Institut f’ur Informatik
Lehr- und Forschungseinheit
f’ur Datenbanksysteme

LMU
Ludwig
Maximilians
Universität
München

Diplomarbeit

Inkrementelles hierarchisches Clustering

Elke Achtert

Aufgabensteller: Prof. Dr. Hans-Peter Kriegel
Betreuer: Karin Kailing, Peer Kröger
Abgabetermin: 23.03.2004

Erkl"arung

Hiermit versichere ich, da"s ich diese Diplomarbeit selbst"andig verfa"st und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 23.03.2004

. . . .
Elke Achtert

Zusammenfassung

In einem sogenannten Data Warehouse werden alle in einem Unternehmen oder einer Organisation vorhandenen Informationen gespeichert. Mit Hilfe automatisierter Methoden zur Datenanalyse werden diese oft unstrukturierten Informationen aufbereitet und daraus neues Wissen extrahiert. Typischerweise werden Änderungen in einem Data Warehouse nicht sofort, sondern in periodischen Abständen kumuliert ausgeführt. Nach einem solchen Update müssen unter anderem auch die Ergebnisse der Clusteranalyse aktualisiert werden. Dabei ist es aus Performanzgründen erforderlich, die Clusteranalyse nicht noch einmal auf die gesamten Datenmenge anzuwenden, sondern inkrementelle Clusteringverfahren einzusetzen, die nur noch die Objekte behandeln, die eingefügt bzw. gelöscht wurden.

Im ersten Abschnitt der vorliegenden Arbeit wird eine inkrementell arbeitende Version des hierarchischen Clusteringverfahrens Single-Link erarbeitet. Dazu wird der bereits bestehende Algorithmus SLINK von R. Sibson erweitert, so dass neben dem Einfügen von Objekten in ein bestehendes Single-Link-Dendrogramm auch das Löschen von Objekten aus einem Dendrogramm möglich ist. Das Einfügen von Objekten wird dabei in linearer Laufzeit unterstützt, das Löschen besitzt verfahrensbedingt durch den Single-Link-Ansatz eine quadratische Laufzeit.

Der zweite Abschnitt dieser Diplomarbeit beschäftigt sich mit der Entwicklung einer inkrementellen Version des dichtebasierten hierarchischen Clusteringverfahrens OPTICS. Das Originalverfahren generiert eine Permutation der Datenmenge, so dass Objekte eines Clusters in zusammenhängenden Intervallen auftreten, d.h. in der Permutation aufeinander folgen. Die erarbeitete inkrementelle Version von OPTICS unterstützt dabei die effiziente dynamische Aktualisierung dieser sogenannten Clusterordnung, indem die Objekte, die von einer Updateoperation betroffen sind, geeignet determiniert und aktualisiert werden. Die Evaluierung des inkrementellen OPTICS Algorithmus zeigt eine signifikante Laufzeitverbesserung gegenüber dem Originalverfahren.

Inhaltsverzeichnis

1 Einleitung **3**

2 Grundlagen **5**

2.1 Knowledge Discovery in Databases 5

2.2 Clusteranalyse 7

2.2.1 Typen von Clusteringverfahren 7

2.2.2 Distanzfunktion als Ähnlichkeiten 9

3 Hierarchisches Clustering **10**

3.1 Mathematische Struktur 10

3.2 Repräsentation der Struktur durch ein Dendrogramm 11

3.3 Der HACM-Algorithmus 13

3.4 Single-Link und Varianten 14

3.5 Die Lance-Williams Rekursionsformel 16

4 Inkrementelles Single-Link **18**

4.1 Die Pointer-Repräsentation 18

4.1.1 Überführung eines Dendrogramms in die Pointer-Repräsentation 19

4.1.2 Überführung der Pointer-Repräsentation in ein Dendrogramm 19

4.1.3 Überführung der Pointer-Repräsentation in eine Baumstruktur 20

4.2 Einfügen neuer Objekte 21

4.3 Löschen bestehender Objekte 24

4.4 Bewertung 30

5 Dichtebasiertes hierarchisches Clustering **31**

5.1 Cluster als dichteverbundene Mengen 31

5.2 Algorithmus DBSCAN 34

5.3 Dichtebasierte Clusterordnung 36

5.4 Algorithmus OPTICS 39
5.5 Visualisierung der Clusterstruktur 41

6 Inkrementelles OPTICS 43
6.1 Der RDkNN-Baum 44
6.1.1 Grundlagen 44
6.1.2 Struktur des RDkNN-Baums 45
6.1.3 Reverse-k-Nearest-Neighbor-Anfrage 46
6.1.4 k-Nearest-Neighbor-Anfrage 47
6.1.5 Einfügen und Löschen von Punkten 48
6.2 Algorithmus Inkrementelles OPTICS 54
6.2.1 Definitionen 54
6.2.2 Einfügen eines neuen Objekts 55
6.2.3 Löschen eines bestehenden Objekts 59
6.2.4 Einfügen und Löschen mehrere Objekte 62
6.3 Experimentelle Untersuchung 64
6.3.1 Einfügen und Löschen einzelner Punkte 64
6.3.2 Einfügen und Löschen mehrerer Punkte 66

7 Zusammenfassung 69
7.1 Ergebnis 69
7.2 Ausblick 70

Abbildungsverzeichnis 71

Literaturverzeichnis 74

Kapitel 1

Einleitung

> An intelligent being cannot treat every object it sees as a unique entity unlike anything else in the universe. It has to put objects in categories so that it may apply its hard-won knowledge about similar objects encountered in the past, to the object at hand.
>
> Steven Pinker, *How the Mind Works*, 1997.

Die Fortschritte in der Informationstechnologie ermöglichen es, große Mengen an Informationen aus den verschiedensten Bereichen relativ einfach zu erfassen und in sogenannten Data Warehouses zu speichern. Diese Daten enthalten potentiell wichtiges Wissen, deren manuelle Analyse die menschlichen Kapazitäten bei weitem übersteigt. In diesen Fällen bieten sich automatisierte Methoden zur Datenanalyse an, die unter dem Begriff Knowledge Discovery in Databases (KDD) zusammengefasst werden. Ein wichtige Funktion des KDD-Prozesses ist das sogenannte Data Mining. Darunter versteht man die Anwendung effizienter Algorithmen, um die in einer Datenbank gültigen Muster zu finden [FHS96]. Zu den Hauptaufgaben des Data Mining zählt das Clustering, dessen Ziel es ist, Daten mit ähnlichen Merkmalen (semi-) automatisch zu identifizieren und anhand dieser in Gruppen (Cluster) einzuteilen, die sich möglichst stark voneinander unterscheiden.

Typischerweise werden in einem Data Warehouse neue Daten nicht sofort eingefügt und alte Daten auch nicht sofort gelöscht. Die Änderungen werden vielmehr gesammelt und in periodischen Abständen in einem Schritt ausgeführt. Durch ein solches Update verändern sich unter anderem auch die Cluster, die durch die Clusteranalyse erkannt wurden. Für die Nutzbarkeit des Data Warehouses ist es von sehr großer Bedeutung, gleichzeitig die Ergebnisse dieser Clusteringverfahren zu aktualisieren. Bei sehr großen Datenmengen besitzen solche Verfahren häufig relativ hohe Laufzeiten. In einem Data Warehouse ist es daher sinnvoll, die Clusteranalyse nach einem Update nicht noch

einmal auf die gesamten Datenmenge anzuwenden, sondern nur noch die Punkte zu behandeln, die eingefügt bzw. gelöscht wurden. Ein solches sogenanntes inkrementelles Verfahren wirkt sich im allgemeinen positiv auf die Laufzeit des Algorithmus aus.

Im Rahmen dieser Diplomarbeit werden inkrementell arbeitende Versionen der hierarchischen Clusteringverfahren Single-Link und OPTICS entwickelt. Dazu werden einleitend in Kapitel 2 zunächst die Hauptaufgaben des KDD-Prozesses beschrieben und die Grundlagen zum Teilbereich der Clusteranalyse behandelt. Im Anschluss daran werden die dabei zum Einsatz kommenden unterschiedlichen Clusteringverfahren erörtert. Kapitel 3 behandelt die grundlegenden Methoden und Techniken des hierarchischen Clusterings und stellt dessen bekanntesten Vertreter Single-Link mit seinen Varianten vor. Das nachfolgende Kapitel 4 beschäftigt sich mit dem Entwurf und der Implementierung eines inkrementellen Single-Link Algorithmus. Die theoretischen Grundlagen des dichtebasierten und hierarchischen dichtebasierten Clustering sind Inhalt des Kapitels 5 und werden am Beispiel der beiden Verfahren DBSCAN und OPTICS erläutert. Anschließend beschäftigt sich Kapitel 6 mit der Herleitung und effizienten Implementierung einer inkrementell arbeitenden Version des OPTICS-Algorithmus. Die Auswertung der experimentellen Evaluation dieses Ansatzes wird im letzten Abschnitt des Kapitels behandelt. Kapitel 7 schließt mit einem Fazit und gibt einen Ausblick auf zukünftige Erweiterungen der vorgestellten Lösungsansätze.

Kapitel 2

Grundlagen

In diesem Kapitel wird das Gebiet des Knowledge Discovery in Databases anhand der fundamentalen Begriffe und der wichtigsten Aufgaben eingeführt. Anschließend werden die Grundlagen zum Teilbereich der Clusteranalyse behandelt und die dabei zum Einsatz kommenden unterschiedlichen Clusteringverfahren skizziert.

2.1 Knowledge Discovery in Databases

Knowledge Discovery in Databases (KDD) [FHS96] ist der Prozess der (semi-) automatischen Extraktion von Wissen aus Datenbanken, das

- gültig im statistischen Sinne
- bisher unbekannt und
- potentiell nützlich für eine gegebene Anwendung ist.

KDD ist ein iterativer Prozess, der aus den Abbildung 2.1 illustrierten Schritten besteht, die im folgenden kurz erläutert werden (vgl. [ES00]).

Im ersten Schritt des KDD-Prozesses wird determiniert, in welchen Daten Wissen zu suchen ist, wie diese Daten zu beschaffen sind und in welchen Strukturen sie verwaltet werden. Im einfachsten Fall kann man dabei auf eine vorhandene Datenbank zurückgreifen; ggf. müssen die Daten aber auch erst durch Messungen, Fragebögen oder ähnlichem erhoben werden.

Ziel der Vorverarbeitung ist es, die benötigten Daten aus unterschiedlichen Quellen in ein System zu integrieren, auf Konsistenz zu prüfen und fehlende Daten zu vervollständigen. Obwohl diese Aufgaben meist keine konzeptionellen Probleme in sich bergen, umfasst der Aufwand für die Vorverarbeitung in vielen KDD-Projekten einen großen Teil des Gesamtaufwands.

Im nächsten Schritt werden die vorverarbeiteten Daten in eine für das Ziel des KDD geeignete Repräsentation transformiert. Typische Transformationen sind z.B. die Attribut-Selektion, d.h. die Auswahl relevanter Attribute oder die Diskretisierung von numerischen Attributen, falls der im nächsten Schritt eingesetzte Data Mining-Algorithmus nur kategorische Attribute verarbeiten kann.

Unter dem sich an den Transformationsprozess anschließenden Schritt des Data Minings versteht man die Anwendung effizienter Algorithmen, um die in einer Datenbank enthaltenen gültigen Muster zu finden [FHS96]. Zu den wichtigsten Data Mining Aufgaben zählen:

- Clustering / Entdeckung von Ausreißern

  Ziel des Clustering ist die Einteilung einer Datenbank in Gruppen (Cluster), so dass innerhalb einer Gruppe die Objekte möglichst homogen, die Gruppen untereinander aber möglichst heterogen sind. Ausreißer sind Objekte, die keinem der gefundenen Cluster angehören.

- Klassifikation

  Aufgabe der Klassifikation ist es, Objekte aufgrund ihrer Attributwerte einer der vorgegeben Klassen zuzuordnen. Gegeben ist dazu eine Menge von Trainingsobjekten, die bereits einer Klasse zugeordnet sind. Mit Hilfe dieser Trainingsdaten soll eine Funktion gelernt werden, die andere Objekte mit unbekannter Klassenzugehörigkeit in ein der Klassen zuweist.

- Assoziationsregeln

  Assoziationsregeln beschreiben, welche Gruppen von Objekten häufig gemeinsam auftreten. Die bekannteste Anwendung dieser Regeln ist die sogenannte Warenkorbanalyse, in der versucht wird festzustellen, welche Produkte mit einer bestimmten Wahrscheinlichkeit zusammen gekauft werden.

- Generalisierung

  Die Generalisierung fokussiert sich auf die möglichst kompakte Beschreibung einer Datenmenge, indem Attributwerte generalisiert und die Anzahl der Datensätze reduziert werden.

Im letzten Schritt des KDD-Prozesses werden die gefundenen Muster geeignet repräsentiert und in Bezug auf die definierten Schritte evaluiert. Falls die Ziele noch nicht erreicht wurden, wird eine weitere Iteration des KDD-Prozesses initiiert. Sobald die Evaluation erfolgreich ist, wird das vorhandene

Wissen dokumentiert und in das bestehende System als Ausgangspunkt für künftige KDD-Prozesse integriert.

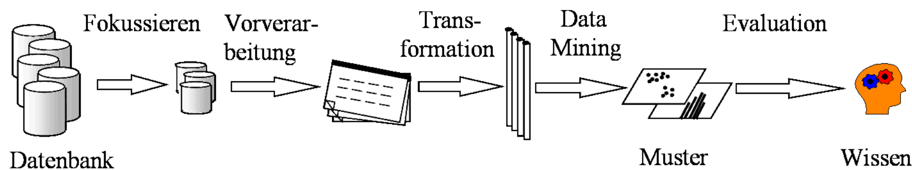


Abbildung 2.1: Die Schritte des KDD-Prozesses

2.2 Clusteranalyse

2.2.1 Typen von Clusteringverfahren

Ziel der Clusteranalyse ist die Partitionierung einer Datenbank in Gruppen von Objekten, sogenannte Cluster, so dass die Elemente innerhalb eines Clusters möglichst ähnlich, die Elemente unterschiedlicher Cluster hingegen möglichst unähnlich zueinander sind [And73]. Man unterscheidet grundsätzlich zwischen den folgenden Methoden zur Clusteranalyse:

- Partitionierende Verfahren
- Hierarchische Verfahren
- Dichtebasierte Verfahren

Partitionierende Clusteringverfahren erzeugen genau eine Partition des Datenraums, indem sie eine Menge von n Objekten in m disjunkte Cluster zerlegen. Dabei gehört jedes Objekt zu genau einem Cluster und jeder Cluster enthält mindestens ein Objekt. Der Anwender gibt bei diesen Verfahren die Anzahl m der gewünschten Cluster vor. Zu Beginn werden aus dem Datenraum m initiale Cluster-Repräsentanten ausgewählt. Diese werden dann solange iterativ modifiziert, bis ein lokales Optimum der Partitionsgüte gefunden wurde. Je nach Verfahren werden dabei unterschiedliche Typen von Cluster-Repräsentanten gewählt, so konstruiert z.B. die k-Means-Methode [Mac67] zentrale Punkte, sogenannte Centroide, der Cluster. Die Verfahren PAM [KR90] und CLARANS [NH94] wählen real vorhandene repräsentative Punkte, sogenannte Medoide, der Datenmenge aus. Da die Verfahren in den einzelnen Iterationsschritten nur lokal optimieren, ist das Ergebnis stark abhängig von

der Wahl der initialen Zerlegung. Auch der Eingabeparameter m, der die Anzahl der zu findenden Cluster spezifiziert, besitzt einen großen Einfluss auf die Qualität des Clusteringergebnisses.

