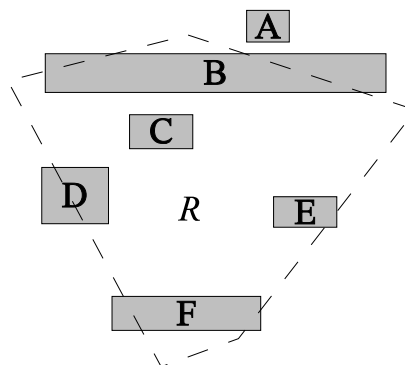


Abbildung 10: für einzuschließende MRH (Abb. nach [4])

- MRH A liegt außerhalb des Suchgebietes R (nicht zu gebrauchen)
- MRH C liegt vollständig im Suchgebiet R

- eine MRH, die keine vollständige Kante im Suchgebiet R hat, ist nicht zu gebrauchen (F, B)
- eine Kante oder mehrere der MRHs D und E liegen in R . $minmaxdist(q, D)$ kann nur in Bezug auf die Kanten, die sich in R befinden, angewendet werden

Folgende Modifikationen werden in [6] vorgeschlagen:

Modifizierung der $mindist(q, M)$ Funktion

Wir definieren die überlappende Fläche von M mit R als I_R . Weil I_R höchstens so groß ist wie M muss also $mindist(q, I_R) \geq mindist(q, M)$ gelten. Wenn $mindist(q, I_R)$ größer ist als $minmaxdist(q, M', R)$ für alle MRHs M' kann die MRH M ausgeschlossen werden.

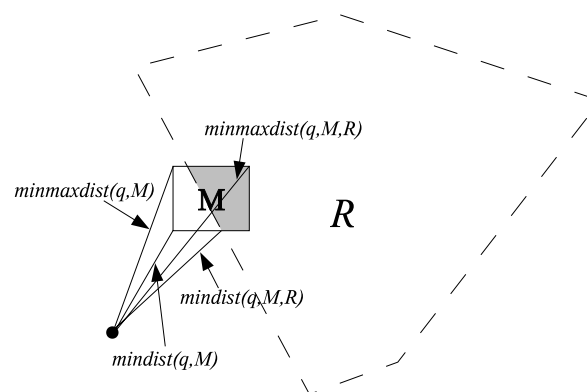


Abbildung 11: Modifizierte $mindist(q, M)$ und $minmaxdist(q, M)$

$mindist(q, I_R)$ ist das Minimum aller Distanzen von der Anfrageposition zu den Kanten von des überlappenden Gebietes I_R .

Neue Überlegung zur Enthaltung bei CNNQ

$minmaxdist(q, M)$ kann zu falschen Ergebnissen führen denn diese Funktion garantiert nicht, dass auch wirklich ein Objekt aus M in R enthalten ist, weil M vielleicht nicht ganz in R enthalten ist. Bei der Berechnung von $minmaxdist(q, M)$ sollten also nur Kanten in Betracht gezogen werden, die auch vollständig in R enthalten sind. Wenn keine Kante von M in R enthalten ist, dann ist $minmaxdist(q, M) = \infty$.

Bedingungen dafür, dass eine Kante von M in R enthalten ist:

1. konstruiere unendliche Linien, die parallel zur x-Achse sind, durch die obere und untere Kante von M .
2. berechnen der x-Koordinaten der Schnittpunkte mit dem Polygon M .
3. eine Kante der MRH M ist vollständig in R enthalten, wenn ihre beiden Endpunkte in R enthalten sind.
4. $\minmaxdist(q, M, R)$ wird aus den vollständig in R enthaltenen Kanten berechnet

Die zwei konstruierten Linien y und y' schneiden das beschränkte Zielgebiet jeweils zwei mal oder kein mal. Wenn es keine Schnittpunkte mit den Kanten des Zielgebietes gibt dann liegt M außerhalb von R .

Wenn es aber Schnittpunkte gibt, dann müssen noch die vier Schnittpunkte von y und y' mit dem Zielgebiet berechnet werden.

M wird im Folgenden als $[(x_{tief}, y_{tief}), (x_{hoch}, y_{hoch})]$ beschrieben. Wir erhalten nun folgende Schnittpunkte:

1. x_1 = Schnittpunkt von y mit dem Zielgebiet auf der linken Seite
2. x_2 = Schnittpunkt von y mit dem Zielgebiet auf der rechten Seite
3. x'_1 = Schnittpunkt von y' mit dem Zielgebiet auf der linken Seite
4. x'_2 = Schnittpunkt von y' mit dem Zielgebiet auf der rechten Seite

Nachdem die Schnittpunkte berechnet wurden muss nun noch überprüft werden ob alle Ecken von M innerhalb des Zielgebietes sind. Dies geschieht durch folgende Gleichung:

$[x_{tief}, y_{tief}] \in R$ wenn $x_1 \leq x_{tief} \leq x'_1$ (analog für die alle weiteren Ecken von M).

2.4. Föderierte Nearest Neighbor Queries

Als Erstes soll die Frage geklärt werden, was föderierte Nearest Neighbor Queries von den oben vorgestellten Nearest Neighbor Query Arten unterscheidet. Der entscheidende Unterschied zwischen den oben vorgestellten Query Arten und dem föderierten Ansatz ist die Tatsache, dass die Objekte nicht auf einem einzigen Server gehalten werden, auf dessen Datenbank direkt zugegriffen werden kann, sondern auf mehreren Servern verteilt gespeichert sind.

Um den Zugriff auf die gespeicherten Objekte in diesen Servern für die Clients zu vereinfachen, wird eine Komponente benötigt, die die Datenhaltung transparent macht. Diese Komponente heißt Föderation und stellt Schnittstellen zur Verfügung mit denen einheitlich auf die Objekte der Server zugegriffen werden kann, ohne dass Kenntnisse (z.B. über den Ort des Servers, die Art der Daten, usw.) über die Server benötigt werden. Die Föderation selbst verfügt über einen Verzeichnisdienst (VD), der bei der Ermittlung des für eine Anfrage zuständigen Servers genutzt werden kann. So kann die Föderation an den Verzeichnisdienst eine Gebietsanfrage stellen und dieser liefert der Föderation zurück, welche Server für das angefragte Gebiet zuständig sind. Die Föderation leitet dann die Anfrage an diese Server weiter und fasst die Anfrageergebnisse von den verschiedenen Servern zusammen und stellt diese dann durch die Schnittstellen einheitlich zur Verfügung.

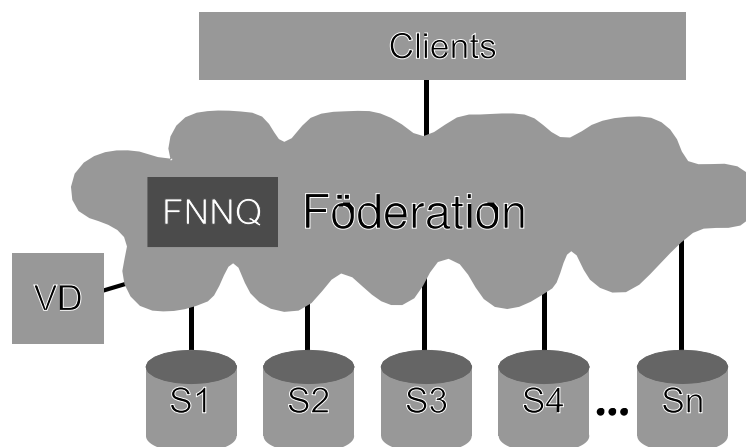


Abbildung 12: verteilte Datenhaltung mit Föderation

Dieses Konzept löst zwar viele Probleme beim Zugriff auf die Daten und funktioniert gut bei Gebietsanfragen, es schafft aber auch ein Neues, welches uns bei den nächsten

Nachbar Anfragen behindert. Alle bisher vorgestellten Algorithmen für nächste Nachbar Anfragen benötigen direkten Zugriff auf die Indexstrukturen der Daten. Der Zugriff auf die Indexstrukturen ist aber durch den föderierten Ansatz nicht möglich. Es kann lediglich über eine (restriktive) Anfrageschnittstelle auf die Daten der Server zugegriffen werden. So stellt diese Anfrageschnittstelle zwei Zugriffsarten auf die Server zur Verfügung. Zum einen die Gebietsanfragen bei der alle Ergebnisse aus einem spezifizierten Gebiet zurückgeliefert werden und zum anderen die k nächste Nachbarn Anfrage, die die k nächsten Nachbarn zu einem Anfrageort von einem Server zurückliefert. Der Vorteil der nächsten Nachbar Anfrage an einen Server hat den Vorteil, dass man diesen Server für eine FNNQ nur einmal abfragen muss, was zeitaufwendige Netzanfragen an die Server spart. Man kann sicher sein, dass der Server keine für die FNNQ interessanten Objekte mehr enthält. Es muss allerdings beachtet werden, dass es dadurch oft zu unnötigen Ergebnissen kommt, die später wieder verworfen werden und zusätzlichen Netzwerktraffic bedeuten. Gebietsanfragen an die Server verringern die Anzahl der unnötigen Ergebnisse pro Server, jedoch muss mit vielen Serveranfragen gerechnet werden, wenn das Gebiet öfter zu klein gewählt wird. Kurz gesagt reduzieren die Gebietsanfragen die Anzahl der unnötigen Ergebnisse und nächste Nachbar Anfragen die Anzahl der Anfragen an die Server. An dieser Stelle bleibt offen, ob die Reduzierung der unnötigen Ergebnisse bei den Gebietsanfragen ein besseres Tradeoff bieten als die Reduzierung der Netzanfragen an die Server bei den nächste Nachbarn Anfragen.