Im Gegensatz zu partitionierenden Verfahren erzeugen hierarchische Clusteringverfahren eine Sequenz ineinander verschachtelter Partitionen. Es wird eine Baumstruktur generiert, bei der ein Knoten im Baum jeweils einem Cluster entspricht. Die Wurzel des Baums repräsentiert dabei die gesamte Datenbank, die Blätter repräsentieren Cluster, die jeweils nur aus einem einzelnen Objekt bestehen. Bei diesen Verfahren wählt der Anwender aus der erzeugten hierarchischen Struktur der Daten ein geeignetes konkretes Clustering selbst aus. Der hierarchische Ansatz findet vor allem Anwendung in der Molekularbiologie [JD88] und beim Clustern von Textdaten [EHW89]. Bekannte hierarchische Clusteringverfahren sind das Single-Link Verfahren und seine Varianten, die in Kapitel 3.4 ausführlich behandelt werden. Es existieren zwei Arten von hierarchischen Algorithmen:

- Agglomerative Algorithmen erzeugen den Baum „bottom-up“: sie beginnen mit den einzelnen Objekten und verschmelzen diese schrittweise zu neuen Clustern, bis sich alle Objekte in einem einzigen Cluster befinden. Dabei werden immer diejenigen Cluster zusammengefasst, die maximale Ähnlichkeit zueinander besitzen. Agglomerative Methoden zählen zu den am weitesten verbreiteten hierarchischen Clusteringverfahren [ESM01]. In Kapitel 3.3 wird der allgemeine agglomerative Algorithmus vorgestellt.

- Divisive (teilende) Methoden erzeugen die Baumstruktur in umgekehrter Reihenfolge „top-down“: sie beginnen mit dem kompletten Objektraum und zerlegen ihn schrittweise in einzelne Cluster, bis jedes Objekt einen eigenen Cluster darstellt. Diese Algorithmen sind sehr zeitintensiv, da in jedem Schritt die $2^{k-1} - 1$ möglichen Zerlegungen eines k-elementigen Clusters in zwei Subcluster untersucht werden müssen. Wegen seiner hohen Laufzeitkomplexität ist dieser Ansatz weitaus weniger verbreitet als der agglomerative Ansatz [ESM01].

Die Grundidee des dichtebasierten Clusterings ist es, Cluster als Gebiete im d-dimensionalen Raum anzusehen, in denen eine hohe Objektdichte vorherrscht. Diese dichten Gebiete sind getrennt durch Gebiete, in denen die Objekte weniger dicht beieinander liegen. Für dichtebasierte Cluster gilt, dass für jedes Objekt eines Clusters die lokale Punktdichte einen gegebenen Grenzwert überschreitet und die Objekte innerhalb eines Clusters räumlich zusammenhängend sind. Ziel eines dichtebasierten Clusteringverfahrens ist es, diese dichten Gebiete zu identifizieren. Ein solcher dichtebasierter Algorithmus ist z.B. DBSCAN [EKSX96], der in Kapitel 5.2 behandelt wird. In Kapitel 5.4

wird das Verfahren OPTICS vorgestellt, das den dichtebasierten Ansatz mit einer hierarchischen Datenrepräsentation vereinigt.

2.2.2 Distanzfunktion als Ähnlichkeiten

Um Objekte in Cluster zusammenfassen zu können, muss ihre Ähnlichkeit zueinander bestimmt werden. Meist wird die Ähnlichkeit zweier Objekte durch eine sogenannte Distanzfunktion d spezifiziert: je ähnlicher zwei Objekte x und y sind, desto kleiner ist der Wert von d.

Definition 2.1 (Distanzfunktion). Die Funktion d ist eine Distanzfunktion, falls sie folgende Bedingungen erfüllt:

- Positiv definit: $\text{dist}(x, y) \geq 0$ und $\text{dist}(x, y) = 0 \Leftrightarrow x = y$
- Symmetrie: $\text{dist}(x, y) = \text{dist}(y, x)$

Beispiele von Distanzfunktionen

Die Qualität einer Clusteranalyse hängt sehr stark von der Wahl der passenden Distanzfunktion ab. Nachfolgend sind einige typische Beispiele für Distanzfunktionen auf Objekten $x = (x_1, \ldots, x_n)$ und $y = (y_1, \ldots, y_n)$ mit numerischen Attributwerten x_i bzw. y_i aufgeführt (vgl. [ES00]):

- Allgemeine L_p-Metrik: $\text{dist}(x, y) = \sqrt[p]{\sum_{i=1}^{n} (x_i - y_i)^p}$
- Euklidische-Distanz ($p = 2$): $\text{dist}(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$
- Manhattan-Distanz ($p = 1$): $\text{dist}(x, y) = \sum_{i=1}^{n} |x_i - y_i|$
- Maximums-Metrik ($p = \infty$): $\text{dist}(x, y) = max\{\ |x_i - y_i| \ \mid 1 \leq i \leq n\}$

In der Praxis wird oftmals nicht mit der Distanzfunktion dist gearbeitet, sondern mit einer sogenannten Distanzmatrix. Diese $n \times n$-Matrix hält alle paarweisen Distanzen der n Datenbankobjekte und reicht im allgemeinen als Input für einen Clusteringalgorithmus aus.

Kapitel 3

Hierarchisches Clustering

Hierarchische Clusteringverfahren erzeugen eine Sequenz ineinander verschachtelter Partitionen auf einer Objektmenge, die meist in Form einer speziellen Baumstruktur (Dendrogramm) repräsentiert wird. In diesem Kapitel werden zunächst die mathematischen Grundbegriffe des hierarchischen Clusterings definiert und erläutert. Anschließend wird der am häufigsten eingesetzte Algorithmus unter den hierarchischen Clusteringverfahren, Single Link, mit seinen Varianten vorgestellt.

3.1 Mathematische Struktur

Im folgenden wird die mathematische Struktur definiert, die ein hierarchisches Clusteringverfahren auf einer Objektmenge $\mathcal{O}$ erzeugt.

Definition 3.1 (Partition). [Wil97] Sei $\mathcal{O}$ eine Menge. Eine Partition (oder Zerlegung) von $\mathcal{O}$ ist eine Menge $\mathcal{P} = \{P_1, \ldots, P_n\}$ für die gilt:

- Alle Mengen aus $\mathcal{P}$ sind nicht-leer: $P_i \neq \emptyset$ für $i = 1, \ldots, n$
- Alle Mengen aus $\mathcal{P}$ sind paarweise disjunkt: $P_i \cap P_j = \emptyset$ für $i, j = 1, \ldots, n$, $i \neq j$
- Die Vereinigung der Mengen aus $\mathcal{P}$ ist $\mathcal{O}$: $P_1 \cup P_2 \cup \ldots \cup P_n = \mathcal{O}$

Definition 3.2 (Eingebettete Partition). [JD88] Eine Partition $\mathcal{P}$ ist in eine Partition $\mathcal{Q}$ eingebettet, falls jedes Element von $\mathcal{P}$ eine echte Teilmenge eines Elements von $\mathcal{Q}$ ist.

Z. B. ist $\mathcal{P} = \{\{x_1, x_3\}, \{x_5\}, \{x_2, x_4\}, \{x_6, x_7\}\}$ in $\mathcal{Q} = \{\{x_1, x_3, x_5\}, \{x_2, x_4, x_6, x_7\}\}$ eingebettet.

Ein agglomeratives hierarchisches Clusteringverfahren erzeugt eine Sequenz von n ineinander verschachtelten (eingebetteten) Partitionen $\mathcal{P}_n, \mathcal{P}_{n-1}, \ldots, \mathcal{P}_1$ des Datenraums $\mathcal{O} = \{o_1, \ldots, o_n\}$. Die Teilmengen der einzelnen Partitionen werden Cluster genannt. Dabei besteht die erste Partition $\mathcal{P}_n$ aus n Clustern, die jeweils aus einem einzelnen Objekt bestehen. Die letzte Partition $\mathcal{P}_1$ besteht aus einem einzigen Cluster, der alle n Datenbankobjekte beinhaltet [JD88].

3.2 Repräsentation der Struktur durch ein Dendrogramm

Die hierarchische Struktur der Daten wird grafisch durch ein zweidimensionales Diagramm, ein sogenanntes Dendrogramm dargestellt. Ein Dendrogramm ist eine spezielle Baumstruktur, deren Knoten jeweils einem Cluster entsprechen. Die Wurzel des Dendrogramms repräsentiert dabei einen einzigen Cluster, der alle n Datenbankobjekte beinhaltet. Die n Blätter des Dendrogramms entsprechen jeweils einem Cluster, der nur ein einzelnes Datenbankobjekt enthält. Ein innerer Knoten repräsentiert die Vereinigung seiner beiden Sohnknoten. Eine Kante zwischen zwei Knoten wird dabei proportional zur Distanz zwischen den beiden verbundenen Clustern dargestellt [ES00].

Zur Bestimmung eines konkreten Clusterings einer Datenmenge wird zu einem festen Level h ein horizontaler Schnitt durch das Dendrogramm gelegt. Dabei repräsentieren die Teilbäume mit maximaler Kantenlänge h Cluster, die eine maximale Distanz von h zueinander besitzen. Abbildung 3.1 zeigt ein Beispiel für ein Dendrogramm mit der Zerlegung des Datenraums in vier Cluster.

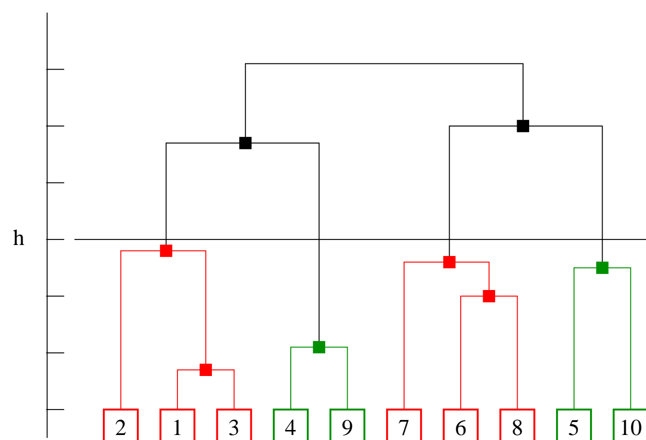


Abbildung 3.1: Dendrogramm: Zerlegung der Datenmenge in vier Cluster

Es gibt diverse Möglichkeiten, eine hierarchische Struktur in Form eines Dendrogramms abzubilden. Beispielsweise könnten im Dendrogramm aus Abbildung 3.1 die Objekte 1 und 3 vertauscht oder der linke Teilbaum der Wurzel als rechter Teilbaum (und umgekehrt) dargestellt werden. Diese Diagramme besitzen jedoch alle die Gemeinsamkeit, dass die Cluster zu jedem Level identisch sind und repräsentieren somit eine identische hierarchische Struktur [JS77].

Ein Dendrogramm kann mathematisch mit Hilfe der zu einer Partition korrespondierenden Äquivalenzrelation beschrieben werden:

Definition 3.3 (Äquivalenzrelation zu einer Partition). [Wil97] Sei $\mathcal{P} = \{P_1, \ldots, P_n\}$ eine Partition einer Menge $\mathcal{O}$. Dann wird durch

$$x \sim y \Leftrightarrow (\exists P_i \in \mathcal{P}) : x \in P_i \wedge y \in P_i$$

eine Äquivalenzrelation $\sim$ definiert, deren Äquivalenzklassen die Mengen P_i sind.

Damit lässt sich nun ein Dendrogramm mathematisch wie folgt definieren:

Definition 3.4 (Dendrogramm). [JS77] Sei $E(\mathcal{O})$ die Menge der Äquivalenzrelationen auf der Objektmenge $\mathcal{O}$. Die Funktion

$$c : [0, \infty[\rightarrow E(\mathcal{O})$$

beschreibt ein Dendrogramm, falls c die folgenden Bedingungen erfüllt:

- $0 \leq h \leq h' \Rightarrow c(h) \subseteq c(h')$
- $c(h)$ ist $\mathcal{O} \times \mathcal{O}$ für ein genügend großes h
  (Man sagt: $c(h)$ ist schließlich $\mathcal{O} \times \mathcal{O}$)
- Für ein gegebenes h existiert ein $\delta > 0$, so dass
  $c(h + \delta) = c(h)$

Legt man zu einem festen Level h einen Schnitt durch das Dendrogramm, dann ist $c(h)$ genau die Äquivalenzrelation, die der Partition der Objektmenge $\mathcal{O}$ entspricht, die durch die Teilbäume mit maximaler Kantenlänge h definiert wird. D.h ein Dendrogramm entspricht im mathematischen Sinn einer Folge von Partitionen zu einem bestimmten Level h, wobei die Partition zu einem entsprechend hohem Level h die komplette Objektmenge $\mathcal{O}$ darstellt.

Für das Beispiel in Abbildung 3.1 ergibt sich die folgende Partition für die Objektmenge $\mathcal{O} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

$$\mathcal{P} = \{\{2, 1, 3\}, \{4, 8\}, \{7, 6, 9\}, \{10, 5\}\}$$

und damit die Äquivalenzrelation

$$\begin{aligned} c(h) = \{&(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9),(10,10),\\ &(2,1),(2,3),(1,2),(1,3),(3,2),(3,1),(4,8),(8,4),\\ &(7,6),(7,9),(6,7),(6,9),(9,7),(9,6),(10,5),(5,10)\} \end{aligned}$$

3.3 Der HACM-Algorithmus

Es existieren diverse Algorithmen zur Berechnung der hierarchischen Baumstruktur einer Objektmenge. Ein einfacher und sehr intuitiver Algorithmus ist der HACM-Algorithmus (Hierarchical agglomerative clustering methods). Er ist in der Literatur wie folgt definiert [FBY92]:

Schritt 1: Ordne jedem Objekt einen eigenen Cluster zu.

Schritt 2: Wähle die beiden Cluster, deren Interclusterdistanz minimal ist und verschmelze diese beiden Cluster.

Schritt 3: Wiederhole Schritt 2, bis nur noch ein Cluster übrig ist.

Als Abbruchkriterium in Schritt 3 kann z.B. auch das Erreichen einer bestimmten Anzahl k von Clustern gewählt werden, dann sind bei n Datenbankobjekten nur $n - k$ statt n Vereinigungsoperationen in Schritt 2 notwendig. Ausgehend von der Distanzfunktion d ergeben sich je nach Verfahren verschiedene Methoden, Distanzen zwischen Clustern zu definieren. Diese werden in Kapitel 3.4 erläutert.

Komplexitätsbetrachtung

Je nach verwendeter Methode zur Bestimmung des Minimums der Interclusterdistanzen im Schritt 2 des HAC-Algorithmus ergeben sich unterschiedliche Komplexitäten. In den nachfolgenden Betrachtungen (vgl. [FBY92]) sind die Kosten für die paarweisen Distanzberechnungen vernachlässigt, sie können jedoch im Falle komplexer, hochdimensionaler Daten stark ins Gewicht fallen.

- Verwendung der Datensätze (stored data):
  Die paarweisen Abstände zwischen den Objekten werden bei Bedarf jeweils neu berechnet. Daraus ergibt sich ein Speicherbedarf von $O(n)$ und eine Laufzeitkomplexität von $O(n^3)$. Die Komplexität der Distanzberechnungen fällt bei diesem Ansatz jedoch stark ins Gewicht.

- Verwendung der Distanzmatrix (stored matrix):
  Die paarweisen Abstände werden einmal zu Beginn berechnet und gespeichert. Der Speicherbedarf für die Matrix beträgt damit $O(n^2)$, die

Initialisierung der Matrix besitzt einen Zeitaufwand von $O(n^2)$, das Erzeugen der Clusterhierarchie erfordert einen Zeitaufwand von $O(n^3)$ ($n^2 \cdot$ lineare Suche).

- Verwendung einer sortierten Distanzmatrix (sorted matrix):
  Dieser Ansatz ist analog zur einfachen Distanzmatrix, allerdings werden die Einträge in der Matrix sortiert, so dass die Suche nach dem Minimum der Interclusterdistanzen die Komplexität $O(1)$ besitzt. Der Speicherbedarf beträgt wie bei der einfachen Distanzmatrix $O(n^2)$, der Zeitaufwand zur Matrixerstellung ergibt sich zu $O(n^2)$, das Sortieren der Matrix nach jedem Vereinigungsschritt benötigt insgesamt $O(n^2 \log n^2)$ Zeit und das Erzeugen der Clusterhierarchie besitzt eine Zeitkomplexität von $O(n^2)$.

3.4 Single-Link und Varianten

Im folgenden werden die drei am weitesten verbreiteten hierarchischen Clusteringmethoden vorgestellt: Single-Link, Complete-Link und Average-Link. Abbildung 3.2 illustriert die unterschiedlichen Distanzmaße der drei Clusteringverfahren.

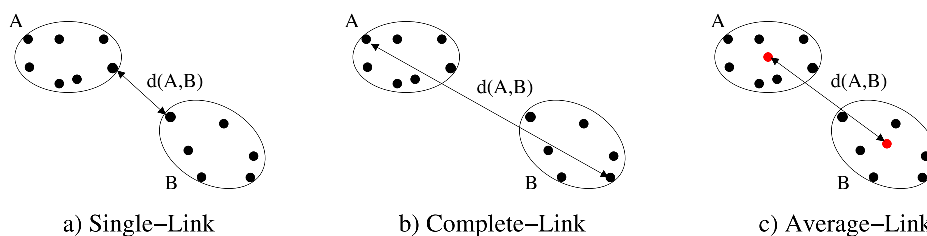


Abbildung 3.2: Single-Link, Complete-Link und Average-Link Distanz

Single-Link

Das Single-Link Verfahren, auch bekannt als Nächste-Nachbarn Methode, ist eines der ältesten und wohl das bekannteste Verfahren zur hierarchischen Clusteranalyse. Es wurde erstmals von Florek et al. 1951 vorgeschlagen [FLP+51]. Der Abstand dist_{SL} zwischen zwei Clustern X und Y wird definiert als der minimale Abstand über alle Elementpaare aus beiden Clustern:

$$\text{dist}_{SL}(X,Y) = \min_{x \in X,\ y \in Y} \text{dist}(x,y)$$

Es existieren sehr viele Algorithmen zum Single-Link Verfahren [Roh82], beispielsweise können Single-Link Cluster auch mit Hilfe der Graphen-Theorie

ermittelt werden. Gower und Ross haben in [GR69] gezeigt, dass aus dem minimalen Spannbaum einer Objektmenge deren Single-Link Hierarchie in $O(n^2)$ Zeit erzeugt werden kann. Single-Link Cluster werden daher auch als maximal verbundene Teilgraphen bezeichnet [JD88]. Ein weiterer effizienter Single-Link Algorithmus ist SLINK, der von Sibson in [Sib73] vorgestellt wurde. SLINK besitzt eine Speicherplatzkomplexität von $O(n)$ sowie eine Laufzeitkomplexität von $O(n^2)$. Dieses Verfahren wird in Kapitel 4 ausführlich beschrieben.

Der Single-Link Ansatz kann besonders bei großen Datenmengen zu „kettenförmigen“ Clustern führen, da nur wenige Objekte zwischen zwei Clustern erforderlich sind, um diese beiden zu vereinigen. Single-Link Cluster weisen daher häufig eine starke Streuung und eine langgezogene Struktur auf [JD88] (vgl. auch Abbildung 3.3).

Complete-Link

Um den Ketten-Effekt des Single-Link Verfahrens zu vermeiden, wählt der Complete-Link Ansatz ein anderes Vereinigungskriterium. Bei diesem Ansatz wird der Durchmesser der Cluster minimiert. Die Complete-Link Methode oder auch Furthest-Neighbor Methode wurde 1948 von Sorensen entwickelt [Sor48]. Der Abstand dist_{CL} zwischen zwei Clustern X und Y wird definiert als der maximale Abstand über alle Elementpaare aus beiden Clustern:

$$\mathrm{dist}_{CL}(X, Y) = \max_{x \in X,\ y \in Y} \mathrm{dist}(x, y)$$

Graphentheoretisch werden Complete-Link Cluster auch als Cliquen oder maximal vollständige Teilgraphen bezeichnet [JD88].

Nach derzeitigem Kenntnisstand existiert kein Complete-Link Algorithmus, der effizienter als der generelle HAC-Algorithmus bei gleichbleibender Effektivität ist [FBY92]. Wegen der hohen Laufzeitkomplexität ist dieser aber für große Datenmengen eher ungeeignet. Der effizienteste Complete-Link Algorithmus ist CLINK von Defays [Def77]. Er ist dem SLINK Algorithmus von Sibson sehr ähnlich und benötigt ebenfalls eine Speicherplatzkomplexität von $O(n)$ sowie eine Laufzeitkomplexität von $O(n^2)$. Allerdings hat er in Bezug auf seine Effektivität einige unbefriedigende Ergebnisse geliefert [FBY92].

Der Complete-Link Ansatz generiert durch die Minimierung des Clusterdurchmessers tendenziell kleine, stark voneinander abgegrenzte Cluster mit ähnlichem Durchmesser. Die Clusterelemente weisen zwar einerseits einen geringen Abstand zueinander auf, andererseits werden dadurch aber viele Cluster mit eher wenig Elementen erzeugt [JD88] (vgl. auch Abbildung 3.3).

Abbildung 3.3 verdeutlicht den Unterschied zwischen Single-Link und Complete-Link: es existieren zwei, durch eine „Brücke“ von Ausreißern miteinander verbundene Cluster. Der Single-Link Algorithmus erzeugt Cluster, wie sie in

Abbildung 3.3 a) dargestellt sind, das Ergebnis des Complete-Link Algorithmus wird in Abbildung 3.3 b) aufgeführt (vgl. [JMF99]).

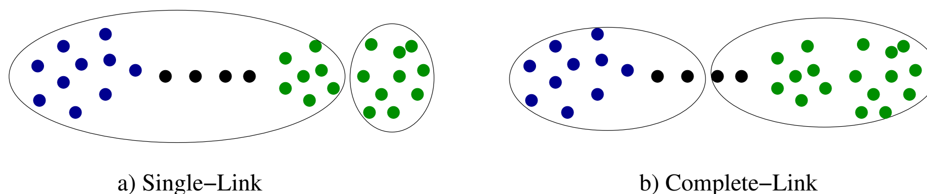


Abbildung 3.3: Single-Link bzw. Complete-Link Cluster einer Datenmenge mit zwei Klassen, die durch eine Kette von Objekten verbunden sind.

Average-Link

Die Average-Link Methode wurde erstmalig von Sokal und Michener 1958 erwähnt und ist auch unter dem Namen UPGMA (unweighted pair-group method using the average approach) bekannt. Beim Average-Link Ansatz ergibt sich der Abstand dist_{AVG} zwischen zwei Clustern X und Y als der durchschnittliche Abstand über alle Elementpaare beider Cluster:

$$\text{dist}_{AVG}(X,Y) = \frac{1}{|X| \cdot |Y|} \sum_{x \in X,\ y \in Y} \text{dist}(x,y)$$

Für den Average-Link Ansatz ist kein effizienter Algorithmus bekannt, der eine Speicherplatzkomplexität von $O(n)$ und eine Laufzeitkomplexität von $O(n^2)$ besitzt [Wil88]. Dies resultiert aus der Tatsache, dass die Distanz zwischen zwei Clustern aus allen paarweisen Distanzen der Clusterelemente berechnet wird und nicht wie bei Single-Link oder Complete-Link von einer einzigen Distanz zwischen zwei Elementen bestimmt wird. Voorhees beschreibt in [Voo86] einen effizienten Algorithmus für den Average-Link Ansatz, der unter bestimmten Voraussetzungen angewandt werden kann.

Die Clusterergebnisse des Average-Link Verfahrens liegen in ihrer Struktur zwischen den langgezogenen Ketten des Single-Links Clustering und den stark abgegrenzten Clustern des Complete Link Clustering [ESM01].

3.5 Die Lance-Williams Rekursionsformel

Die Distanz zwischen zwei Clustern kann rekursiv mit Hilfe der sogenannten Lance-Williams Rekursionsformel berechnet werden:

Definition 3.5 (Lance-Williams Rekursionsformel). [LW67] Die Distanz zwischen einem neuen Cluster, bestehend aus Cluster i und j und dem Cluster k ist wie folgt definiert:

$$\text{dist}(i+j,k) = \alpha_i \,\text{dist}(i,k) + \alpha_j \,\text{dist}(j,k) + \beta \,\text{dist}(i,j) + \gamma \;|\,\text{dist}(i,k) - \text{dist}(j,k)|$$

Diese Formel ermöglicht eine flexible Berechnung für verschiedene Clusteringverfahren. Die Parameter α, β und γ für die vorgestellten Verfahren sind in Abbildung 3.4 angegeben. Dabei bezeichnet $|x|$ ist die Anzahl der Objekte in Cluster x.

	α_i	α_j	β	γ
Single-Link	$\frac{1}{2}$	$\frac{1}{2}$	0	-$\frac{1}{2}$
Complete-Link	$\frac{1}{2}$	$\frac{1}{2}$	0	$\frac{1}{2}$
Average-Link	$\frac{\|i\|}{\|i\|+\|j\|}$	$\frac{\|j\|}{\|i\|+\|j\|}$	0	0

Abbildung 3.4: Parameter in der Lance-Williams Rekursionsformel

Kapitel 4

Inkrementelles Single-Link

Grundlage des im nachfolgenden vorgestellten Algorithmus ist das Single-Link-Verfahren SLINK, das 1973 von R. Sibson vorgestellt wurde [Sib73]. SLINK zeichnet sich durch die kompakte Darstellung der hierarchischen Baumstruktur in Form einer sogenannten Pointer-Repräsentation aus, die im ersten Abschnitt dieses Kapitels erläutert wird. Sibson entwickelte eine Methode, die Änderungen, die durch das Einfügen eines weiteren Objekts in ein bestehendes Dendrogramm erforderlich werden, zu bestimmen und effizient umzusetzen. SLINK ermöglicht es, ein n-elementiges Dendrogramm ausgehend von einem Startobjekt rekursiv in n Schritten zu erzeugen. Aufbauend auf diesem Verfahren wird im zweiten Teil ein inkrementeller Algorithmus vorgestellt, der es ermöglicht, in ein bestehendes Single-Link-Dendrogramm Objekte einzufügen bzw. aus einem bestehendem Dendrogramm Objekte zu löschen. Im letzten Abschnitt dieses Kapitels erfolgt eine abschließende Bewertung des inkrementellen Single-Link-Algorithmus.

4.1 Die Pointer-Repräsentation

Definition 4.1 (Ordnung auf Objekten). Gegeben sei die Objektmenge $\mathcal{O} = \{o_1, \dots o_n\}$. Jedes Objekte in $\mathcal{O}$ trage als zusätzliches Attribut die entsprechende Objekt-ID von $1, \dots, n$. Dann beschreibt die Relation $\prec$ mit

$$\forall o_i, o_j \in \mathcal{O}: \; o_i \prec o_j \Leftrightarrow o_i.id < o_j.id$$

eine strenge Ordnung auf $\mathcal{O}$.

Definition 4.2 (Pointer-Repräsentation). Gegeben sei die Objektmenge $\mathcal{O} = \{o_1, \dots o_n\}$. Die beiden Funktionen $\pi : \mathcal{O} \to \mathcal{O}$ und $\lambda : \mathcal{O} \to [0, \infty[$ definieren eine sogenannte Pointer-Repräsentation (π, λ) eines Dendrogramms, falls folgende Bedingungen erfüllt sind:

- $\pi(o_n) = o_n$
- $\pi(o_i) \succ o_i \qquad$ für $o_i \prec o_n$
- $\lambda(o_n) = \infty$
- $\lambda(\pi(o_i)) > \lambda(o_i) \qquad$ für $o_i \prec o_n$

Die folgenden Überlegungen zeigen, wie aus einem Dendrogramm die korrespondierende Pointer-Repräsentation und umgekehrt ermittelt werden kann.

4.1.1 Überführung eines Dendrogramms in die Pointer-Repräsentation

Sei die Objektmenge $\mathcal{O} = \{o_1, \ldots o_n\}$ gegeben und beschreibe $c : [0, \infty[\to E(\mathcal{O})$ ein Dendrogramm gemäß Definition 3.4. Die Funktionen π und λ der zugehörigen Pointer-Repräsentation (π, λ) sind dann wie folgt definiert:

$$\pi(o_i) = \begin{cases} \max\{o_j : (o_i, o_j) \in c(\lambda(o_i))\} & \text{, für } o_i \prec o_n \\ o_n & \text{, für } o_i = o_n \end{cases}$$

$$\lambda(o_i) = \begin{cases} \inf\{h : \exists o_j \succ o_i : (o_i, o_j) \in c(h)\} & \text{, für } o_i \prec o_n \\ \infty & \text{, für } o_i = o_n \end{cases}$$

D.h. der Level $\lambda(o_i)$ ist der kleinste Level im Dendrogramm, an dem das Objekt o_i nicht mehr das letzte Objekt in seinem Cluster ist. Die Funktion $\pi(o_i)$ beschreibt dann das letzte Objekt im Cluster, mit dem o_i verbunden ist.

4.1.2 Überführung der Pointer-Repräsentation in ein Dendrogramm

Sei nun eine Pointer-Repräsentation durch die Funktionen π und λ gegeben. Sei $\sigma(o_i, h)$ das erste Element o_k in der Sequenz

$$o_i,\ \pi(o_i),\ \pi^2(o_i),\ \pi^3(o_i),\ \ldots,\ o_n$$

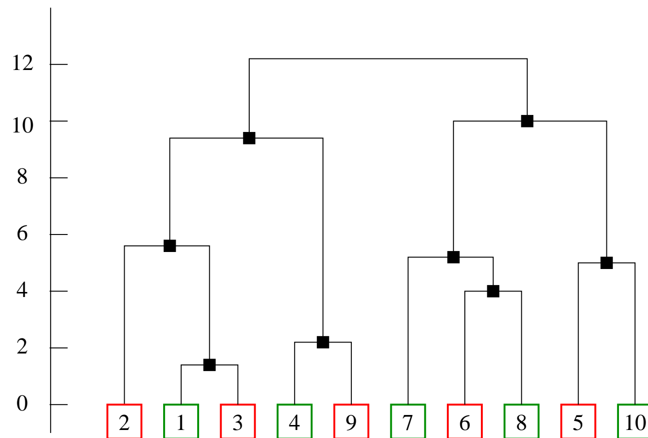
für das $\lambda(o_k) > h$ gilt.