Es gilt nun einen Algorithmus zu entwerfen der auch ohne lokalen Zugriff auf die Daten und deren Indices effizient arbeitet.

3. Der Algorithmus

3.1. Nexus

Bei der Nexus Plattform sind die Objekte, wie oben beim allgemeinen föderierten Prinzip beschrieben, verteilt auf mehreren Servern gespeichert. Durch eine einheitliche Serviceschnittstelle ist es für jeden Dienstleister möglich einen eigenen Server an die Nexus Plattform anzuschließen. Diese Server heißen bei Nexus Spatial Model Server (SpaSe) und speichern jeweils Objekte aus einem geographischen Gebiet. Die Daten eines SpaSe werden sogenannten Augmented Areas (AA) zugeordnet, welche jeweils einen Teil des geographischen Gebietes des SpaSe abdecken. Die AAs bieten eine bestimmte logische Sicht auf die Daten eines SpaSe. Sie können auf bestimmte Objekttypen (z.B. Straßen) beschränkt werden bzw. können unterschiedliche Detaillevel haben (z.B. eine ganze Stadt oder nur ein Gebäude). So kann man sich beispielsweise vorstellen, dass ein SpaSe eine AA für Gebäude, eine für Straßen usw. hat.

Der Verzeichnisdienst, den die Nexus Föderation nutzt, heißt Area Service Register (ASR). Das ASR wird benötigt, um zu ermitteln welche SpaSes für das zu betrachtende geographische Gebiet zuständig sind. Das ASR ist ein Repository, das Informationen über die Größe der Augmented Areas, die geographischen Gebiete welche von ihnen abgedeckt werden, die Objekttypen die sich in ihnen befinden und welcher SpaSe Augmented Areas mit diesen Objekttypen hält. Wie wir sehen werden ist das ASR eine zentrale Komponente für den hier vorgestellten Algorithmus.

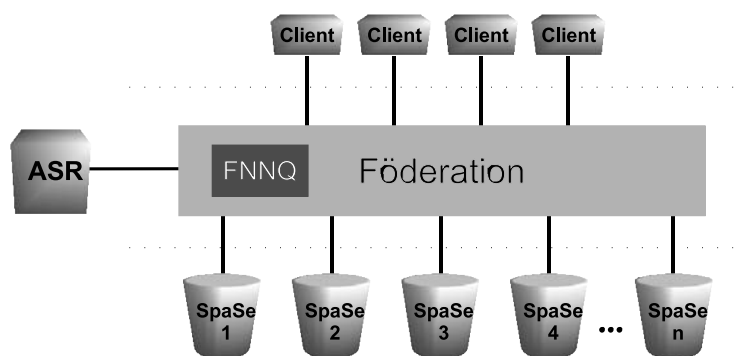


Abbildung 13: Nexus Grobübersicht

Eine weitere wichtige Komponente mit der wir uns bei den föderierten Nearest Neighbor Queries beschäftigen müssen sind die SpaSe. Die SpaSes können grundsätzlich Ge-

bietsanfragen und k nächste Nachbar Anfragen beantworten. Letztere sind sehr nützlich für den hier vorgestellten Algorithmus. In Kapitel 3.2 und ff. werden wir sehen wie uns nächste Nachbar Anfragen an die SpaSe helfen werden die Anzahl der unnötigen Treffer bei der Suche nach den föderierten nächsten Nachbarn zu reduzieren.

Es gilt nun in den nachfolgenden Abschnitten folgende Szenarien zu untersuchen:

Szenarien

1. Alle Ergebnisse befinden sich in den AAS, in denen sich der Benutzer befindet
2. Alle Ergebnisse befinden sich in AAS, in denen sich der Benutzer nicht befindet
3. Ein Teil der Ergebnisse befindet sich in den AAs, in denen sich der Benutzer befindet

Probleme

1. Woher weiß ich, dass ich fertig bin?
2. Wie vermeide ich unnötige Treffer?
3. Wie sieht die Iteration aus?

3.2. Der Algorithmus

Um die k -Nächsten Objekte zu finden werden zunächst, in einer initialen Phase, alle SpaSe abgefragt, deren AAs den gewünschten Objekttyp haben und mit der Position des Benutzers überlappen. Die abzufragenden SpaSe werden vom ASR eingeholt und deren AAs in der Menge $S = \{s_1 \dots s_t\}$ gespeichert. Aus den Ergebnissen der Anfragen an die SpaSe wird eine Ergebnismenge $T = \{x_1 \dots x_n\}$ gebildet. T wird der Entfernung zum Benutzer nach, aufsteigend sortiert gespeichert (vgl. Abbildung 14).

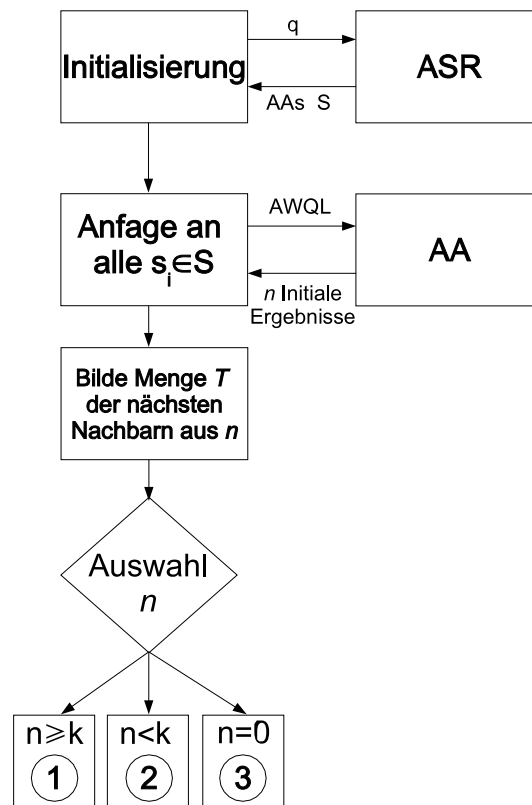


Abbildung 14: Initialer Schritt

Die besuchten AAs werden markiert indem sie in der Menge M gespeichert werden. Aus den ersten k Elementen von T wird später eine Finalmenge F mit den tatsächlich k -nächsten Nachbarn gebildet. Es können nun folgende Fälle eintreten.

3.2.1. Es werden $n \geq k$ Ergebnisse gefunden

Das bedeutet, dass in der initialen Phase mindestens k Nachbarn gefunden worden sind. Es muss nun noch überprüft werden ob unter den gefundenen n Nachbarn tatsächlich die k -nächsten Nachbarn zu der Position des Benutzers sind und der Algorithmus stoppen kann oder ob es nähere Nachbarn, in bisher noch nicht befragten, AAs gibt. Die Menge T wird zunächst auf die ersten k Elemente gekürzt. Es entsteht $T = \{x_1 \dots x_k\}$. Im weiteren werden folgende Schritte durchgeführt (vgl. Abbildung 15):

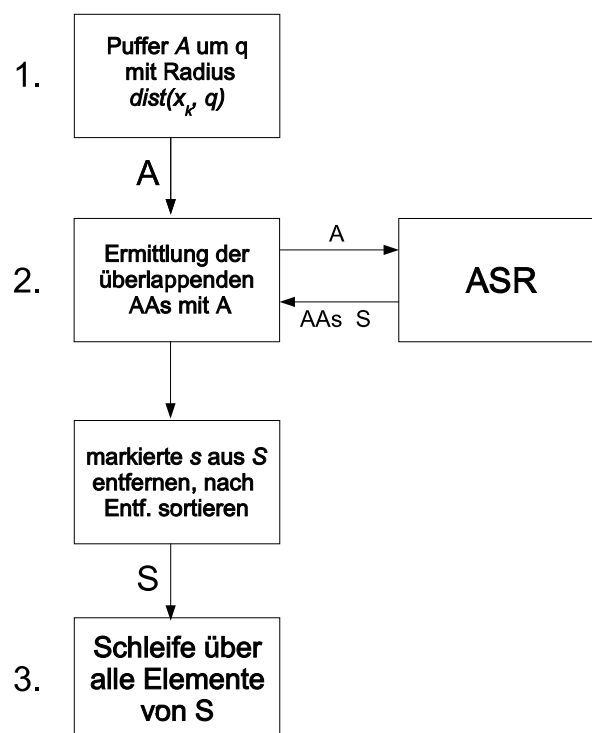


Abbildung 15: $n \geq k$

1. Um den Benutzer q wird ein Kreis (Puffer) durch den am weitesten entfernten nächsten Nachbarn aus T gelegt. Weiter entfernte AAs interessieren nicht, da dort keine näheren Nachbarn gefunden werden können. Der Radius beträgt $dist(x_k, q)$. Der Puffer hat die Fläche A .
2. Nun wird beim ASR angefragt welche AAs mit A überlappen. Zusätzlich wird überprüft, welche AAs schon markiert worden sind, denn diese sollen nicht weiter

betrachtet werden. Die übrigen AAs werden in der Menge $S = \{s_1 \dots s_t\}$ nach der Entfernung (Entfernung zum Rand der Augmented Area) zum Anfrageort aufsteigend sortiert.