Dann beschreibt die Funktion c mit

$$c(h) = \{(o_i, o_j) : \sigma(o_i, h) = \sigma(o_j, h)\}$$

das zur gegebenen Pointer-Repräsentation zugehörige Dendrogramm.

In nachfolgender Abbildung 4.1 ist ein Beispiel eines Dendrogramms und seiner korrespondierenden Pointer-Repräsentation dargestellt.



Objekt	1	2	3	4	5	6	7	8	9	10
π	3	3	9	9	10	8	8	10	10	10
λ	1.4	5.7	9.4	2.2	5.0	4.0	5.1	10.0	12.2	∞

Abbildung 4.1: Dendrogramm mit korrespondierender Pointer-Repräsentation

4.1.3 Überführung der Pointer-Repräsentation in eine Baumstruktur

Die Pointer-Repräsentation stellt zwar eine kompakte und effiziente Speicherstruktur für ein Dendrogramm dar, ist aber für den Anwender insbesondere bei großen Datenmengen nur schwer lesbar. Üblicherweise erfolgt die visuelle Darstellung eines Dendrogramms in Form einer Baums. In Listing 4.1 ist ein Pseudocode angegeben, um eine Pointer-Repräsentation (π, λ) in eine Baumstruktur zu überführen.

Der Aufbau der Baumstruktur erfolgt dabei bottom up: schrittweise werden jeweils die Blätter in einem neuen Knoten zusammengefasst, die die geringste Distanz zueinander besitzen. Das Array `leafs` hält dabei die Objekte aufsteigend nach ihren λ-Werten sortiert, d.h. das erste Objekt aus `leafs` ist jeweils das nächste noch nicht bearbeitete Blatt mit der kleinsten Distanz zu seinem nächsten Nachbarn. Dieses Objekt bzw. genauer der letzte Vorfahr dieses Objekts stellt den linken Kindknoten im neuen Vaterknoten dar, der rechte Kindknoten ergibt sich aus dem π-Wert des linken Knotens bzw. aus dem letzten Vorfahren dieses π-Werts. Ist der π-Wert des linken Knotens gleich diesem linken Knoten, also gilt $\pi(o) = o$, dann wurden alle Blätter bearbeitet und der zuletzt gebildete Vaterknoten entspricht somit der Wurzel des Dendrogramms.

```
convertToTree(Pi,Lambda)
  // Input  : Pointer-Repräsentation (π,λ)
  // Output: zugehörige Baumstruktur

  // hält Objekte aufsteigend nach Lambda sortiert
  List objects;
  TreeNode parent,left,right;

  while (left != right) do
    //Bestimme linkes Kind
    left := objects.first();
    objects.remove(left);

    //Bestimme rechtes Kind
    right := Pi[left];

    //Erzeuge neuen Vaterknoten
    parent := new TreeNode();
    parent.leftChild := left.lastAncestor();
    parent.rightChild := right.lastAncestor();
    parent.distance := Lambda[left];
    left.lastAncestor().parent := parent;
    right.lastAncestor().parent := parent;

  //end while

  Tree tree := new Tree();
  tree.root := parent;
  return tree;
```

Listing 4.1: Überführung der Pointer-Repräsentation in eine Baumstruktur

4.2 Einfügen neuer Objekte

Definition 4.3 (Rekursiver Aufbau der Pointer-Repräsentation). Sei c_n das Single-Link-Dendrogramm, das aus den Objekten der Objektmenge $\mathcal{O} = \{o_1, \ldots, o_n\}$ resultiert und definieren die Funktionen π_n und λ_n die zugehörige Pointer-Repräsentation (π_n, λ_n).

Für gegebenes n sei die Funktion $\mu_n : \mathcal{O} \to [0, \infty[$ wie folgt rekursiv über o_i definiert:

$$\mu_n(o_i) = \min\{\text{dist}(o_i, o_{n+1}), \min_{\pi_n(o_j)=o_i} \max\{\mu_n(o_j), \lambda_n(o_j)\}\}$$

Die Pointer-Repräsentation (π, λ) des Dendrogramms c_{n+1} für die Objektmenge $\mathcal{O} = \{o_1, \ldots, o_{n+1}\}$ wird dann wie folgt durch die beiden Funktionen π und λ definiert:

$$\pi(o_i) = \begin{cases} o_{n+1} & \text{, falls} \quad \mu_n(o_i) \leq \lambda_n(o_i) \quad \text{oder} \\ & \qquad\quad \mu_n(\pi(o_i)) \leq \lambda_n(o_i) \quad \text{oder} \\ & \qquad\quad o_i = o_{n+1} \\ \pi_n(o_i) & \text{, sonst} \end{cases}$$

$$\lambda(o_i) = \begin{cases} \infty & \text{, falls } o_i = o_{n+1} \\ \min\{\mu_n(o_i), \lambda_n(o_i)\} & \text{, für } o_i \prec o_n \end{cases}$$

Startet man mit den Funktionen π_1 und λ_1, wobei $\pi_1(o_1) = o_1$ und $\lambda_1(o_1) = \infty$, so ergeben sich nach $n - 1$ Schritten die Funktionen π_n und λ_n, die die Pointer-Repräsentation (π_n, λ_n) des Single-Link-Dendrogramms der Objektmenge $\mathcal{O} = o_1, \ldots, o_n$ definieren.

Das Einfügen eines neuen Objekts o_{n+1} in ein aus n Objekten bestehendes Dendrogramm entspricht somit dem $n + 1$-ten Rekursionsschritt. Aus der bestehenden Pointer-Repräsentation π_n, λ_n wird mit der in Listing 4.2 angegeben Prozedur `insert` die neue Pointer-Repräsentation π_{n+1}, λ_{n+1} berechnet. Die Prozedur `insert` entspricht dem in Definition 4.3 angegebenen Algorithmus in nicht rekursiver Form und ist analog zum Verfahren SLINK von R. Sibson in [Sib73] aufgebaut.

Als Datenstrukturen werden drei Arrays variabler Länge gewählt, bezeichnet mit `Pi`, `Lambda` und `M`, wobei `Pi` und `Lambda` in ihren ersten n Positionen die Werte von π_n und λ_n halten und `M` als Hilfsarray fungiert. Im ersten Schritt werden die initialen Werte für $\pi(o_{n+1}) = o_{n+1}$ und $\lambda(o_{n+1}) = \infty$ gesetzt. Danach werden die paarweisen Distanzen der bereits eingefügten Objekte o_i zu dem neuen Objekt o_{n+1} berechnet und im Hilfsarray `M` zwischengespeichert. Im darauf folgenden Schritt 3 werden die Werte für π und λ wie folgt aktualisiert: ist das bisherige Level $\lambda(o_i)$ größer gleich dem Distanzwert in $\mathtt{M}(o_i)$, wird o_{n+1} als letztes Objekt in diesen Cluster eingefügt, d.h. $\pi(o_i) = o_{n+1}$ und $\lambda_(o_i) = \mathtt{M}(o_i)$. Zuvor muss jedoch für das bisherige letzte Objekt im Cluster $\pi(o_i)$ der Distanzwert auf das Minimum aus bisherigem Wert $\mathtt{M}(\pi(o_i))$ und bisherigem Level $\lambda(o_i)$ gesetzt werden. Im anderen Fall, also $\lambda(o_i)$ kleiner dem Distanzwert $\mathtt{M}(o_i)$, ändert sich das letzte Objekt im Cluster $\pi(o_i)$ nicht und es muss lediglich der minimale Distanzwert aus $\mathtt{M}(\pi(o_i))$ und $\mathtt{M}(o_i)$ gesetzt werden. Abschließend werden in Schritt 4 aufgrund der Monotonieeigenschaft $\lambda(\pi(o_i)) > \lambda(o_i)$ der Pointer-Repräsentation die letzten Elemente im Cluster $\pi(o_i)$ gegebenenfalls aktualisiert.

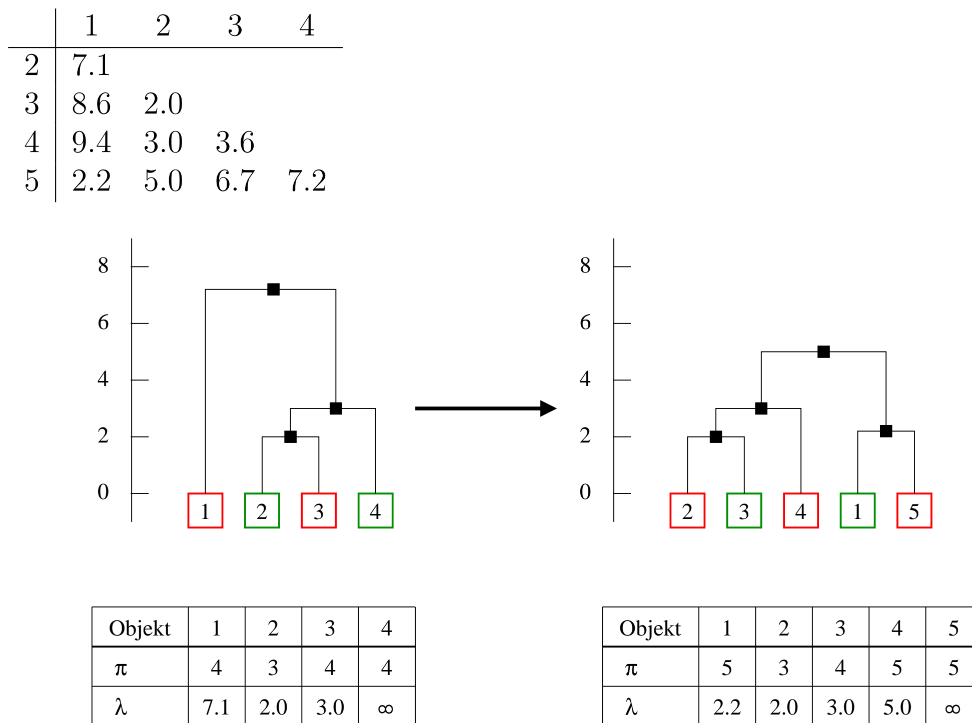
```
insert(Object o_{n+1})
  // Input : einzufügendes Objekt o_{n+1}

  // Initialisiere π(o_{n+1}) und λ(o_{n+1})
  Pi(o_{n+1}) := o_{n+1};
  Lambda(o_{n+1}) := MAXVALUE;

  // Berechne die paarweisen Distanzen
  // der Objekte o_i zum neuen Objekt o_{n+1}
  for i=1 to n do
    M(o_i) := dist(o_i, o_{n+1});

  // Berechne die Werte für π und λ
  for i=1 to n do
    if Lambda(o_i) >= M(o_i) then
      M(Pi(o_i)) := min( M(Pi(o_i)), Lambda(o_i) );
      Lambda(o_i) := M(o_i);
      Pi(o_i) := o_{n+1};

    else
      M(Pi(o_i)) := min( M(Pi(o_i)), M(o_i) );

  // Aktualisiere die Cluster, falls nötig
  for i=1 to n do
    if Lambda(o_i) >= Lambda(Pi(o_i)) then
      Pi(o_i) := o_{n+1};
```

Listing 4.2: Inkrementelles Single-Link: Einfügen eines neuen Objekts

Die Speicherplatzbedarf für ein Dendrogramm aus n Objekten beträgt $O(n)$ (genau: $O(3n)$) für die drei Arrays `Pi`, `Lambda` und `M` der Länge n, wobei `M` nur zur Laufzeit im Hauptspeicher gehalten werden muss. Darüberhinaus ist es nicht erforderlich, die komplette $n \times n$-Distanzmatrix im Hauptspeicher zu halten, da bei der Initialisierung von `M` die Distanzwerte jeweils nur zeilenweise benötigt werden.

Obwohl die Pointer-Repräsentation eines Dendrogramms aus Anwendersicht nicht sehr übersichtlich ist, hat sie den Vorteil, dass sie sehr effizient berechnet werden kann. Der Aufbau der hierarchischen Struktur eines Dendrogramms aus n Objekten besitzt eine Laufzeitkomplexität von $O(n^2)$. Das Einfügen von m Objekten benötigt m Rekursionsschritte und besitzt somit eine Laufzeitkomplexität von $O(m(n + m))$.

Nachfolgende Abbildung 4.2 illustriert anhand eines Beispiels das Einfügen eines neuen Objekts mit der ID 5 in ein aus 4 Objekten bestehendes Single-Link-Dendrogramm. Die Distanzmatrix ist dabei wie folgt gegeben:



	1	2	3	4
2	7.1			
3	8.6	2.0		
4	9.4	3.0	3.6	
5	2.2	5.0	6.7	7.2

Objekt	1	2	3	4
π	4	3	4	4
λ	7.1	2.0	3.0	∞

Objekt	1	2	3	4	5
π	5	3	4	5	5
λ	2.2	2.0	3.0	5.0	∞

Abbildung 4.2: Einfügen der neuen Objekt-ID 5

4.3 Löschen bestehender Objekte

Das Löschen bestehender Objekte gestaltet sich komplexer als das Einfügen neuer Objekte. Unter Umständen bewirkt das Löschen eines Objektes x (ebenso wie das Einfügen) eine vollständige Reorganisation des Dendrogramms. Beim Einfügen kann aber aus der Pointer-Repräsentation π_n, λ_n rekursiv die neue Pointer-Repräsentation π_{n+1}, λ_{n+1} ermittelt werden. Der Rückwärtsschritt von π_n, λ_n nach π_{n-1}, λ_{n-1} beim Löschen eines Objekts ist aber bedingt durch den Rekursionsansatz nicht möglich.

Zur Vermeidung einer kompletten Neuberechnung des Dendrogramms werden zunächst die Cluster gesucht, deren Level kleiner ist als eine kritische Distanz $\lambda_{crit}(x)$. Für diese Cluster gilt, dass sie durch das Löschen von x nicht verändert werden. Im nachfolgenden Schritt wird ausgehend von diesen Clustern mit SLINK sukzessive das neue Dendrogramm aufgebaut. Der Pseudocode für das Löschen eines Objekts aus einem Single-Link-Dendrogramm ist nachfolgend in den Listings 4.3 und 4.4 angegeben.

Aus dem Single-Link-Ansatz folgt, dass die kritische Distanz, d.h. der Level $\lambda_{crit}(x)$, bis zu dem die gebildeten Cluster in jedem Fall gültig sind, der Level

des Großvater des zu löschenden Objekts x ist. Ist x direkt unter der Wurzel angeordnet, so ist $\lambda_{crit}(x)$ der Level der Wurzel des Dendrogramms. Erfolgte bereits eine Transformation der Pointer-Repräsentation in eine Baumstruktur gemäß dem in Listing 4.1 angegebenen Pseudocode, so kann der Großvater von x und damit auch $\lambda_{crit}(x)$ bequem aus der Baumstruktur ermittelt werden. Ansonsten kann der kritische Level $\lambda_{crit}(x)$ wie nachfolgend erläutert aus der Pointer-Repräsentation berechnet.

Enthalte die Menge $\mathcal{Y} = \{y_1, \dots y_n\}$ alle Objekte o aus der Objektmenge $\mathcal{O}$, für die gilt $\pi(o) = x$. Sei $\lambda_{y_j} = \lambda(y_j)$ und gelte ohne Beschränkung der Allgemeinheit $\lambda_{y_1} \leq \lambda_{y_2} \leq \dots \leq \lambda_{y_n}$. Weiterhin enthalte die Menge $\mathcal{Z} = \{z_1, \dots z_m\}$ alle Objekte $o \in \mathcal{O}$, für die gilt $\pi(o) = \pi(x)$. Bezeichne $\lambda_{z_j} = \lambda(z_j)$ und gelte wiederum ohne Beschränkung der Allgemeinheit $\lambda_{z_1} \leq \dots \leq \lambda_{z_m}$. Der kritische Level $\lambda_{crit}(x)$ ergibt sich gemäß folgender Fallunterscheidung:

$$\lambda_{crit}(x) = \begin{cases} \lambda_{x_2} & \text{, falls } n \geq 2 \\ \max\{\lambda(x), \lambda_{x_1}\} & \text{, falls } n = 1 \\ \lambda_{z_{k+1}} & \text{, falls } n = 0 \text{ und } m \geq 2 \text{, wobei } z_k = x \\ \lambda(\pi(x)) & \text{, sonst} \end{cases}$$

Die Elemente der gültigen Cluster ergeben sich aus den Blättern der Teilbäume des Dendrogramms, deren Höhe $\lambda_{crit}(x)$ nicht überschreitet. Die gültigen Cluster können auch aus der Pointer-Repräsentation wie folgt berechnet werden: für jedes Objekt $o_i \in O \backslash \{x\}$ wird der Wert von $\sigma(o_i, \lambda_{crit}(x))$ ermittelt, wobei $\sigma(o_i, h)$ das erste Element o_k in der Sequenz o_i, $\pi(o_i)$, $\pi^2(o_i)$, $\pi^3(o_i)$, $\dots$, o_n ist, für das $\lambda(o_k) \geq h$ gilt. Alle Objekte o_i, die denselben Wert für $\sigma(o_i, \lambda_{crit}(x))$ besitzen, liegen zusammen in einem gültigen Cluster.

Im nächsten Schritt wird ausgehend von den gültigen Clustern als Input mit Hilfe des SLINK-Algorithmus eine vereinfachte Pointer-Repräsentation (π_{vc}, λ_{vc}) des Dendrogramms erzeugt. Die IDs der Cluster, die Ordnung auf den Clustern gemäß Definition 4.1 definieren, werden dabei aufsteigend nach dem jeweils letzten Element im Cluster vergeben. Abschließend wird aus dieser vereinfachten Pointer-Repräsentation (π_{vc}, λ_{vc}) die neue Pointer-Repräsentation $(\pi_{n-1}, \lambda_{n-1})$ des Dendrogramms, das durch das Löschen von x entstanden ist, berechnet. Hierzu werden statt der gültigen Cluster wieder die ursprünglichen

Objekte betrachtet und es gilt für $o_i \in O \backslash \{x\}$:

$$\pi_{n-1}(o_i) = \begin{cases} \pi_n(o_i) & \text{, falls } \lambda_n(o_i) < \lambda_{crit}(x) \\ \text{last}(\pi_{vc}(\text{Cluster}(o_i))) & \text{, sonst} \end{cases}$$

$$\lambda_{n-1}(o_i) = \begin{cases} \lambda_n(o_i) & \text{, falls } \lambda_n(o_i) < \lambda_{crit}(x) \\ \lambda_{vc}(\text{Cluster}(o_i)) & \text{, sonst} \end{cases}$$

wobei Cluster(o_i) den gültigen Cluster von o_i und last(C) das letzte Element des Clusters C bezeichnet.

```
delete(Object o)
  // Input : zu löschendes Objekt o

  // Berechne  λcrit(o)
  TreeNode node := tree.getLeaf(o);
  TreeNode grandfather := node.parent.parent;
  lambda_crit := grandfather.height;

  // Berechne die gültigen Cluster
  // und wende SLINK darauf an
  SLink slink := new SLink;
  Cluster[] clusters := getValidClusters(lambda_crit);
  for i=1 to n do
    slink.insert(clusters[i]);

  // Berechne die neue Pointer-Repräsentation (π, λ)
  Pi_vc := slink.pi;
  Lambda_vc := slink.Lambda;
  for i=1 to n do
    if o_i=o then
      Pi.remove(o);
      Lambda.remove(o);

    else
      if Lambda(o_i) >= lambda_crit or Pi(o_i)=o then
        Cluster c := cluster(o_i);
        Pi(o_i) := last(Pi_vc(c));
        Lambda(o_i) := Lambda_vc(c);
```

Listing 4.3: Inkrementelles Single-Link: Löschen eines Objekts

Anhand eines Beispiels soll das Löschen eines Objekts verdeutlicht werden. Gegeben ist die folgende Distanzmatrix, das zu löschende Objekt trägt die ID 2. Abbildung 4.3 zeigt das Dendrogramm vor dem Löschen des Objekts o_2.

```
getValidClusters(double lambda_crit)
  // Input : kritische Distanz λcrit
  // Output: gültige Cluster zur Distanz λcrit

  Cluster[] result;
  DepthFirstEnumeration dfs :=
      tree.DepthFirstEnumeration();

  while (dfs.hasNext()) do
    TreeNode node := dfs.next();
    if node.height < lambda_crit and
       node.parent.height >= lambda_crit then
      result.add(new Cluster(node.leafs());

  return result;
```

Listing 4.4: Inkrementelles Single-Link: Berechnung der gültigen Cluster

	1	2	3	4	5	6	7	8	9
2	7.1								
3	1.4	5.7							
4	18.6	11.7	17.2						
5	17.7	15.6	17.1	16.1					
6	25.0	25.0	24.8	26.5	10.4				
7	29.0	29.7	29.0	31.6	15.5	5.1			
8	21.9	22.8	21.9	26.0	10.0	4.0	7.1		
9	16.4	9.4	15.0	2.2	15.5	26.0	31.0	25.2	
10	13.0	12.2	12.5	16.3	5.0	12.8	17.5	15.0	10.8

Im ersten Schritt wird der kritische Level λ_{crit} ermittelt, bis zu dem die gebildeten Cluster gültig sind. Im Beispiel ergibt sich für die Höhe des Großvaters des Objekts o_2 der Wert $\lambda_{crit}(o_2) = 9.4$. Die gültigen Cluster werden aus den Teilbäumen des Dendrogramms ermittelt, deren Höhe $\lambda_{crit}(o_2)$ nicht überschreitet. Die IDs der Cluster werden dabei nach dem letzten Element im jeweiligen Cluster vergeben. Man erhält somit die gültigen Cluster $c_3 = \{o_1, o_3\}$, $c_8 = \{o_6, o_7, o_8\}$, $c_9 = \{o_4, o_9\}$ und $c_{10} = \{o_5, o_{10}\}$ (vgl. Abbildung 4.4).

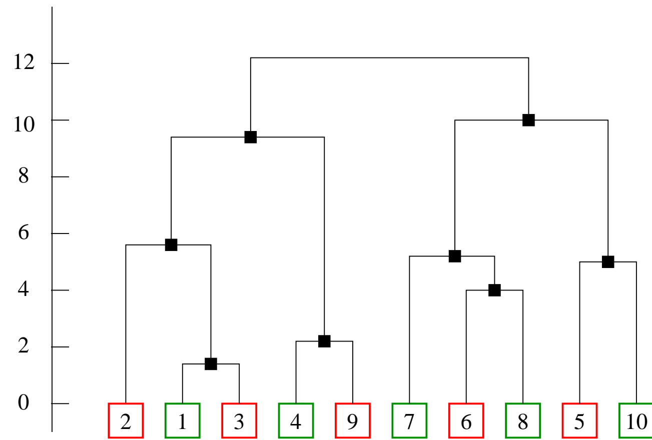


Abbildung 4.3: Dendrogramm vor dem Löschen der Objekt-ID 2

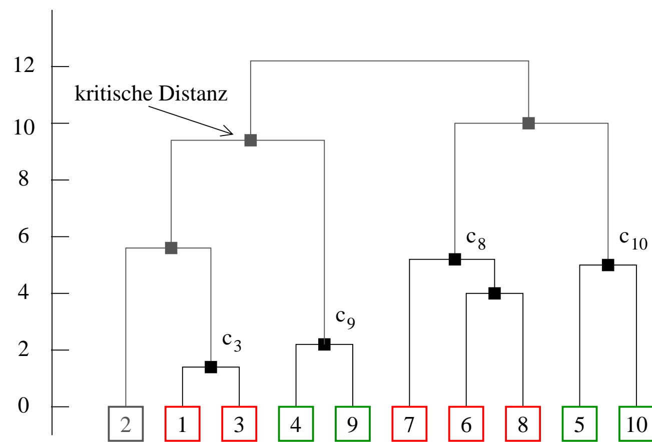


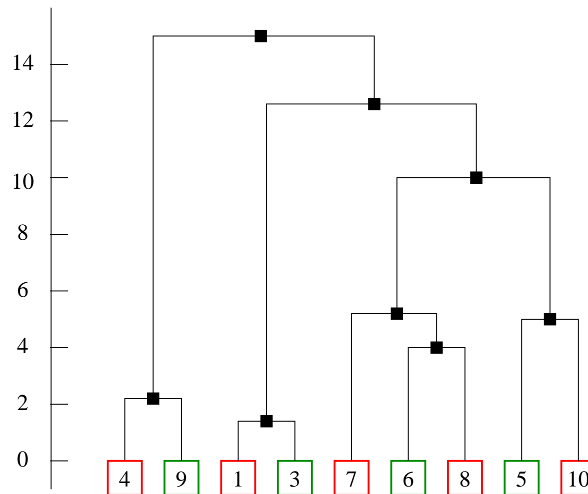
Abbildung 4.4: Ermitteln der gültigen Cluster

Im nächsten Schritt wird ausgehend von den gültigen Clustern als Input mit Hilfe des SLINK-Algorithmus ein neues vereinfachtes Dendrogramm erzeugt. Im Beispiel ergibt sich die folgende Distanzmatrix bzw. Pointer-Repräsentation π_{vc},λ_{vc} für die gültigen Cluster:

	c_3	c_8	c_9
c_8	21.9		
c_9	15.0	25.2	
c_{10}	12.5	10.0	15.0

	c_3	c_8	c_9	c_{10}
π_{vc}	c_{10}	c_{10}	c_{10}	c_{10}
λ_{vc}	12.5	10.0	15.0	∞

Abschließend wird in Schritt 4 aus dieser vereinfachten Pointer-Repräsentation das neue durch das Löschen von o_2 entstandene Dendrogramm berechnet. Hierzu werden statt der gültigen Cluster wieder die ursprünglichen Objekte betrachtet und die Pointer-Repräsentation gemäß obigem Algorithmus aktualisiert. Nach dem Löschen des Objekts o_2 ergibt sich somit das Dendrogramm aus Abbildung 4.5 :



Objekt	1	3	4	5	6	7	8	9	10
π	3	10	9	10	8	8	10	10	10
λ	1.4	12.5	2.2	5.0	4.0	5.1	10.0	15.0	∞

Abbildung 4.5: Resultierendes Dendrogramm nach dem Löschen der Objekts mit der ID 2

4.4 Bewertung

Das Einfügen eines neuen Objekts in ein bestehendes Dendrogramm entspricht einem Rekursionsschritt des SLINK-Algorithmus und besitzt eine lineare Laufzeit von $O(n)$. Das Löschen eines Objekts weist bedingt durch den Single-Link-Ansatz eine quadratische Laufzeitkomplexität von $O(n^2)$ auf. Im worst-case Fall ergeben sich gültige Cluster, die aus jeweils einem Objekt der Objektmenge bestehen. Die durchschnittliche Laufzeit des Löschvorgangs liegt dennoch unter der Laufzeit einer kompletten Neuberechnung des Dendrogramms. Allerdings sind der Performanz algorithmusspezifische Grenzen gesetzt. Ursache hierfür ist die aufwändige Neuberechnung der Distanz zwischen zwei Clustern, die sich aus dem Minimum der paarweisen Distanzen zwischen den Clusterelementen ergibt. Dem positiven Effekt, dass sich bei einem hohen kritischen Level λ_{crit} ein geringer Input zur Bestimmung der vereinfachten Pointer-Repräsentation ergibt, steht der Umstand entgegen, dass die gültigen Cluster entsprechend mehr Objekte enthalten und sich die Distanzberechnung zwischen den Cluster somit aufwändiger gestaltet.

Die in Kapitel 6 entwickelte inkrementelle Version des dichtebasierten hierarchischen Clusteringverfahren OPTICS stellt eine effiziente Alternative zu oben vorgestellten Verfahren dar. Wie in [SQL$^+$03] gezeigt wurde, kann ein Dendrogramm für geeignete Parameterwahl in ein von OPTICS erzeugtes Erreichbarkeitsdiagramm transformiert werden und umgekehrt. Somit repräsentiert die inkrementelle Version von OPTICS indirekt auch eine effiziente inkrementelle Version für Single Link.

Kapitel 5

Dichtebasiertes hierarchisches Clustering

Cluster können auch als Gebiete im d-dimensionalen Raum angesehen werden, in denen eine hohe Objektdichte vorherrscht. Diese dichten Gebiete sind getrennt durch Gebiete, in denen die Objekte weniger dicht beieinander liegen. Dichtebasierte Clusteringverfahren versuchen, solche dichten Gebiete im Raum zu identifizieren. Für dichtebasierte Cluster gilt, dass für jedes Objekt eines Clusters die lokale Punktdichte einen gegebenen Grenzwert überschreitet und die Objekte innerhalb eines Clusters räumlich zusammenhängend sind.

Im folgenden werden zunächst die Grundbegriffe des dichtebasierten Clusterings erläutert und der dichtebasierte Algorithmus DBSCAN [EKSX96] vorgestellt. Anschließend wird das Verfahren OPTICS [ABKS99] behandelt, das den dichtebasierten Ansatz erweitert und mit einer hierarchischen Datenrepräsentation vereinigt.

5.1 Cluster als dichteverbundene Mengen

Sei $\mathcal{O}$ eine Menge von Objekten, sei $p, q \in \mathcal{O}$ und sei $\varepsilon \in \mathbb{R}^+$, $MinPts \in \mathbb{N}^+$.

Definition 5.1 (ε-Umgebung eines Objekts). Die ε-Umgebung eines Objekts p, kurz $N_\varepsilon(p)$, ist die Menge aller Objekte q, deren Distanz zu p kleiner gleich einem gegebenen Wert ε ist:

$$N_\varepsilon(p) = \{q \in \mathcal{O} \mid \operatorname{dist}(p, q) \leq \varepsilon\}$$

Definition 5.2 (Kernobjekt). Ein Objekt p heißt Kernobjekt in $\mathcal{O}$, wenn in seiner ε-Umgebung mindestens $MinPts$ viele Objekte liegen (inkl. p selbst), d.h. wenn gilt:

$$|N_\varepsilon(p)| \;\geq\; MinPts$$

Definition 5.3 (Direkte Dichte-Erreichbarkeit). Ein Objekt p ist von einem Objekt q aus bzgl. ε und $MinPts$ direkt dichte-erreichbar in der Menge $\mathcal{O}$, falls gilt:

1. $p \in N_\varepsilon(q)$ (p liegt in der ε-Umgebung von q)
2. $|N_\varepsilon(q)| \geq MinPts$ (q ist ein Kernobjekt in $\mathcal{O}$)

Abbildung 5.1 illustriert die Begriffe Kernobjekt und direkte Dichte-Erreichbarkeit anhand eines zweidimensionalen Beispiels für $MinPts = 3$.