3. In einer Schleife (vgl. Abbildung 16) wird die Suche in den Service Areas s_i , mit $1 \leq i \leq t$, der Menge S fortgesetzt. Es wird immer die Service Area aus S abgefragt, die am nächsten zum Anfrageort ist und noch nicht markiert ist. In der ersten Schleifeniteration ist das s_1 . Diejenigen Objekte in der Ergebnismenge T , die näher am Anfrageort sind als die zur Abfrage übrig gebliebenen Service Areas aus S können als endgültige Ergebnisse gewertet werden und werden in die Menge F aufgenommen. Es müssen nur noch die $r = (k - |F|)$ nächste Objekte von den noch abzufragenden Service Areas angefordert werden. Wenn $r = 0$, dann kann die Schleife an dieser Stelle beendet werden. Ansonsten werden von s_i die r nächsten Nachbarn angefordert. Liefert die Anfrage an s_i keine Ergebnisse zurück so wird s_i markiert und die Schleife mit dem nächsten Element aus S durchlaufen. Liefert die Anfrage an s_i jedoch Ergebnisse zurück so werden diese in eine Menge $U = \{u_1 \dots u_j\}$ eingetragen. Es muss noch überprüft werden ob sich die Menge der k -nächsten Nachbarn durch die neu gefundenen Objekte ändert. Die neue Menge der k -nächsten Nachbarn ergibt sich wie folgt: $T_{neu} = (T_{alt} \cup U \mid \text{Menge auf st. nach Entfernung zu } q \text{ sortiert und die } j \text{ letzten Elemente entfernt})$. Die Menge S der zu durchsuchenden Augmented Areas wird an das neue $x_k \in T_{neu}$ angepasst. Alle Augmented Areas die weiter vom Kunden entfernt sind als $dist(x_k, q)$ können jetzt aus S entfernt werden. Die besuchte Augmented Area wird markiert. Wenn in der Schleife alle Elemente aus S abgearbeitet worden sind kann der Algorithmus hier stoppen, ansonsten wird die Schleife nochmals durchlaufen.

Die Menge T enthält nun die tatsächlichen k -nächsten Nachbarn. Bei diesem Ansatz, immer die Augmented Area abzufragen welche dem Benutzer am nächsten ist, wird vermieden, dass unnötig Anfragen an weit entfernte Augmented Areas gestellt werden, da durch den iterativen Ansatz der zu betrachtende Puffer nach jeder Iteration kleiner wird und somit Anfragen an weit entfernte Service Areas entfallen.

Der Nachteil ist, dass alle AAs nacheinander abgefragt werden, was recht lange dauern kann.

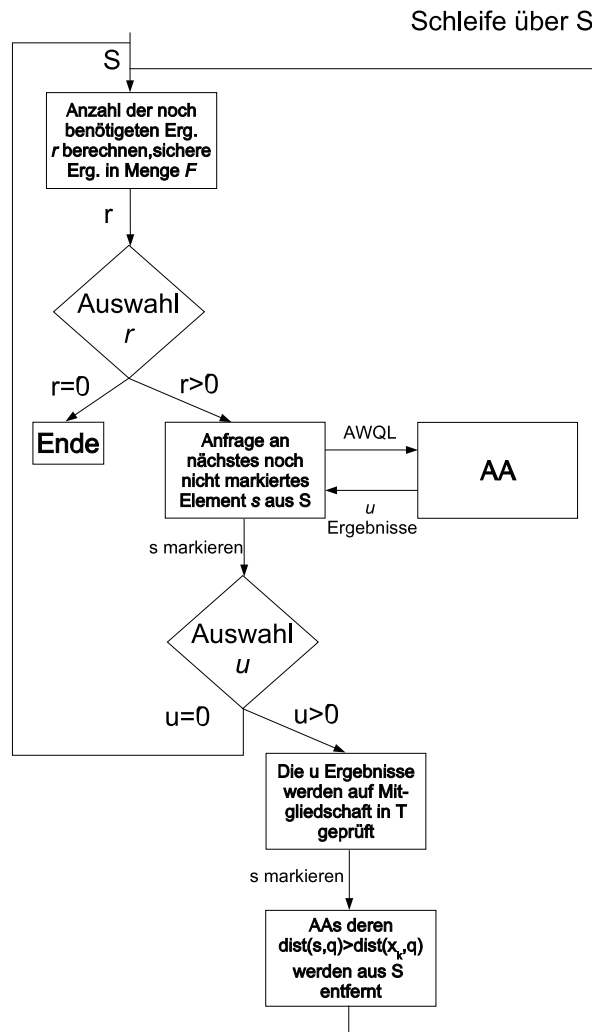


Abbildung 16: $n \geq k$ Schleife

3.2.2. Es werden $0 < n < k$ Ergebnisse gefunden

Auch hier wird eine Ergebnismenge $T = \{x_1 \dots x_n\}$ gebildet. Da mindestens ein Ergebnis aus den AAs, in denen sich der Benutzer befindet, zurückgeliefert wird, kann zunächst so vorgegangen werden, wie in 3.2.1 beschrieben, wobei das x_k in 3.2.1 durch das x_n aus diesem T ersetzt wird. Wenn sich nach dem Abarbeiten von 3.2.1 mindestens k Elemente in T befinden, haben wir genau die oben erwähnte Situation mit $n \geq k$ und es kann nach einem erneuten Durchlauf von 3.2.1 gestoppt werden.

Wenn nun aber nicht genügend weitere Ergebnisse gefunden wurden, wird der Puffer, in dem nach passenden AAs gesucht wird, iterativ vergrößert. Hier besteht dann besteht

die Hauptproblematik darin, zu entscheiden, wie groß der Puffer, mit dem man weiter arbeiten will, gewählt werden muss.

Es werden nun folgende Schritte durchgeführt:

1. Der Puffer, der um die Position des Benutzer gelegt wird, hat den Radius $r_1 = \frac{k}{n_1} \cdot dist(x_{n_1}, q)$, wobei n_1 die Anzahl gefundenen Ergebnisse nach dem initialen Schritt ist und x_{n_1} das von q am weitesten entfernte Ergebnis aus T nach dem initialen Schritt ist.
2. In einer Schleife wird nun mit diesem Puffer wie in 3.2.1 vorgegangen. Falls wir dannach k Ergebnisse gefunden haben, kann nach einem erneuten Durchlauf von 3.2.1 hier gestoppt werden ansonsten wird der Radius des Puffers entweder proportional zum Anteil der noch fehlenden Treffer, oder auf den jetzt am weitesten entfernten Treffer vergrößert und dieser Schritt nochmals durchgeführt.

Formal: $r_{i+1} = \max(\frac{k}{n_i} \cdot r_i, \frac{k}{n_i} \cdot dist(x_{n_i}, q))$

Wir brauchen $\frac{k}{n_i} \cdot r_i$ für den Fall, dass wir nach einer Iteration keine neuen Ergebnisse gefunden haben. Bei $\frac{k}{n_i} \cdot dist(x_{n_i}, q)$ ist der Radius des Puffers nicht von der Größe des alten Radius abhängig sondern nur von der Anzahl der gefundenen Ergebnisse. Wenn aber nach einer Iteration keine neuen Ergebnisse gefunden werden und wir für die Puffervergrößerung nur $\frac{k}{n_i} \cdot dist(x_{n_i}, q)$ verwenden, wird der Algorithmus in einer Endlosschleife hängen bleiben. Durch $\frac{k}{n_i} \cdot r_i$ wird gewährleistet, dass der Radius des Puffers vergrößert wird, auch wenn keine neuen Ergebnisse gefunden wurden.

Dass $dist(x_{n_i}, q)$ größer sein kann als r_i , falls neue Ergebnisse gefunden werden, zeigt Abbildung 17.

3.2.3. Es werden keine Ergebnisse gefunden

Auch hier besteht die Hauptproblematik darin, zu entscheiden, wie groß der Puffer, mit dem man arbeiten kann, gewählt werden muss. Auch hier wird wie in 3.2.2 der Puffer iterativ vergrößert. Im Gegensatz zum Abschnitt 3.2.2 haben wir aber kein Referenzobjekt mit dem wir eine initiale Puffergröße bestimmen können. In diesem naiven Ansatz, der hier zunächst vorgestellt wird, nehmen wir als initialen Pufferradius 500m an. Dann kann wie in Schritt 2 aus Abschnitt 3.2.2 fortgefahren werden.

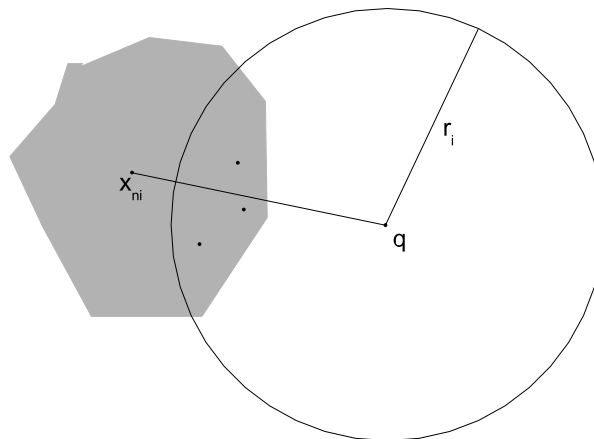


Abbildung 17: $\text{dist}(x_{n_i}, q)$ ist größer als r_i

3.2.4. Terminierung in den Fällen $0 < n < k$ und $n = 0$

Für die Fälle aus Kapitel 3.2.2 und 3.2.3, wo es zu wenige Treffer bzw. keine Treffer gibt kann nicht garantiert werden, dass überhaupt k nächste Nachbarn vom gewünschten Objekttyp gefunden werden.