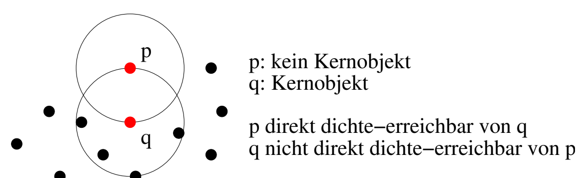


Abbildung 5.1: Kernobjekte und direkte Dichte-Erreichbarkeit für $MinPts = 3$

Definition 5.4 (Dichte-Erreichbarkeit). Ein Objekt p ist von einem Objekt q aus bzgl. ε und $MinPts$ dichte-erreichbar, kurz $Reach^{\mathcal{O}}_{\varepsilon,MinPts}(p, q)$ in der Menge $\mathcal{O}$, falls gilt:

1. Es existiert eine Folge von Objekten $p_1, \ldots, p_n$ in $\mathcal{O}$ mit $p_1 = q$ und $p_n = p$
2. Jedes p_{i+1} ist direkt dichte-erreichbar von p_i bzgl. ε und $MinPts$.

Ein Objekt p ist von einem Objekt q aus dichte-erreichbar, wenn es zwischen p und q eine Kette von direkt dichte-erreichbaren Objekten gibt. Alle Objekte außer eventuell p sind dabei Kernobjekte. Die Relation der Dichte-Erreichbarkeit ist die transitive Hülle der direkten Dichte-Erreichbarkeit und ist im allgemeinen nicht symmetrisch. Abbildung 5.2 veranschaulicht nochmal den Begriff der Dichte-Erreichbarkeit.

Definition 5.5 (Dichte-Verbundenheit). Ein Objekt p ist mit einem Objekt q bzgl. ε und $MinPts$ dichte-verbunden in der Menge $\mathcal{O}$, falls es ein Objekt $o \in \mathcal{O}$ gibt, so dass sowohl p als auch q von o aus bzgl. ε und $MinPts$ dichte-erreichbar sind.

Die Relation der Dichte-Verbundenheit ist eine symmetrische Relation. Die Dichte-Verbundenheit für zwei Punkte p und q ist in Abbildung 5.3 dargestellt.

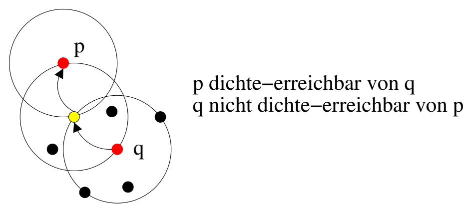


Abbildung 5.2: Dichte-Erreichbarkeit für $MinPts = 3$

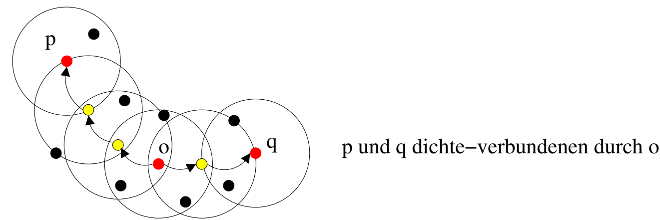


Abbildung 5.3: Dichte-Verbundenheit für $MinPts = 3$

Definition 5.6 (Dichtebasierter Cluster). Ein Cluster C bzgl. ε und $MinPts$ in $\mathcal{O}$ ist eine nicht-leere Teilmenge von $\mathcal{O}$, die folgende Bedingungen erfüllt:

1. Maximalität:
   $\forall p, q \in \mathcal{O}$: $p \in C$ und q dichte-erreichbar von p bzgl. ε und $MinPts \Rightarrow q \in C$

2. Verbundenheit:
   $\forall p, q \in \mathcal{O}$: p ist dichte-verbunden mit q bzgl. ε in $\mathcal{O}$

Anschaulich ist ein Cluster C eine Menge von Objekten, die alle miteinander dichte-verbunden sind. Alle Objekte, die von einem Kernpunkt des Clusters aus dichte-erreichbar sind, gehören auch schon zum Cluster. Objekte, die nicht zu einem Cluster gehören, heißen Rauschen. Das folgende Lemma formuliert eine wichtige Eigenschaft dichtebasierter Cluster.

Lemma 5.1. Sei C ein dichtebasierter Cluster bzgl. ε und $MinPts$ in $\mathcal{O}$ und sei $p \in C$ ein Kernobjekt. Dann gilt:

$$C = \{o \in \mathcal{O} \mid o \text{ dichte-erreichbar von } p \text{ bzgl. } \varepsilon \text{ und } MinPts\}$$

Lemma 5.1 besagt, dass ein Cluster C gefunden werden kann, indem ausgehend von einem beliebigen Kernobjekt von C alle dichte-erreichbaren Objekte aufgesammelt werden. Die Menge dieser Punkte entspricht dem kompletten

Cluster C. Lemma 5.1 bildet die Grundlage für den Algorithmus DBSCAN zur Bestimmung aller dichtebasierter Cluster in der Objektmenge $\mathcal{O}$, der im nachfolgenden Kapitel vorgestellt wird.

5.2 Algorithmus DBSCAN

Der Algorithmus DBSCAN [EKSX96] ermittelt ein dichtebasiertes Clustering gemäß den Definitionen aus Kapitel 5.1. Das Verfahren basiert auf Lemma 5.1, das besagt, dass ein Cluster äquivalent ist zur Menge all derjenigen Objekte aus der Datenmenge $\mathcal{O}$, die von einem beliebigen Kernobjekt des Clusters aus dichte-erreichbar sind.

Zu Beginn sind alle Objekte unklassifiziert. In der Prozedur `DBSCAN`, die im Pseudocode in Listing 5.1 angegeben ist, wird die Datenmenge schrittweise bearbeitet. Ausgehend von jedem noch unklassifizierten Objekt wird dabei versucht, mittels der Prozedur `expandCluster` aus Listing 5.2, einen kompletten Cluster zu finden. Hierbei wird zunächst geprüft, ob das aktuelle Objekt o ein Kernobjekt ist. Falls dies nicht der Fall ist, wird dieses Objekt (vorläufig) dem Rauschen zugeordnet und der Wert `false` zurückgegeben. Im anderen Fall, d.h. wenn es sich bei dem aktuellen Objekt o um ein Kernobjekt handelt, kann gemäß Lemma 5.1 ein neuer Cluster gefunden werden, indem alle von o aus dichte-erreichbaren Objekte in der Datenbank gesucht werden. Die Objekte der ε-Umgebung von o gehören sicher zu dem Cluster, da sie gemäß Definition 5.3 direkt dichte-erreichbar sind. Die von diesen Objekten wiederum direkt dichte-erreichbaren Objekte gehören ebenfalls zum Cluster, da sie transitiv dichte-erreichbar sind. Mit Hilfe der Prozedur `expandCluster` wird somit durch iterative Berechnung der direkten Dichte-Erreichbarkeit der gesamte Cluster zum aktuellen Kernobjekt o gefunden. Die Laufzeitkomplexität von DBSCAN beträgt $O(n\cdot$ Aufwand zur Bestimmung einer ε-Nachbarschaft).

```
DBSCAN(Object[] DB, Real epsilon, Integer minPts)
  // Input: Objektmenge DB, Parameter ε und MinPts
  // Zu Beginn sind alle Objekte unklassifiziert, d.h.
  // o.clID = UNCLASSIFIED für alle o ∈ DB
  Integer clusterID := nextID(NOISE);

  for i=1 to DB.size do
    Object o := DB.get(i);
    if o.clID = UNCLASSIFIED then
      if expandCluster(DB, o, clusterID, epsilon, minPts) then
        clusterID := nextID(clusterID);
```

Listing 5.1: Algorithmus DBSCAN

```
expandCluster(Object o, Integer clusterID,
              Real epsilon, Integer minPts)
  // Input: Objekt o des aktuellen Clusters,
  //        ID des aktuellen Clusters,
  //        Parameter ε und MinPts
  //
  // Output: true, falls ein Cluster gefunden
  //         wurde; false, sonst

  SetOfObjects seeds := Nε(o);

  // o ist kein Kernobjekt
  if seeds.size < minPts then
    o.clID := NOISE;
    return false;

  // o ist Kernobjekt
  for each o in seeds do
    o.clID := clusterID;

  seeds.remove(o);

  while seeds != ∅ do
    Object current := seeds.first();
    seeds.remove(current);
    SetOfObjects neighbors := Nε(current);

    // current ist Kernobjekt
    if neighbors.size >= minPts) then
      for i=1 to neighbors.size do
        Object p := neighbors.get(i);
        if p.clID = UNCLASSIFIED then
          seeds.append(p);
          p.clID := clusterID;

        if p.clID = NOISE then
          p.clID := clusterID;
  // endwhile

  return true;
```

Listing 5.2: DBSCAN: Prozedur `expandCluster`

5.3 Dichtebasierte Clusterordnung

Die Einführung einer dichtebasierten Clusterordnung basiert auf folgender Beobachtung: für einen konstanten $MinPts$-Wert sind dichtebasierte Cluster bzgl. einer höheren Dichte, d.h. bzgl. eines kleineren ε-Werts vollständig in dichte-basierten Clustern bzgl. einer niedrigeren Dichte, d.h. bzgl. eines höheren ε-Werts enthalten. Abbildung 5.4 illustriert diesen Zusammenhang. Aufgrund dieser Beziehung können dichtere Cluster, die in weniger dichten Clustern enthalten sind, in einem einzigen DBSCAN-ähnlichem Durchlauf gleichzeitig zusammen mit den weniger dichten Clustern gefunden werden. Die einzige Voraussetzung ist die Einhaltung einer bestimmten Reihenfolge während des Durchlaufs: es muss immer dasjenige Objekt aus der Menge `seeds` in der Prozedur `expandCluster` gewählt werden, das bzgl. des kleinsten ε-Wertes dichte-erreichbar ist von einem bereits bearbeiteten Objekt.

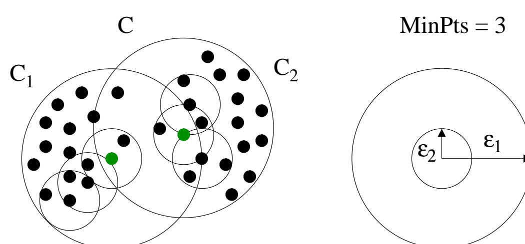


Abbildung 5.4: Ineinander enthaltene, unterschiedlich dichte Cluster

Um den Algorithmus zu formulieren, werden die Begriffe der k-Nächsten-Nachbarn-Distanz, Kerndistanz und Erreichbarkeitsdistanz eines Objekts benötigt, die im folgenden definiert werden. Dabei sei $\mathcal{O}$ eine Menge von Objekten, seien die Objekte $p, q \in \mathcal{O}$ und sei $\varepsilon \in \mathbb{R}^+$, $MinPts \in \mathbb{N}^+$. Die Distanz ε steht dabei für die größte Distanz, bis zu der man Umgebungen von Objekten betrachtet, $MinPts$ bezeichnet analog zum dichtebasierten Clustering die minimale Anzahl von Objekten, die in der Umgebung eines Objekts liegen müssen, damit dieses Objekt als Kernobjekt angesehen werden kann.

Definition 5.7 (k-Nächste-Nachbarn-Distanz). Sei $k \in \mathbb{N}^+$ eine natürliche Zahl mit $k \geq 1$ und $\mathcal{O}$ eine Objektmenge mit mindestens k Objekten. Dann ist die k-Nächste-Nachbarn-Distanz eines Objekts o definiert als $\text{nnDist}(o, k) = \text{dist}(o, p)$ für ein Objekt $p \in \mathcal{O}$, so dass die folgenden Bedingungen erfüllt sind:

- $\text{dist}(o, q) \leq \text{dist}(o, p)$ für mindestens k Objekte $q \in \mathcal{O}$
- $\text{dist}(o, q) < \text{dist}(o, p)$ für höchstens $k - 1$ Objekte $q \in \mathcal{O}$

Die k-Nächste-Nachbarn-Distanz $\text{nnDist}(o, k)$ eines Objekts o ist die Distanz von o zu seinem k-ten Nachbarn, wobei das Objekt o selbst mitgezählt wird, falls es in $\mathcal{O}$ enthalten ist. In Abbildung 5.5 ist die k-Nächste-Nachbarn-Distanz für ein Objekt o und Parameter $k = 4$ grafisch veranschaulicht ($o \notin \mathcal{O}$).

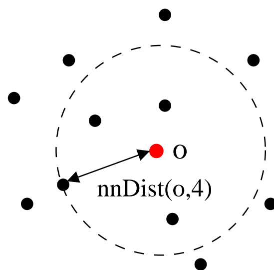


Abbildung 5.5: k-Nächste-Nachbarn-Distanz $\text{nnDist}(o, k)$ für $k = 4$

Definition 5.8 (Kerndistanz eines Objekts). Sei $\text{nnDist}(p, MinPts)$ die Distanz von p zum $MinPts$-Nächsten-Nachbarn gemäß Definition 5.7. Dann ist die Kerndistanz eines Objekts p bzgl. ε und $MinPts$, kurz $\text{coreDist}_{\varepsilon,MinPts}(p)$ wie folgt definiert:

$$\text{coreDist}_{\varepsilon,MinPts}(p) = \begin{cases} \infty & \text{, falls } |N_\varepsilon(p)| < MinPts \\ \text{nnDist}(p, MinPts) & \text{, sonst} \end{cases}$$

Die Kerndistanz eines Objekts p ist die kleinste Distanz $\varepsilon' < \varepsilon$, bei der p ein Kernobjekt ist. Ist p selbst bei der Distanz ε kein Kernobjekt, so ist die Kerndistanz undefiniert bzw. ∞.

Definition 5.9 (Erreichbarkeitsdistanz eines Objekts). Die Erreichbarkeitsdistanz eines Objekts p bzgl. ε und $MinPts$ relativ zu einem Objekt q, kurz $\text{reachDist}_{\varepsilon,MinPts}(p, q)$ ist wie folgt definiert:

$$\text{reachDist}_{\varepsilon,MinPts}(p, q) = \max\{\text{coreDist}_{\varepsilon,MinPts}(q),\ \text{dist}(p, q)\}$$

Die Erreichbarkeitsdistanz eines Objekts p relativ zu einem Objekt q ist die kleinste Distanz, bei der p von q aus direkt dichte-erreichbar ist, sofern q ein Kernobjekt ist.

Abbildung 5.6 veranschaulicht die Begriffe Kerndistanz und Erreichbarkeitsdistanz.

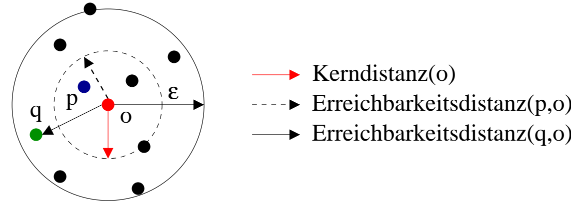


Abbildung 5.6: Kerndistanz und Erreichbarkeitsdistanz für $MinPts = 4$

Das Grundprinzip einer hierarchischen Clusterordnung besteht darin, die Objekte einer Objektmenge derart linear anzuordnen, dass Objekte eines Clusters in der Ordnung aufeinander folgen, d.h. dass Objekte eines Clusters in zusammenhängenden Intervallen in der Clusterordnung erscheinen. Formal kann eine Clusterordnung wie folgt rekursiv definiert werden.