Grundsätzlich kann hier eine der zwei Erweiterungen für das ASR Abhilfe schaffen:

1. Das ASR kennt die gesamte Fläche, die von allen in Frage kommenden AAs abgedeckt wird. Wenn der Puffer während der Iterationen größer wird als diese Fläche kann nach einer weiteren Iteration gestoppt werden.
2. Das ASR kennt die Anzahl der für die Anfrage in Frage kommenden AAs. Wenn die Menge der markierten AAs dieser Anzahl entspricht kann terminiert werden.

Beide Erweiterungen haben jedoch den Nachteil, dass sie an die Anfrage gebunden sind und jedes mal neu berechnet werden müssen. Da diese Erweiterungen sehr viel Rechenleistung kosten, kann überlegt werden ob man Abstriche bei der Genauigkeit in Kauf nimmt und zu folgenden Alternativen greift:

1. Das ASR berechnet die Gesamtfläche aller AAs unabhängig von einer Anfrage. Damit ist die Gesamtfläche in einigen Fällen größer als notwendig, muss aber nicht bei jeder Anfrage neu berechnet werden.
2. Das ASR berechnet die zur Abfrage verbleibenden AAs erst nach einer gewissen Anzahl an Iterationen. Es wird davon ausgegangen, dass in den meisten Fällen

genügend Ergebnisse gefunden wurden bevor dieser Vorgang angestoßen wird und somit das Zählen der AAs nur selten durchgeführt werden muss. Erstrebenswert wäre es die Anzahl der durchzuführenden Iterationen nicht statisch festzulegen sondern Erfahrungswerte zu sammeln und die Anzahl dynamisch an die Erfahrungswerte anzupassen.

3.3. Optimierungen

3.3.1. Bestimmung der Puffergröße für die erste Iteration

Wir haben in Abschnitt 3.2.2 und 3.2.3 gesehen, dass es schwierig ist eine initiale Puffergröße zu bestimmen, wenn keine zusätzliche Information über ein Gebiet vorhanden ist, wie z.B. die Anzahl der darin enthaltenen Objekte von dem gewünschten Objekttyp.

Hier soll nun ein Ansatz vorgestellt werden (nach [1]), der ausgehend von der Objektdichte in einer Augmented Area einen initialen Puffer für die Suche bestimmt. Bei diesem Ansatz wird von einer Gleichverteilung der Objekte innerhalb der Augmented Areas ausgegangen.

Um die Größe des Puffers für die Anfragen zu bestimmen wird zunächst, die gemittelte Dichte der Augmented Areas berechnet. Die Berechnung des Radius anhand der Objektdichte hat den Vorteil, dass nur wenig statistisches Wissen über die Augmented Area notwendig ist. Benötigt wird lediglich die Anzahl der Objekte vom angeforderten Typ und die Fläche der Augmented Area. Die Anzahl der gespeicherten Objekte von einem Typ wird zum momentanen Zeitpunkt noch nicht vom ASR bereitgestellt, die Größe der Fläche der AAs wird vom ASR zur Verfügung gestellt.

Die Dichte einer Augmented Area $D(AA)$ wird berechnet aus der Anzahl von Objekten w des gewünschten Typs die sich in der Augmented Area befinden und der Fläche der Augmented Area .

$$D(AA) = \frac{w}{Area(AA)}$$

Für die Bestimmung des Puffers müssen wir folgende Fallunterscheidung durchführen:

1. *Es gibt mindestens einen Treffer:* in diesem Fall benötigen wir die mittlere Dichte aller Augmented Areas aus der Menge S der Augmented Areas, in der sich der Kunde befindet, denn wir können von der Objektdichte einer einzelnen Augmented Area nicht auf die Dichte einer anderen schließen. Die gemittelte Dichte aus mehreren AAs ist repräsentativer.
2. *Es gibt wie in Kapitel 3.2.3 beschrieben keine Treffer:* in diesem Fall hätten wir die Dichte $D(AA) = 0$. Es kann nun so vorgegangen werden, dass das Mittel aus den nächstgelegenen 10 AAs oder das Mittel aus allen AAs in 10km Umkreis oder sogar das Mittel aus allen verfügbaren AAs berechnet wird.

Die mittlere Dichte D_m wird wie folgt berechnet:

$$D_m = \frac{D(AA_1) + \dots + D(AA_p)}{p}$$

D_m wird nun dazu verwendet den Radius r für den Puffer zu berechnen.

$$r = \sqrt{\frac{k}{\pi D_m}}$$

Da r unter der Annahme der Gleichverteilung von Objekten errechnet wurde, kann r nur als Schätzwert betrachtet werden und muss keinesfalls schon nach einer Iteration des Algorithmus zu den gewünschten k -nächsten Nachbarn führen. Wenn nun weniger als k nächste Nachbarn gefunden werden führen wir folgende Fallunterscheidung durch:

1. Wenn noch keine Ergebnisse gefunden wurden wird der Radius des Puffers um den Faktor 2 vergrößert und der Algorithmus noch einmal ausgeführt
2. Wenn noch nicht genügend Ergebnisse gefunden wurden, wird die Dichte der Objekte innerhalb des Puffers $D(Puffer)$ berechnet und diese wiederum als Grundlage für eine neue Berechnung von r verwendet und der Algorithmus noch einmal mit dem neu berechneten r ausgeführt. Dieser Schritt kann auch als Berechnungsgrundlage für Schritt zwei in Kapitel 3.2.2 genommen werden.

Damit diese Optimierung in der Nexus Föderation genutzt werden kann muss das ASR lediglich die Anzahl der Objekte von jedem Objekttyp aus jeder AA speichern. Als zusätzliche Erweiterung kann angedacht werden, die Berechnung der Dichte für alle Objekttypen pro AA vom ASR durchführen zu lassen.

3.3.2. Mehrere Server gleichzeitig abfragen

Der Flaschenhals bei dem oben beschriebenen Verfahren für föderierte Nearest Neighbor Queries sind die Latenzzeiten bei den Anfragen an die SpaSe und die sequentielle Abarbeitung der anzufragenden SpaSe.

Eine Optimierung ist es Anfragen an mehrere Server gleichzeitig zu stellen. Gruppen von $ld(|S|) \cdot 2$, wobei $|S|$ die Anzahl der Server die mit dem Puffer überlappen ist, Servern gleichzeitig abzufragen bietet hier ein gutes Tradeoff. Der Logarithmus Dualis hat den Vorteil, dass selbst bei großem k die Anzahl der gleichzeitig abzufragenden Server verhältnismäßig klein bleibt. So wird eine parallele Verarbeitung mit relativ geringem Rechenaufwand gewährleistet.

Der Sonderfall $|S| = 1$, wenn es nur einen Server gibt, der mit dem Puffer überlappt, muss abgefangen werden, da hier $ld(1) = 0$ ergibt.

3.4. Korrektheit

Der erste Teil des Korrektheitsbeweises, die Terminierung, wurde schon oben in Kapitel 3.2.4 und Kapitel 3.2.1 besprochen. In Kapitel 3.2.4 wurde gezeigt, dass im Falle $n < k$ und $n = 0$ der Algorithmus spätestens dann aufhört, wenn entweder alle in Frage kommenden AAs durchsucht worden sind oder die Fläche des zu durchsuchenden Gebietes größer ist als die gesamte Fläche in der Ergebnisse zu erwarten sind. Wenn $n \geq k$ werden nach dem initialen Schritt noch höchstens so viele AAs durchsucht wie mit dem Puffer aus Kapitel 3.2.1 Schritt 1 überlappen. Dann terminiert der Algorithmus auch hier. Da die besuchten AAs markiert werden besteht auch nicht die Gefahr einer Endlosschleife durch immer wiederkehrende Anfragen an besuchte AAs.

Im zweiten Teil gilt es zu zeigen, dass auch tatsächlich die nächsten Nachbarn gefunden werden. Zunächst versucht der Algorithmus den Puffer immer weiter zu vergrößern bis er k nächste Nachbarn findet. Falls der Algorithmus nicht genügend Ergebnisse findet terminiert er irgendwann, wie oben beschrieben, und die gefundenen Objekte sind automatisch die nächsten Nachbarn, da keine weiteren Nachbarn im gesamten Gebiet gefunden wurden. Falls sich nach einer Iteration k Ergebnisse in dem Puffer befinden beginnt der Algorithmus zu konvergieren. Es werden nur noch AAs abgefragt, die in der jeweiligen Iteration mit dem Puffer überlappen. Alle anderen AAs bleiben unberücksichtigt, da schon k Ergebnisse gefunden worden sind und alle AAs die nicht mit dem Puffer überlappen keine näheren Ergebnisse mehr liefern können. Da der Algorithmus im Fall $n \geq k$ erst stoppt wenn alle AAs, die mit dem Puffer überlappen, abgefragt werden und die anderen AAs nicht relevant sind ist sichergestellt, dass die k nächsten Nachbarn gefunden werden.

4. Die Implementierung

4.1. Benötigte Komponenten

Folgende Komponenten werden für die hier vorgestellte Implementierung des Algorithmus benötigt:

- Nexus Geoklassen. Diese werden für die geometrischen Berechnungen im Algorithmus benötigt (z.B. Abstandsberechnungen zwischen Objekten, Flächenberechnungen, usw.)
- JTS Topology Suite (JTS) in der Version 1.3 [14]. Die JTS Bibliotheken werden von den Nexus Geoklassen für geometrische Berechnungen genutzt.
- Generalised Search Tree (GiST) [13]. Die R-Baum Implementierung der GIST Bibliotheken wird für die ASR Implementierung benötigt. Sie beschleunigt die Suche nach den für eine Anfrage relevanten SpaSes.
- Java Runtime Environment 1.4.x [15].

4.2. Durchgeführte Anpassungen an der Nexus Föderation

Folgende Anpassungen wurden im Rahmen dieser Studienarbeit durchgeführt:

- Portierung und Anpassung der Startupskripte der Nexus Föderation von DOS-Batch nach Unix-Bash.
- Modifizierung des DummyASR und des DummySpaSe mit dem Ziel der Plattformunabhängigkeit. Die Konfigurationsdateien befinden sich nun nicht mehr statisch auf dem Laufwerk C:\ sondern im Homeverzeichnis des jeweils am Rechner angemeldeten Benutzers. Durch die Änderung sind nun mehrere Konfigurationen pro Rechner und eine Benutzung der Klassen unter Linux und Unix Betriebssystemen möglich.

4.3. Implementierung des Algorithmus

Im Rahmen dieser Studienarbeit wurden folgende Klassen implementiert

- *GenerateTestdata*: ein Testdatengenerator

- *ASR, SpaSe*: eine ASR und eine SpaSe Implementierung, die nur die von dem hier vorgestellten Algorithmus geforderte Funktionalität haben.
- *FNNQ*: der eigentliche Algorithmus
- *StartQuery*: die Hauptklasse für den Benutzer. Sie steuert die Testläufe. Hier kann der Benutzer einstellen, wieviele Objekte erzeugt werden sollen, wieviele AAs erzeugt werden sollen und wieviele nächste Nachbarn gesucht werden sollen.

Der Einfachheit halber gibt es nur einen Objekttyp und nur jeweils eine AA pro SpaSe.

Die folgenden UML Klassendiagramme zeigen jeweils im oberen Teil die Klassenattribute und im unteren Teil die Methoden und Konstruktoren der Klassen. Ein Minuszeichen (*private*) vor dem jeweiligen Attribut oder der jeweiligen Methode bedeutet, dass von außen nicht darauf zugegriffen werden kann und diese nur für die interne Verarbeitung benötigt werden. Ein Pluszeichen (*public*) bedeutet, dass der Zugriff gestattet wird. Hinter dem Doppelpunkt steht der jeweilige Typ des Instanzattributs bzw. der Rückgabotyp bei den Methoden.

4.3.1. Testdatengenerator

Um den oben vorgestellten Algorithmus testen zu können wurde ein Testdatengenerator geschrieben (vgl. Abbildung 18). Dieser erzeugt eine vom Benutzer vorgegebene Anzahl an Testobjekten und AAs.

Die Testobjekte sind in dieser Implementierung vom Typ `CPoint` aus den Nexus Geoklassen und bestehen aus einem Koordinatenpaar deren räumliches Bezugssystem (*Spatial Reference System*) in diesem Fall *DHDN Gauss Kruger Stuttgart 64235* ist. Die Objekte werden von der Methode *generateObjects* zufällig auf einer Fläche von der Größe der Bundesrepublik Deutschland verteilt.

Die AAs sind vom Geoklassentyp `CPolygon`. Sie werden durch die Methode *generateAAs* zufällig erzeugt und auf der gleichen Fläche wie die oben erzeugten Objekte verteilt. Die Anzahl der Ecken wird für jedes Polygon zufällig gewählt (die Werte liegen bei dieser Implementierung zwischen 3 und 10), ebenso die Größe.

Die Methode *generateSpaSes* bildet aus den erzeugten Testobjekten und den erzeugten AAs SpaSes vom Typ `SpaSe`, welcher in Abschnitt 4.3.3 genauer beschrieben wird. Hierzu wird für jedes erzeugte Testobjekt mit Hilfe der Nexus Geoklassen überprüft mit

GenerateTestdata
<pre> -aa : CPolygon[] -areaHeight : double -areaIndexByGeo : RS = new RS() -areaWidth : double -asr : ASR -dummyAsr : ASR -modifySpaSe : SpaSe -numAAs : int -numObjects : int -point : CPoint[] -randomseed : long -randPolygon : RandomPolygon -spaSe : SpaSe[] -srs : CSpatialReferenceSystem = CProjectedCoordinateSystem.DHDNGaussKrugerStuttgart_64235 -unusedPoints : Vector -usedPoints : Vector -generateAAs() : void -generateObjects() : void -generateSpaSes() : void +GenerateTestdata(no : int, nas : int, seed : long) +getASR() : ASR +getSpaSes() : SpaSe[] -getSpaSeToInsertPoint(point : CPoint) : Integer +getUnusedObjects() : Vector +getUsedObjects() : Vector +showAAs() : void </pre>

Abbildung 18: *GenerateTestdata* - Testdatengenerator für den Algorithmus

welchen AAs es überlappt. Falls ein Testobjekt mit mehreren AAs überlappt, wird zufällig eine der überlappenden AAs ausgewählt und das Objekt dieser zugeordnet. Dies geschieht in der Methode *getSpaSeToInsertPoint*. Objekte, die mit keiner AA überlappen werden in dem Java Vector *unusedPoints* gespeichert, alle anderen werden zusätzlich zu deren Speicherung in einem SpaSe in dem Java Vector *usedPoints* gespeichert. Diese beiden Java Vektoren können für Testzwecke jeweils über die Methoden *getUnusedObjects* bzw. *getUsedObjects* abgefragt werden. Ebenfalls in der Methode *generateSpaSes* wird ein ASR für die SpaSes erzeugt. Das ASR wird in Abschnitt 4.3.2 näher beschrieben.

Die erzeugten SpaSes können mit der Methode *getSpaSes* von dem Testdatengenerator abgefragt werden, welcher die SpaSe Objekte als Array zurück gibt. Das ASR kann mit der Methode *getASR* vom Testdatengenerator abgefragt werden.

4.3.2. ASR

Das hier implementierte ASR (vgl. Abbildung 19) kennt die Anzahl der AAs, die vom Testdatengenerator erzeugt werden, und auch deren Größe und Position. Dies ist die gesamte Funktionalität, die der FNNQ Algorithmus von dem ASR benötigt.

Die AAs werden im ASR in einem R-Baum organisiert, so dass ein schneller Zugriff

auf die AAs gewährleistet wird. Der R-Baum wird in der Methode *buildRTree* aufgebaut.

ASR
-areaIndexByGeo : RS = new RS() -numAAs : int -numObjects : int
+ASR(no : int, nas : int) +buildRTree(spaSe : SpaSe[]) : void +getNumberOfAAs() : int +getSpaSesOverlappingWithGeometry(geometry : CGeometry) : Set

Abbildung 19: ASR

In der Methode *getSpaSesOverlappingWithGeometry* berechnet das ASR, welche AAs mit einer bestimmten Geometrie (Punkt, Polygon, usw.) überlappen und gibt diese als Java *LinkedHashSet* zurück. Das *LinkedHashSet* wurde gewählt, da beim Iterieren über ein solches Set die Reihenfolge der Werte in dem Set bei gleicher Anfrage stets gleich ist. So sind die Testergebnisse besser vergleichbar.

Über die Methode *getNumberOfAAs* kann die Anzahl der im ASR gespeicherten AAs abgefragt werden. Die Anzahl der AAs wird als Abbruchkriterium benötigt. So kann überprüft werden ob schon alle AAs vom Algorithmus besucht wurden. Sobald die Anzahl der markierten Server größer ist als die Anzahl aller vom ASR gespeicherten AAs stoppt der Algorithmus. Dies ist der Fall wenn vom Benutzer mehr nächste Nachbarn angefordert werden als insgesamt Objekte existieren.

4.3.3. SpaSe

Zur Vereinfachung speichert jeder SpaSe, in der hier vorgestellten Implementierung (vgl. Abbildung 20), nur eine AA. Ein SpaSe hat somit nur ein einziges Zuständigkeitsgebiet. Mit der Methode *getArea* kann das Zuständigkeitsgebiet eines SpaSes abgefragt werden. Der SpaSe liefert dieses als Geoklassentyp *CPolygon* zurück. Mit der Methode *getNumPoints* kann die Anzahl der vom SpaSe gespeicherten Objekte abgefragt werden. Mit der Methode *getSpaSeIdent* kann die Identifikationsnummer des SpaSe abgefragt werden. Diese wurde vom Testdatengenerator erzeugt. Mit Hilfe dieser Nummer kann ein SpaSe eindeutig identifiziert werden. Dies ist notwendig, um herausfinden zu können, welcher SpaSe schon besucht worden ist, damit ein SpaSe nicht mehrmals vom Algorithmus abgefragt wird. Mit der Methode *insert* kann ein weiteres Objekt in die AA, die der SpaSe speichert, hinzugefügt werden.