Definition 5.10 (Clusterordnung). Sei CO eine vollständig geordnete Permutation der Objekte von $\mathcal{O}$ und besitze jedes $o \in \mathcal{O}$ die zusätzlichen Attribute $o.pos$, $o.core$ und $o.reach$. Weiterhin bezeichne CO_n die ersten n Elemente von CO. Die Permutation CO definiert eine Clusterordnung bzgl. ε und $MinPts$ auf $\mathcal{O}$, falls das i-te Element o von CO die folgenden Bedingungen erfüllt:

(1) $\text{reachDist}_{\varepsilon,MinPts}(o,p) =$
$\min\{\text{reachDist}_{\varepsilon,MinPts}(q,p) \mid q \in \mathcal{O} \setminus CO_{i-1},\ p \in CO_{i-1}\}$,
wobei $\min\{\emptyset\} = \infty$

(2) $o.pos = i$

(3) $o.core = \text{coreDist}_{\varepsilon,MinPts}(o)$

(4) $o.reach = \min\{\text{reachDist}_{\varepsilon,MinPts}(o,p) \mid p.pos < o.pos\}$,
wobei $\min\{\emptyset\} = \infty$

Bedingung 1 fordert, dass das nächste Objekt in der Clusterordnung CO immer dasjenige Objekt o ist, das aus allen Objekten, die noch nicht in CO enthalten sind eine minimale Erreichbarkeitsdistanz relativ zu einem bereits in der Clusterordnung enthaltenen Objekt besitzt. Diese minimale Erreichbarkeitsdistanz heißt auch Erreichbarkeitswert von o und wird dem Attribut $o.reach$ zugewiesen (vgl. Bedingung 4). Bedingungen 2 und 3 besagen, dass das Attribut $o.pos$ die Position des Objekts in der Clusterordnung hält und das Attribut $o.core$ eines Objektes o der Kerndistanz von o bzgl. ε und $MinPts$ entspricht.

5.4 Algorithmus OPTICS

Wie bereits in Kapitel 5.3 beschrieben, können dichtere Cluster, die in weniger dichten Clustern enthalten sind, in einem einzigen DBSCAN-ähnlichem Durchlauf gleichzeitig zusammen mit den weniger dichten Clustern gefunden werden. Voraussetzung hierfür ist die Einhaltung einer bestimmten Reihenfolge während der Bearbeitung: als nächstes Objekt muss immer dasjenige Objekt gewählt werden, das bzgl. des kleinsten ε-Wertes von einem bereits bearbeiteten Objekt dichte-erreichbar ist. Dadurch wird garantiert, dass Cluster bzgl. höherer Dichten, d.h. bzgl. niedrigerer ε-Werte früher gefunden werden als Cluster bzgl. niedrigerer Dichten.

Das Verfahren OPTICS arbeitet im Prinzip wie ein solcher erweiterter DBSCAN-Algorithmus für eine unendliche Anzahl von Distanzwerten ε_i, die kleiner einem gegebenen ε sind. Im Gegensatz zu DBSCAN werden jedoch keine Clusterzugehörigkeiten ermittelt, sondern die Objekte nach ihrer Bearbeitungsreihenfolge in einer Clusterordnung gemäß Definition 5.10 zusammengefasst.

Der Algorithmus besteht aus dem Hauptprogramm `OPTICS`, der Prozedur `expandClusterOrder` und der Prozedur `OrderedSeeds::update`. Der Pseudocode für die Prozedur `OPTICS` ist in Listing 5.3 angegeben, die Listings 5.4 und 5.5 zeigen den Pseudocode für die Prozeduren `expandClusterOrder` bzw. `OrderedSeeds::update`.

```
OPTICS(Object[] DB, Real epsilon,
       Integer minPts, OutputFile file)
  // Input: Objektmenge DB, Parameter ε und MinPts,
  //        Ausgabedatei

  file.open();

  for i = 1 to DB.size do
    Object o := DB.get(i);
    if not o.handled then
      expandClusterOrder(o, epsilon, minPts, file);

  file.close();
```

Listing 5.3: Algorithmus OPTICS

Das Hauptprogramm arbeitet ähnlich dem Hauptprogramm in DBSCAN. Die Objekte der Datenmenge werden der Reihe nach betrachtet und falls sie noch nicht bearbeitet wurden, an die Prozedur `expandClusterOrder` übergeben. Dort wird zunächst die Erreichbarkeitsdistanz des übergebenen Objekts o auf UNDEFINIERT gesetzt und dessen Kerndistanz bestimmt. Das Objekt

```
expandClusterOrder(Object o, Real epsilon,
                   Integer minPts, OutputFile file)
  // Input: Objekt o des aktuellen Clusters,
  //        Parameter ε und MinPts, Ausgabedatei


  o.reach := UNDEFINED;
  updateCoreDistance(o);
  o.handled := true;
  file.write(o);

  if o.core != UNDEFINED then
    orderedSeeds.update(neighbors, o);

    while orderedSeeds != ∅ do
      Object p := orderedSeeds.next();
      updateCoreDistance(p);
      p.handled := true;
      file.write(p);

      if p.core != UNDEFINED then
        Object[] neighbors := neighbors(p, epsilon);
        orderedSeeds.update(neighbors, p);
```

Listing 5.4: OPTICS: Prozedur `expandClusterOrder`

o wird anschließend in die Ausgabedatei geschrieben. Falls das Objekt o kein Kernobjekt ist, terminiert die Prozedur an dieser Stelle. Ansonsten werden die Objekte aus der ε-Nachbarschaft von o bestimmt und in die Priority Queue `orderedSeeds` nach ihrer Erreichbarkeitsdistanz relativ zu o eingefügt. Nun wird iterativ immer das nächste Objekt `current` aus `orderedSeeds` ausgewählt und dessen Nachbarschaft und Kerndistanz bestimmt. Ist dieses Objekt wiederum ein Kernobjekt, dann werden seine Nachbarn ebenfalls in `orderedSeeds` einsortiert. Zum Abschluss wird das Objekt als bearbeitet markiert und die Ausgabedatei geschrieben. Das Einsortieren eines Nachbarn in die Priority Queue `orderedSeeds` wird von der Prozedur `OrderedSeeds::update` durchgeführt. Ist das Objekt noch nicht in `orderedSeeds` enthalten, wird sein Erreichbarkeitswert berechnet und das Objekt wird an die richtige Stelle in der Priority Queue eingefügt. Falls das Objekt bereits in der Priority Queue enthalten ist und der neue Erreichbarkeitswert kleiner dem alten ist, wird das Objekt in der Priority Queue entsprechend nach vorne geschoben. So ist gewährleistet, dass die Objekte in `orderedSeeds` zu jedem Zeitpunkt nach der kleinsten Erreichbarkeitsdistanz, die diese Objekte relativ zu einem bereits bearbeiteten Objekt besitzen, sortiert sind. Existieren mehrere Objekte mit identischem Erreichbarkeitswert, so werden diese Objekte entsprechend ihrer Objekt-ID

```
OrderedSeeds::update(Object[] neighbors, Object o)
  // Input: Liste der Nachbarobjekte die eingefügt
  // werden sollen, Zentrumsobjekt o

  for each n in neighbors do
    if not n.handled then
      Real newReach := max(o.core, dist(o, n));

      // n ist noch nicht in orderedSeeds enthalten
      if n.reach = UNDEFINED then
        n.reach := newReach;
        insert(n, newReach);

      // n ist schon in orderedSeeds enthalten
      // => orderedSeeds ggf. aktualisieren
      else
        if newReach < n.reach then
          n.reach := newReach;
          decrease(n, newReach);
```

Listing 5.5: OPTICS: Prozedur `OrderedSeeds::update`

geordnet. Der Algorithmus wählt somit immer dasjenige Objekt als nächstes aus, das die kleinste Erreichbarkeitsdistanz relativ zu einem bereits bearbeiteten Objekt besitzt. Die Laufzeitkomplexität von OPTICS beträgt $O(n\cdot$ Aufwand zur Bestimmung einer ε-Nachbarschaft).

5.5 Visualisierung der Clusterstruktur

Abbildung 5.7 illustriert eine vom Algorithmus OPTICS erstellte Clusterordnung anhand eines zweidimensionalen Beispiels. Dabei sind für die einzelnen Objekte die Kerndistanzen als dunkle Balken und die Erreichbarkeitswerte als helle Balken eingezeichnet. Dichtere Gebiete sind durch kleine Werte für die Kerndistanzen und Erreichbarkeitswerte zu erkennen. Größere Sprünge in der Clusterordnung von einem Cluster zum nächsten sind dadurch zu identifizieren, dass der erste Punkt des neuen Clusters, da er schon zu einem dichteren Gebiet gehört, eine kleine Kerndistanz, aber, da er von einem entfernt liegendem Punkt in der Clusterordnung erreicht wurde, einen hohen Erreichbarkeitswert aufweist. Diese Beziehung wird für eine automatisierte Analyse der Clusterstruktur benötigt. Für eine direkte visuelle Clusteranalyse genügt es, das sogenannte Erreichbarkeitsdiagramm zu betrachten.

Mit einem Erreichbarkeitsdiagramm kann die Clusterstruktur für kleine Datenmengen geeignet visualisiert werden. Dazu werden die Erreichbarkeits-

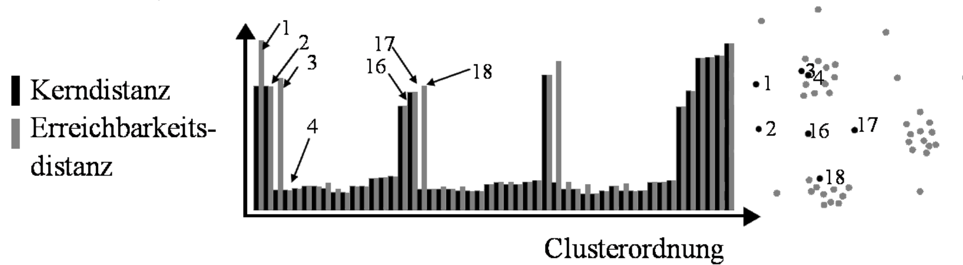


Abbildung 5.7: Kern- und Erreichbarkeitsdistanzen in der Clusterordnung

werte für jedes Objekt in der Reihenfolge der Clusterordnung als senkrechte, nebeneinander liegende Balken in ein Koordinatensystem eingetragen. Dichtere Gebiete sind dabei durch kleinere Erreichbarkeitswerte erkennbar, den Übergang von einem Cluster zum nächsten erkennt man an den Sprüngen in den Erreichbarkeitswerten. Cluster sind somit als Vertiefungen im Graphen erkennbar. Abbildung 5.8 zeigt zwei Beispiele von Erreichbarkeitsdiagrammen.

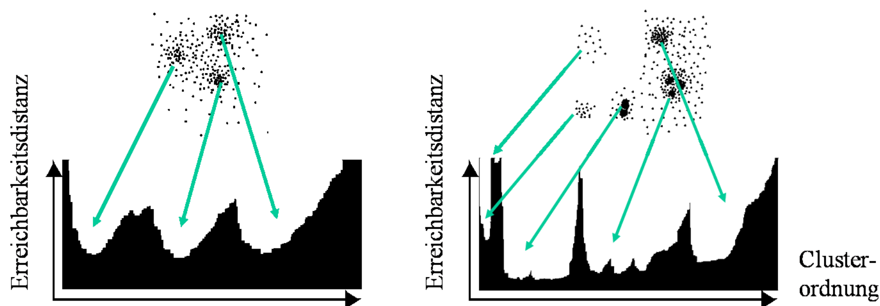


Abbildung 5.8: Erreichbarkeitsdiagramme für verschiedene Datenmengen

Kapitel 6

Inkrementelles OPTICS

Im folgenden wird eine Version des Algorithmus OPTICS aus Kapitel 5.4 entwickelt, die eine dynamische Aktualisierung einer Clusterordnung unterstützt. Das Einfügen und Löschen von Objekten erfolgt dabei inkrementell, ohne dass die Clusterordnung komplett neu erstellt werden muss. Die Grundidee des inkrementellen Algorithmus ist es, die Reorganisation der Clusterordnung auf eine begrenzte Teilmenge der Objektmenge zu beschränken und die von der Reorganisation nicht betroffenen Objekte aus der alten Clusterordnung zu übernehmen. In [KKG03] wurde bereits eine inkrementelle Version von OPTICS vorgestellt, deren konzeptioneller Ansatz ähnlich der hier entwickelten Version ausgestaltet ist. Signifikante Unterschiede bestehen jedoch hinsichtlich konkreter Umsetzung und Performanz.

Durch das Einfügen oder Löschen eines Objekts können sich die Kerndistanzen und als Folge davon auch die Erreichbarkeitswerte einiger Objekte ändern. Diese Objekte können dann unter Umständen ihre Position in der Clusterordnung verändern. Um die Menge der Objekte zu erhalten, die infolge einer Updateoperation ihre Kerndistanz ändern können, muss eine sogenannte Reverse-$MinPts$-Nearest-Neighbor-Anfrage durchgeführt werden, die alle diejenigen Objekte zurückliefert, zu denen das Anfrageobjekt $MinPts$-Nächster-Nachbar ist. Zur effizienten Durchführung einer solchen Anfrage wird im ersten Abschnitt dieses Kapitels der RDkNN-Baum vorgestellt. Anschließend wird der inkrementelle Algorithmus zum Einfügen und Löschen von Objekten in eine bestehende Clusterordnung detailliert erläutert. Im letzten Abschnitt dieses Kapitels erfolgt die Präsentation der experimentellen Untersuchung des inkrementellen OPTICS Algorithmus.

6.1 Der RDkNN-Baum

Bei einer k-Nearest-Neighbor-Anfrage auf eine Objektmenge $\mathcal{O}$ werden zu einem gegeben Anfrageobjekt q diejenigen k Objekte in $\mathcal{O}$ gesucht, die zu q am nächsten liegen. Das sogenannte Reverse-k-Nearest-Neighbor Problem besteht darin, in einer Objektmenge $\mathcal{O}$ diejenigen Objekte zu finden, in deren k-Nächster-Nachbarn-Menge ein gegebenes Anfrageobjekt q enthalten ist. Der von Yang und Lin in [YL01] vorgeschlagenen RDNN-Baum (R*-Tree with Distance of Nearest Neighbor) stellt eine Indexstruktur basierend auf dem R*-Baum [BKSS90] zur effizienten Bearbeitung von Nearest-Neighbor- und Reverse-Nearest-Neighbor-Anfragen dar. Im folgenden wird eine Erweiterung des RDNN-Baumes, der sogenannte RDkNN-Baum vorgestellt. Dieser unterstützt die effiziente Bearbeitung von k-Nearest-Neighbor- und Reverse-k-Nearest-Neighbor-Anfragen auf Objektmengen.