SpaSe
<pre> -area : CPolygon -numpoints : int -points : Vector -spaSeIdent : int -srs : CSpatialReferenceSystem = CProjectedCoordinateSystem.DHDNGaussKrugerStuttgart_64235 </pre>
<pre> +getArea() : CPolygon +getNearestObjects(point : CPoint, numberNearestNeighbors : int) : Map +getNumPoints() : int +getSpaSeIdent() : Integer +insert(point : CPoint) : void +SpaSe(a : CPolygon, ident : int) </pre>

Abbildung 20: *SpaSe*

Die Methode *getNearestObjects* implementiert die in Kapitel 2.4 besprochene Funktion der Server, die die nächsten Nachbarn zu einem Anfrageort berechnet. So muss jeder *SpaSe* nur einmal abgefragt werden, was Zeit spart. Werden mehr nächste Nachbarn von einem *SpaSe* angefordert als dieser Objekte hat, so gibt dieser all seine Objekte zurück.

4.3.4. FNNQ

In der Klasse *FNNQ* ist der in Kapitel 3.2 vorgestellte Algorithmus implementiert.

FNNQ
<pre> -asr : ASR -bufferSize : double -location : CPoint -markedSpaSes : Set = new HashSet() -nearestNeighbors : TreeMap = new TreeMap() </pre>
<pre> -cutNNTree(numberOfNearestNeighbors : int) : void +FNNQ(p : CPoint, a : ASR) -fnnqLoop(tempSet : Set, oldResults : TreeMap, numberOfNearestNeighbors : int, mode : int) : TreeMap +getFNN(numberOfNearestNeighbors : int) : TreeMap -getNearestNeighborsFromResults(results : TreeMap, numberOfNearestNeighbors : int) : TreeMap -getSpaSesToQuery(buffer : CGeometry) : Set -resizeBuffer(results : TreeMap, numberOfNearestNeighbors : int, oldBuffer : CGeometry) : CGeometry </pre>

Abbildung 21: *FNNQ*

Dem Konstruktor dieser Klasse wird der Anfrageort als Geoklassentyp *CPoint* übergeben und das zuständige ASR. Die Anzahl der gesuchten k nächsten Nachbarn wird nicht im Konstruktor der Klasse festgelegt. Dies hat den Vorteil, dass für einen Anfrageort mehrmals eine *FNNQ* gestartet werden kann, auch mit unterschiedlichem k .

Die Methode *getFNN* ist das Kernstück der Klasse. Diese Methode berechnet die k nächsten Nachbarn zu dem aus dem Konstruktor bekannten Anfrageort. k entspricht

dem ganzzahligen Parameter `numberOfNearestNeighbors` mit dem die Methode aufgerufen wird. Die von `getFNN` berechneten nächsten Nachbarn werden in dem Attribut `nearestNeighbors` (welches ein Instanzattribut ist) vom Typ `TreeMap` gespeichert. Von diesem Instanzattribut wird profitiert, wenn `getFNN` mehrmals aufgerufen wird. Wenn nach einer oder mehreren Berechnungen von nächsten Nachbarn, durch `getFNN`, vom Benutzer weniger nächste Nachbarn angefordert werden als zuvor schon einmal berechnet wurden, muss keine neue Berechnung gestartet werden. Es wird lediglich die angeforderte Anzahl an nächsten Nachbarn aus der `nearestNeighbors` `TreeMap` zurückgeliefert. Ebenso wird von `nearestNeighbors` profitiert wenn mehr nächste Nachbarn angefordert werden als schon berechnet wurden. Der Pufferradius für die Konvergenzphase ist geringer wenn schon einmal nächste Nachbarn berechnet wurden. Es existieren dann schon Objekte in der Ergebnismenge, von denen der Algorithmus Gebrauch macht, die im Initialschritt nicht gefunden werden.

Die Berechnung der nächsten Nachbarn findet nicht ausschließlich in der Methode `getFNN` statt. `getFNN` ruft die Methode `fnnqLoop` drei mal auf. Einmal im Initialschritt, in der inkrementellen Phase in der der Puffer vergrößert wird, wenn noch nicht genügend nächste Nachbarn gefunden worden sind und schließlich in der Konvergenzphase des Algorithmus, wenn die endgültige Ergebnismenge berechnet wird. `fnnqLoop` ist die Implementierung der Schleife aus Kapitel 3.2.1, Schritt 3. Die Methode `fnnqLoop` kennt drei Modi (0,1,2), da die Initialphase, die inkrementelle Phase und die Konvergenzphase nicht identisch sind. Der Modus, in welchem die Schleife laufen soll wird beim Aufruf der Methode als Parameter übergeben. Im Modus 0, der Initialphase, werden k nächste Nachbarn in den AAs gesucht mit denen der Anfrageort überlappt. Modus 1 ist die inkrementelle Phase, wenn noch nicht genügend nächste Nachbarn in der Initialphase gefunden worden sind. In diesem Modus wird die Schleife unterbrochen sobald nach einer Puffervergrößerung genügend nächste Nachbarn gefunden werden. Im Modus 2 werden die endgültigen nächsten Nachbarn berechnet. Wir wissen in dieser Phase, dass mindestens `numberOfNearestNeighbors` Objekte gefunden wurden. Wenn in einem Schleifendurchgang, in diesem Modus, Objekte gefunden werden, deren Distanz zum Anfrageort kleiner ist als die Distanz derer aus der `nearestNeighbors` Menge, so wird `nearestNeighbors` unter Berücksichtigung der neu gefundenen Ergebnisse erneut aufgebaut. Auch werden in diesem Modus nur `SpaSes` abgefragt deren AAs näher am Abfrageort sind als das am weitesten entfernte Objekt aus `nearestNeighbors`, da nur dort neue nächste Nachbarn zu erwarten sind. Der Pufferradius wird sozusagen

dynamisch an die aktuelle Ergebnismenge `nearestNeighbors` angepasst, was zeitaufwendige Anfragen an `SpaSes` außerhalb des Pufferradius spart.

Weiterhin gibt es in dieser Klasse noch vier Helfermethoden. `getNearestNeighborsFromResults`, `cutNNTree`, `getSpaSesToQuery` und `resizeBuffer`. `getNearestNeighborsFromResults` gibt die ersten n Elemente aus einer an die Methode übergebenen `TreeMap` als neue `TreeMap` zurück, wobei $n \leq |uebergebene\ TreeMap|$. `cutNNTree` benutzt `getNearestNeighborsFromResults` um die Menge `nearestNeighbors` auf eine bestimmte Anzahl von Elementen zu kürzen. `getSpaSesToQuery` wird benötigt um die `SpaSes` zu erhalten, die mit dem Puffer überlappen. `getSpaSesToQuery` ruft `getSpaSesOverlappingWithGeometry` im ASR auf und erhält somit die `SpaSes` welche mit dem Puffer überlappen. `resizeBuffer` vergrößert den Puffer. Diese Methode wird in der inkrementellen Phase des Algorithmus benötigt.

4.4. Evaluierung

Im folgenden Kapitel werden Messungen, die mit der oben vorgestellten Implementierung des Algorithmus durchgeführt wurden, aufgezeigt. Die Testobjekte und `SpaSes` wurden von dem Testdatengenerator aus Abschnitt 4.3.1 erzeugt. Jede Anfrage wurde von 1000 verschiedenen und zufällig generierten Anfrageorten gestartet. Es wurde jeweils die maximale, die minimale und die durchschnittliche Anzahl der von den `SpaSes` erhaltenen Objekte pro Anfrage berechnet. Ebenso wurde bei jeder Anfrage die maximale, die minimale und die durchschnittliche Anzahl der besuchten Server pro Anfrage festgehalten.

In Abbildung 22 ist das Ergebnis einer Messung zu sehen, bei der die Anzahl der gesuchten nächsten Nachbarn (`numNN`) und die generierten AAs konstant sind und die Anzahl der vom Testdatengenerator erzeugten Testobjekte variiert. Bei dieser Messung wurden 1000 AAs erzeugt und es wurde nach den 10 nächsten Nachbarn gefragt. Die Zahl der generierten Testobjekte lag zwischen 1.000 und 100.000. Bei 1.000 Testobjekten gibt es ein 1:1 Verhältnis zwischen der Anzahl der AAs und der Testobjekte, was bedeutet, dass bei einer Gleichverteilung der AAs und der Testobjekte jede AA ein Objekt enthält. Bei 100.000 Testobjekten hat jede AA bei einer Gleichverteilung 100 Objekte und somit eine höhere Objektdichte.

Die in Abbildung 22 dargestellte Messung zeigt, dass die durchschnittliche Anzahl der Serveranfragen mit zunehmender Objektdichte anfangs sehr stark abnimmt und sich später stabilisiert. Die durchschnittliche Anzahl der von den `SpaSes` gelieferten

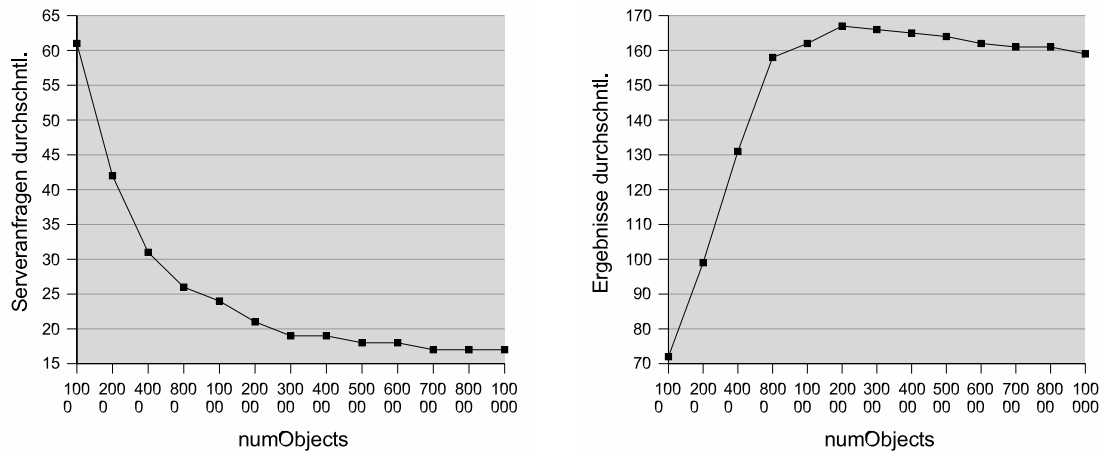


Abbildung 22: Durchschnittliche Anzahl besuchter Server und zurückgelieferter Objekte bei 1000 AAs, 10 numNN und variabler Anzahl von Testobjekten

Ergebnissen nimmt erst sehr stark zu und stabilisiert sich ebenfalls später. Die beiden Messkurven stabilisieren sich ab dem Zeitpunkt, zu dem die mittlere Anzahl an Objekten pro AAs mindestens so groß ist wie die Anzahl der gesuchten nächsten Nachbarn. Bei dieser Messung fangen die beiden Kurven ab etwa 10.000 Testobjekten an sich zu stabilisieren. Wenn die Zahl der Objekte je AAs im Mittel geringer ist als die Anzahl der gesuchten nächsten Nachbarn, werden mehr SpaSes besucht, als dann, wenn sich in jeder AA mindestens so viele Objekte befinden wie nächste Nachbarn gesucht werden (in diesem Fall haben die AAs einen hohen Sättigungsgrad). Bei niedrigem Sättigungsgrad entstehen jedoch weniger überflüssige Ergebnisse, die später verworfen werden müssen, als bei einem hohen Sättigungsgrad. Bei dieser Messung werden bei 1.000 Testobjekten 61 SpaSes besucht und 72 Ergebnisse von den SpaSes zurückgeliefert. Davon werden 62 wieder verworfen. Bei 30.000 Testobjekten werden 19 SpaSes abgefragt und 166 Ergebnisse von den besuchten SpaSes zurückgeliefert, was bedeutet, dass 156 Ergebnisse verworfen werden müssen. Dieses Verhalten ist dadurch zu erklären, dass der Algorithmus versucht von jedem besuchten SpaSe zwischen 0 und 10 Objekte (bei dieser Messung) abzufragen. Bei einer hohen Objektdichte werden weniger SpaSes abgefragt um mindestens 10 Objekte zu finden. Gleichzeitig aber erhalten wir von SpaSes mit gesättigten AAs mehr Ergebnisse als von SpaSes deren AAs nicht gesättigt sind. Das heißt, dass die Zahl der von den SpaSes erhaltenen Ergebnisse von dem Verhältnis zwischen der Objektdichte in den AAs und der Anzahl der gesuchten nächsten Nachbarn abhängt.

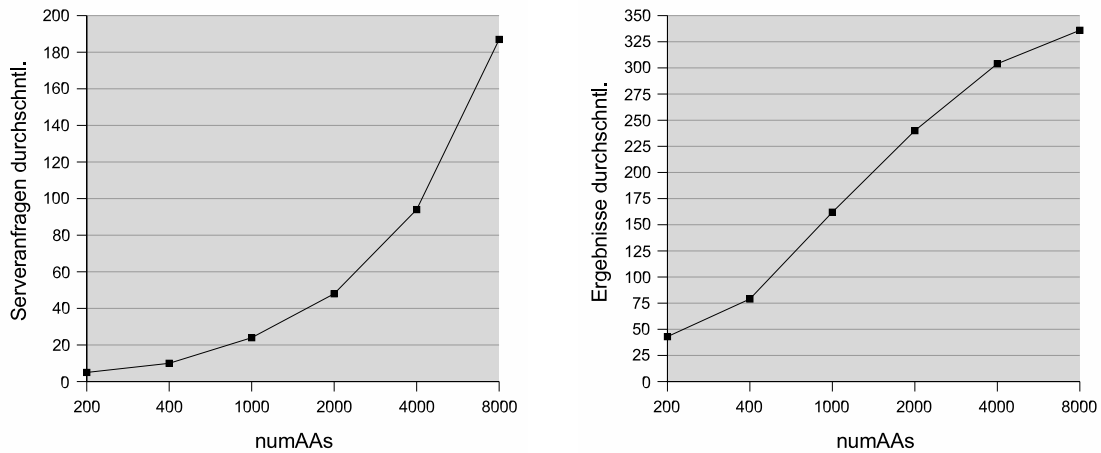


Abbildung 23: Durchschnittliche Anzahl besuchter Server und zurückgelieferter Objekte bei 10.000 Testobjekten, 10 numNN und variabler Anzahl von AAs

In Abbildung 23 ist das Ergebnis einer Messung mit einer konstanten Anzahl von gesuchten nächsten Nachbarn und 10.000 Testobjekten zu sehen. Bei dieser Messung variiert die Anzahl der vom Testgenerator erzeugten AAs. Zu beobachten ist hier, dass sowohl die Anzahl der Serveranfragen als auch die Zahl der Ergebnisse mit zunehmender Anzahl von AAs steigt. Dies liegt daran, dass der Algorithmus immer alle SpaSes abfragt deren AAs mit dem Anfrageorte bzw. mit den Puffern, die der Algorithmus bei der Suche erzeugt, überlappen. Bei einer hohen Anzahl von AAs gibt es mehr Überlappungen und folglich mehr Serveranfragen. Je mehr SpaSes abgefragt werden desto mehr Ergebnisse werden zurückgeliefert. Die Steigung der Kurve, die die Anzahl der Ergebnisse bei zunehmender Anzahl von AAs zeigt, nimmt bei einer großen Anzahl von AAs ab. Dies liegt daran, dass die Objektdichte pro AA und damit ihr Sättigungsgrad sinkt. Wie wir bei der vorherigen Messung gesehen haben sinkt die Anzahl der Ergebnisse bei abnehmendem Sättigungsgrad.

Als Grundlage für die Messung, deren Ergebnis in Abbildung 24 zu sehen ist, werden in jedem Testdurchgang jeweils die gleichen 5.000 Testobjekte und 1.000 AAs verwendet. Die Zahl der angeforderten nächsten Nachbarn liegt zwischen 1 und 512. Auch hier kann beobachtet werden, dass sowohl die Zahl der Serveranfragen als auch die Zahl der Ergebnisse mit zunehmender Anzahl von gesuchten nächsten Nachbarn zunimmt. Die Zunahme ist jedoch geringer als linear. Die Anzahl der Serveranfragen und Ergebnisse verdoppelt sich nicht wenn man die Anzahl der gesuchten nächsten Nachbarn verdoppelt.

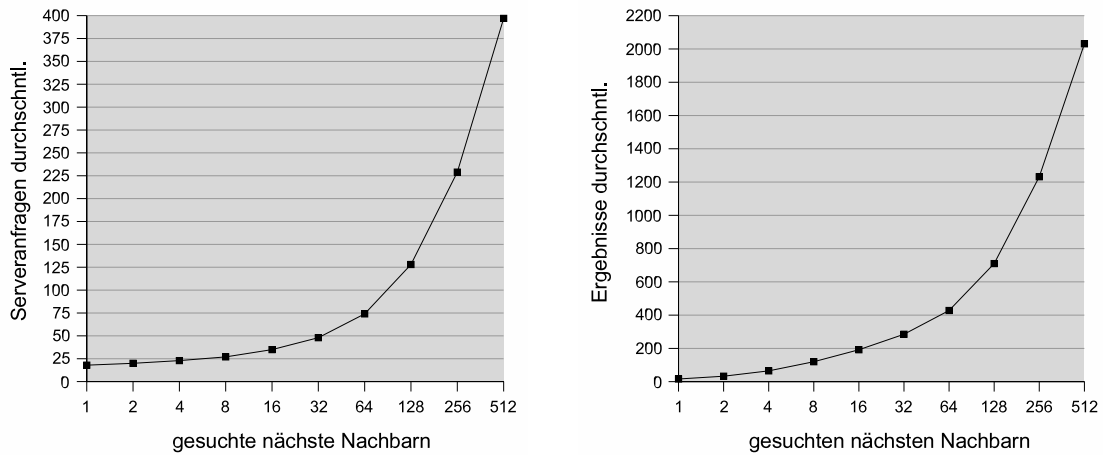


Abbildung 24: Durchschnittliche Anzahl besuchter Server und zurückgelieferter Objekte bei 5.000 Testobjekten, 1.000 AAs und variabler Anzahl von numNN

4.5. Optimierungsvorschläge für die Implementierung

Ein Optimierungsvorschlag für die in Kapitel 4.3 vorgestellte Implementierung ist das Parallelisieren der Anfragen an die SpaSes. Die Parallelisierung kann durch durch Java Threads erreicht werden. Diese Optimierung wirkt sich auf die Laufzeit des Algorithmus aus, würde aber gleichzeitig mehr Netzwerkbandbreite benötigen und eine höhere CPU-Belastung bedeuten. Es gilt hier zu untersuchen, ob eine geringe Bearbeitungszeit bei hoher Netzwerkbandbreite und hoher CPU-Belastung vertretbarer ist als eine hohe Laufzeit bei geringer Netzwerkbandbreite und geringerer CPU-Belastung.

Weiterhin können die Überlegungen aus Kapitel 3.3 für die inkrementelle Phase herangezogen werden. So ist, bei der in Kapitel 4.3 vorgestellten Implementierung, der initiale Pufferradius, wenn keine Ergebnisse aus der initialen Phase vorhanden sind, statisch auf 500m gesetzt. Ebenso wird der Pufferradius, so lange keine Ergebnisse gefunden werden, in jeder weiteren Iteration um den Faktor 2 vergrößert. Der initiale Pufferradius und der Vergrößerungsfaktor könnte, durch Kenntnisse über die Dichte jedes einzelnen Objekttypen, variabel berechnet werden und so zu weniger abgefragten SpaSes und einer geringeren Laufzeit führen.

5. Schlussbemerkungen

Ziel dieser Studienarbeit war es einen Algorithmus zu entwerfen, der das Problem der k nächsten Nachbarn in einer Umgebung mit verteilter Datenhaltung löst. Dabei galt es folgende Dinge zu beachten:

- *die Anzahl der Iterationen sollte minimal gehalten werden:* Bei dem vorgestellten Algorithmus wurde dies dadurch erreicht, dass der Puffer in der inkrementellen Phase nicht wahllos bzw. um einen festen Faktor vergrößert wird. Sein Vergrößerungsfaktor ist nach jeder Iteration, während der inkrementellen Phase, von der Anzahl der bis dahin gefundenen Ergebnisse abhängig.
- *die Zahl der angefragten Server sollte minimal sein:* Dies wurde zum Einen durch den variablen Puffervergrößerungsfaktor erreicht, durch welchen nur SpaSes abgefragt werden, die potentielle nächste Nachbarn enthalten, und zum Anderen werden besuchte SpaSes markiert und nie mehrmals abgefragt. Ebenso wird in der Konvergenzphase kein SpaSe besucht, der weiter vom Anfrageort entfernt ist als das am weitesten entfernte Objekt aus der finalen Treffermenge.
- *die Anzahl der verworfenen Ergebnisse soll minimal sein:* Wie wir bei der Evaluierung gesehen haben hängt die Zahl der von dem Algorithmus verworfenen Ergebnisse im Allgemeinen von dem Verhältnis zwischen der Objektdichte in den AAs und der Anzahl der gesuchten nächsten Nachbarn ab. Der Algorithmus selbst reduziert die Zahl der verworfenen Ergebnisse dadurch, dass in der Konvergenzphase von den noch abzufragenden SpaSes nur noch so viele Objekte angefordert werden wie diese potenzielle nächste Nachbarn liefern können. Wenn beispielsweise 10 nächste Nachbarn gesucht werden und 6 der bisher gefundenen Nachbarn eine geringere Distanz zum Anfrageort haben als der nächste abzufragende SpaSe, dann werden von diesem nur noch 4 Objekte angefordert.
- *maximales Kosten/Nutzen Verhältnis:* Der im Rahmen dieser Studienarbeit vorgestellte Algorithmus berechnet immer die korrekten Ergebnisse. Es ist noch kein geeignetes Konzept entworfen worden, das Iterationen dadurch einspart, dass langsame Server vernachlässigt werden. Es kann ein Flag im ASR angedacht werden, das dem Algorithmus anzeigt, ob ein SpaSe langsam ist. Ebenso muss ein Parameter eingeführt werden, der dem Algorithmus sagt ob er durch das Flag gekennzeichnete, langsame Server bei der Suche ignorieren soll, auf Kosten der Korrektheit der

Ergebnisse. Auch wäre ein Verzicht auf die Konvergenzphase eine Möglichkeit die Zahl der Iterationen und Serveranfragen zu reduzieren.

Ausblick

Die föderierten nächsten Nachbar Anfragen erweitern die Nexus Plattform um eine Komfortanfrage. Ein Nutzer kann mit Hilfe der nächsten Nachbar Anfragen bequem die k nächstgelegenen Tankstellen, Restaurants, usw. abfragen, ohne sich um das Suchgebiet kümmern zu müssen, wie das bei Gebietsanfragen der Fall wäre. Das System liefert dem Nutzer bequem und präzise die nächstgelegenen k Objekte vom jeweils angeforderten Typ.

Um das Einsatzgebiet von föderierten nächsten Nachbar Anfragen in der Nexus Föderations selbst zu vergrößern, kann das Konzept zukünftig auf bewegliche Anfrageorte ausgedehnt werden. Einige Anregungen dazu sind in Kaptiel 2.2 zu finden.

A. Begriffslexikon

Suchraum = Suchgebiet; Gebiet in dem nach Ergebnissen gesucht wird

Query = Anfrage

Nearest Neighbor Query = Nächster Nachbar Anfrage

Netzwerkbandbreite = die Datenmenge, die über das Netzwerk in einen bestimmten Zeitraum transportiert werden kann

Flag = ein binäres Attribut.

B. Verwendete Abkürzungen

bzw. = beziehungsweise

vgl. = vergleiche

durchschntl. = durchschnittlich(e)

ff. = fortfolgende

numNN = Anzahl der gesuchten nächsten Nachbarn

(k)NN = (k) Nearest Neighbor, (k) nächste Nachbarn

(k)NNQ = (k) Nearest Neighbor Query, (k) nächste Nachbar Anfragen

FNNQ = Föderierte Nearest Neighbor Queries

CNNQ = Constrained Nearest Neighbor Queries

Literatur

- [1] Dan-Zhou, Ee-Peng Lim, Wee-Keong Ng: *Efficient k Nearest Neighbor Queries on Remote Spatial Databases Using Range Estimation*, 14th International Conference on Scientific and Statistical Database Management (SSDBM'02) July 24 - 26, 2002 Edinburgh, Scotland p. 121
- [2] Rimantas Benetis, Christian S. Jensen, Gytis Karciauskas, Simonas Saltenis: *Nearest Neighbor and Reverse Neighbor Queries for Moving Objects*, Department of Computer Science, Aalborg University, International Database Engineering and Applications Symposium (IDEAS'02) July 17 - 19, 2002 Edmonton, Canada p. 44
- [3] Nick Roussopoulos, Stephen Kelley, Frederic Vincent: *Nearest Neighbor Queries*. In Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, pp. 71-79, 1995.
- [4] Z. Song, Nick Roussopoulos: *K-Nearest Neighbor Search for Moving Query Point*. In Proceedings of the 7th International Symposium on Spatial and Temporal Databases, pp. 79-96, 2001.
- [5] K. L. Cheung, A. W. Fu: *Enhanced Nearest Neighbor Search on the R-tree*. SIGMOD Record, 27(3):16-21, 1998
- [6] Hakan Ferhatosmanoglu, Ioanna Stanoi, Divyakant, Amr El Abbadi: *Constrained Nearest Neighbor Queries*. SSTD 2001, LNCS 2121, pp. 257-276, 2001
- [7] G. R. Hjaltson and H. Samet: *Ranking in spatial databases*. In Proceedings of the 4th Int. Symp. on Large Spatial Databases, pages 83-95, Portland, ME, 1995
- [8] G. R. Hjaltson and H. Samet: *Distance browsing in spatial databases*. ACM Transactions on Database Systems, 24(2):265-318, 1999
- [9] V. Gaede und O. Guenther: *Multidimensional access methods*. ACM Computer Surveys, 30, 1998.
- [10] A. Guttman: *R-Trees: A dynamic Index Structure for Spatial Searching*. Proceedings of the 1984 ACM SIGMOD, pp. 47-57, 1984

- [11] D. Nicklas, M. Großmann, T. Schwarz, S. Volz, B. Mitschang: *A Model-Based, Open, Arcitecture for Mobile, Spacially Aware Applications*. C.S. Jessen et. al. SSTD 2001, LNCS 2121, pp. 117-135, 2001
- [12] F. Hohl, U. Kubach, A. Leonhardi, K. Rothermel, M. Schwehm: *Next Century Challenges: Nexus - An Open Global Infrastructure for Spatially-Aware Applications*. In Proceedings of the Fifth Anual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '99), Seattle, Washington, USA, August 15-20, 1999, T. Imielinski, M. Steenstrup, (Eds.), ACM Press, pp 249-255, 1999.
- [13] Implementation of a Generic Indexing Structure in Persistent Java. URL: <http://people.cs.uct.ac.za/~evoges/web/>
- [14] JTS Topology Suite (JTS). URL: <http://www.vividsolutions.com/projects.asp>
- [15] Java Development Kit von Sun Microsystems Inc.. URL: <http://java.sun.com>

Ich versichere, dass ich diese Arbeit selbständig verfasst und nur die angegebenen Hilfsmittel verwendet habe.

Markus Iofcea