6.1.1 Grundlagen

Formal lassen sich die Begriffe k-Nearest-Neighbor bzw. Reverse-k-Nearest-Neighbor wie folgt definieren:

Definition 6.1 (k-Nearest-Neighbor). Sei k eine natürliche Zahl mit $k \geq 1$ und $\mathcal{O}$ eine Objektmenge mit mindestens k Objekten. Dann ist die Menge der k-Nearest-Neighbor des Anfrageobjekts q die kleinste Menge $NN_q(k) \subseteq \mathcal{O}$, die mindestens k Objekte enthält und für die gilt:

$$\forall o \in NN_q(k): \ \forall p \in \mathcal{O} \backslash NN_q(k) : \text{dist}(o,q) < \text{dist}(p,q)$$

Definition 6.2 (Reverse-k-Nearest-Neighbor). Sei k eine natürliche Zahl mit $k \geq 1$ und $\mathcal{O}$ eine Objektmenge mit mindestens k Objekten. Dann ist die Menge der Reverse-k-Nearest-Neighbor des Anfrageobjekts q, kurz $RNN_q(k)$ wie folgt definiert:

$$RNN_q(k) = \{p \in O \mid q \in NN_p(k)\}$$

Für jedes Objekt $p \in \mathcal{O}$ gilt: q ist k-Nearest-Neighbor von p genau dann, wenn $\text{dist}(p,q) \leq \text{nnDist}(p,k)$, wobei $\text{nnDist}(p,k)$ die k-Nächste-Nachbarn-Distanz von p ist (vgl. Definition 5.7). Damit lässt sich die Menge der Reverse-k-Nearest-Neighbor von q auch folgendermaßen ausdrücken: $RNN_q(k) = \{p \in O \mid \text{dist}(p,q) \leq \text{nnDist})p,k)\}$.

Die Mengen der k-Nearest-Neighbor bzw. Reverse-k-Nearest-Neighbor eines Anfrageobjekts q sind in Abbildung 6.1 für $k = 1$ illustriert. Im allgemeinen besteht kein Zusammenhang zwischen der Menge der k-Nearest-Neighbor und der Menge der Reverse-k-Nearest-Neighbor eines Objekts, d.h. $p \in NN_q(k) \nRightarrow p \in RNN_q(k)$ und umgekehrt.

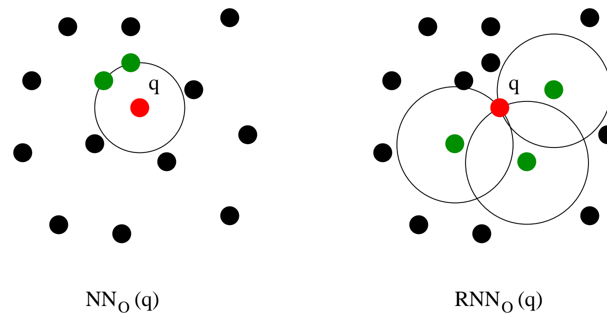


Abbildung 6.1: k-Nearest-Neighbor bzw. Reverse-k-Nearest-Neighbor von q für $k = 1$

6.1.2 Struktur des RDkNN-Baums

Der RDkNN-Baum stellt die kanonische Erweiterung des RDNN-Baums [YL01] für einen festen Parameter $k > 1$ dar. Der Aufbau erfolgt analog zum RDNN-Baum, d.h. die Datenobjekte selbst sind in einem R*-Baum [BKSS90] organisiert. Ein Blattknoten, d.h. eine Datenseite eines RDkNN-Baums enthält Einträge der Form $(oid, knnDist)$, wobei oid ein Objekt p der Datenmenge $\mathcal{O}$ referenziert und $knnDist$ die k-Nächste-Nachbarn-Distanz $\text{nnDist}(p, k)$ dieses Objekts für ein fest vorgegebenes k bezeichnet. Ein innerer Knoten, d.h. eine Directoryseite hält ein Array von Verzweigungen der Form $(ptr, mbr, maxDist)$. Dabei bezeichnet ptr die Adresse des Kindknotens. Ist dieser Kindknoten ein Blatt, dann ist mbr das minimal umgebende Rechteck aller Objekte in diesem Blatt. Zeigt ptr auf einen inneren Knoten, dann bezeichnet mbr das minimal umgebende Rechteck aller Rechtecke mbr, die einen Eintrag in diesem inneren Knoten darstellen. Es gilt $maxDist = \max\{\text{nnDist}(p, k)\}$, wobei p alle Datenpunkte des Teilbaums bezeichnet, der diesen inneren Knoten zur Wurzel hat. In Abbildung 6.2 ist der Aufbau eines RDkNN-Baums skizziert.

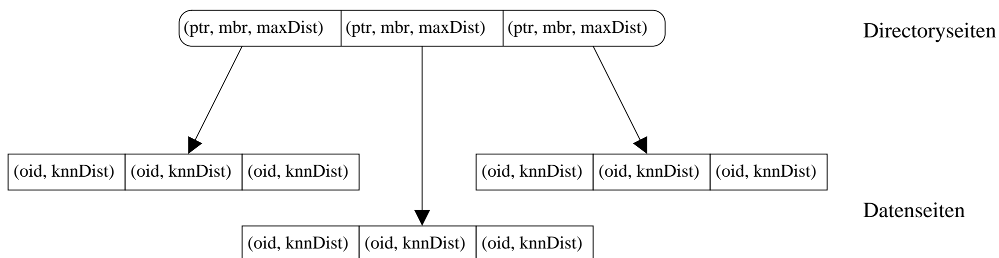


Abbildung 6.2: Struktur eines RDkNN-Baums

6.1.3 Reverse-k-Nearest-Neighbor-Anfrage

Eine Reverse-k-Nearest-Neighbor-Anfrage auf einem RDkNN-Baum ähnelt einer Punktanfrage im R*-Baum [BKSS90]. Der einzige Unterschied besteht im Entscheidungskriterium, welche Zweige des Baumes für die Suche relevant sind. In Listing 6.1 ist der Pseudocode für die Reverse-k-Nearest-Neighbor-Suche angegeben, die mit der Wurzel des RDkNN-Baums als Startknoten beginnt (vgl. [YL01]).

Sei q der Anfragepunkt, zu dem die Menge der Reverse-k-Nearest-Neighbor $RNN_q(k)$ aus der Objektmenge $\mathcal{O}$ bestimmt werden soll. Dann wird auf Blattebene für jeden Eintrag p die Distanz $\text{dist}(p, q)$ des Anfrageobjekts q zum Eintrag p mit der k-Nächsten-Nachbarn-Distanz $\text{nnDist}(p, k)$ von p verglichen. Gilt $\text{dist}(p, q) \leq \text{nnDist}(p, k)$, d.h. ist p mindestens so nahe an q, wie an seinen k-Nächsten-Nachbarn in $\mathcal{O}$, so gehört p zur Menge der Reverse-k-Nearest-Neighbors von q.

Bei einem inneren Knoten muss der Anfragepunkt q mit jedem Eintrag $B_i = (ptr_i,\ mbr_i,\ maxDist_i)$ der Directoryseite verglichen werden. Da laut Definition alle Punkte des Teilbaums mit Wurzel B_i im Rechteck mbr_i enthalten sind und die k-Nächste-Nachbarn-Distanz von jedem dieser Punkte kleiner gleich $maxDist_i$ ist, können diejenigen Teilbäume B_i von der Suche ausgeschlossen werden, deren minimal umgebendes Rechteck mbr_i eine größeren Distanzwert $\text{minDist}(mbr_i, q)$ zum Anfrageobjekt q aufweist als die maximale Distanz $maxDist_i$ in diesem Eintrag. In diesem Fall kann kein Objekt in B_i näher am Anfrageobjekt q sein, als an seinen k-Nächsten-Nachbarn in $\mathcal{O}$.

Das Distanzmaß zwischen dem minimal umgebenden Rechteck mbr_i und dem Anfrageobjekt q ist dabei von zentraler Bedeutung: es bestimmt die minimale Distanz des Rechtecks mbr_i (und damit auch aller in mbr_i eingeschlossenen Objekte) zum Anfrageobjekt o und ist in Abbildung 6.3 für den zweidimensionalen Fall veranschaulicht. Formal ist die Distanz $\text{minDist}(mbr, q)$ zwischen einem Rechteck mbr und einem Punkt q wie folgt definiert:

Definition 6.3 (minDist). [RKV95] Sei $\mathcal{O}$ eine Objektmenge n-dimensionaler Punkte. Dann ist die Distanz zwischen einem n-dimensionalen Rechteck mbr und einem Punkt $p = (p_1, \ldots, p_n) \in O$, kurz $\text{minDist}(mbr, p)$ wie folgt definiert:

$$\text{minDist}(mbr, p) = \sum_{i=1}^{n} |r_i - p_i|^2$$

wobei p_i die Koordinate des Punktes p in der i-ten Dimension, lb_i das Minimum und ub_i das Maximum der Koordinaten des Rechtecks mbr in der jeweiligen

Dimension i bezeichnen und $r_i = \begin{cases} lb_i & \text{, falls } p_i < lb_i \\ ob_i & \text{, falls } p_i > ob_i \\ p_i & \text{, sonst} \end{cases}$.

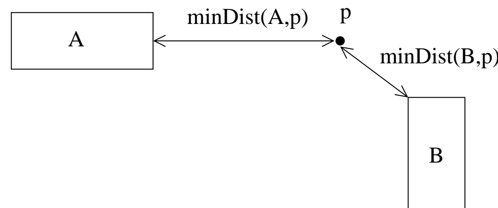


Abbildung 6.3: Definition von minDist

```
RkNN(Node n, Point q, Point[] result)
  // Input : Knoten n in dem RkNN gesucht werden soll,
  //         Anfrageobjekt q, Ergebnisliste result
  // Output: Reverse-k-Nearest-Neighbors von q
  //         in der Ergebnisliste result

  //n ist Blatt
  if n.isLeaf() then
    for each Point p = (oid, knnDist) in n do
      if dist(q,p) <= p.knnDist then
        result.add(p);

  //n ist innerer Knoten
  else
    for each Branch B = (ptr, mbr, maxDist) in n do
      if minDist(B.mbr,q) <= B.maxDist then
        rnn(B.ptr,q);
```

Listing 6.1: RDkNN-Baum: Reverse-k-Nearest-Neighbor-Anfrage

6.1.4 k-Nearest-Neighbor-Anfrage

Eine k-Nearest-Neighbor-Anfrage in einem RDkNN-Baum ist identisch zur bekannten Nächsten-Nachbarn-Suche im R-Baum (vgl. [RKV95]). Die grundlegende Idee besteht darin, diejenigen Knoten aus der Priority Queue zu entfernen, deren minimal umgebenden Rechtecke mbr eine größere Distanz minDist(mbr, q) zum Anfragepunkt q aufweisen als das Maximum der Distanzwerte von q zu den bisher gefundenen k-Nächsten-Nachbarn. Die durch diese

minimal umgebenden Rechtecke approximierten Teilbäume können offensichtlich nicht die k-Nächsten-Nachbarn von q enthalten. Die Priority Queue ist aufsteigend nach den minDist-Werten der Elemente zu q sortiert und es wird jeweils das erste Element dieser Liste zur Bearbeitung ausgewählt.

In Listing 6.2 ist der Pseudocode einer k-Nächsten-Nachbarn-Anfrage zu einem Anfragepunkt q und einem festen Parameter k angegeben.

```
kNN(Point q)
  // Input : Anfrageobjekt q
  // Output: k-Nearest-Neighbors von q
  //         in der Ergebnisliste result

  KNNList result;
  PriorityQueue pq;
  pq.add(0, root);

  while pq != ∅ and
        dist(pq.first,q) <= result.maxDist do
    Node n := pq.removeFirst();

    //n ist Blatt
    if n.isLeaf() then
      for each Point p = (oid, knnDist) in n do
        if dist(q,p) <= result.maxDist then
          result.add(p);

    //n ist innerer Knoten
    else
      for each Branch B = (ptr, mbr, maxDist) in n do
        if minDist(B.mbr,q) <= result.maxDist then
          pq.add(minDist(B.mbr,q), B.ptr);

  return result;
```

Listing 6.2: RDkNN-Baum: k-Nearest-Neighbor-Anfrage

6.1.5 Einfügen und Löschen von Punkten

Beim Einfügen eines neuen Punktes q in einen bestehenden RDkNN-Baum muss zum einen die Menge der k-Nearest-Neighbor von q bestimmt werden, um die k-Nächste-Nachbarn-Distanz *knnDist* als Parameter für den neuen Eintrag von q zu berechnen. Zum anderen müssen diejenigen Punkte aus der Objektmenge $\mathcal{O}$ überprüft werden, die in der Menge der Reverse-k-Nearest-Neighbor von q enthalten sind, denn durch das Einfügen von q werden deren k-Nächste-Nachbarn verändert. Deshalb muss die k-Nächste-Nachbarn-Distanz

knnDist dieser Punkte neu berechnet und die minimal umgebenden Rechtecke *mbr* ihrer Vorfahren entsprechend angepasst werden. Diese Operationen erfolgen in der Prozedur `preInsert`, die vor dem eigentlichen Einfügen des neuen Objekts in den R*-Baum (vgl. hierzu [BKSS90]) aufgerufen wird. In Listing 6.3 ist der Pseudocode der Prozedur `preInsert` aufgeführt, die mit der Wurzel des RDkNN-Baums als Startknoten initiiert wird.

Auf Blattebene wird dabei für jeden Eintrag p die Distanz $\text{dist}(p, q)$ des neuen Objekts q zum Eintrag p mit dem Maximum der Distanzwerte von q zu seinen bisher gefundenen k-Nächsten-Nachbarn verglichen. Ist p näher an q, als der entfernteste bisher gefundene k-Nächste-Nachbar von q, so ist p ein potentieller k-Nächster-Nachbar von q und wird zur Kandidatenliste hinzugefügt und die k-Nächste-Nachbarn-Distanz *knnDist* von q wird aktualisiert. Falls die Distanz zwischen p und q kleiner ist als die k-Nächste-Nachbarn-Distanz *knnDist* von p, dann ist q ein neuer k-Nächste-Nachbar von p und der Eintrag von p muss modifiziert werden. Dazu wird von allen bisherigen $k-1$-Nächsten-Nachbarn von p die maximale Distanz zu p bestimmt. Die neue k-Nächste-Nachbarn-Distanz von p ergibt sich dann aus dem Maximum dieser Distanzen und der Distanz $\text{dist}(p, q)$.

Bei einem inneren Knoten muss der Anfragepunkt q mit jedem Eintrag $B_i = (ptr_i, mbr_i, maxDist_i)$ der Directoryseite verglichen werden. Es können diejenigen Teilbäume B_i von einer weiteren Suche ausgeschlossen werden, deren minimal umgebende Rechtecke eine Distanz zum neuen Objekt q aufweisen, die zum einen größer gleich der maximale Distanz $maxDist_i$ in diesem Teilbaum ist und zum anderen größer gleich dem Maximum der Distanzwerte von q zu seinen bisher gefundenen k-Nächsten-Nachbarn ist. In diesem Fall kann der Teilbaum B_i weder einen k-Nächsten-Nachbarn von q enthalten, noch kann q ein k-Nächster-Nachbar von Objekten in diesem Teilbaum sein.

```
preInsert(Node n, Point q, KNNList knn_q)
  // Input : Knoten n für den preInsert durchgeführt
  //         werden soll, einzufügender Punkt q,
  //         Kandidatenliste der bislang gefundenen kNN(q)
  // Output: maximale knnDist im Knoten n;

  double newMaxDist := 0;

  //n ist Blatt
  if n.isLeaf() then
    for each Point p = (oid, knnDist) in n do
      // p ist näher an q als der am weitesten
      // entfernte kNN-Kandidat von q
      // => p ist Kandidat für kNN(q)
      if dist(p,q) <= knn_q.maxDist then
        knn_q.add(p);
        q.knnDist := knn_q.maxDist;

      // p ist näher an q, als an seinem entferntesten
      // kNN => q ist kNN von p
      if dist(p,q) <= p.knnDist then
        KNNList knn_p = kNN(p);
        double knnDist_p := 0;
        for each Point r = (oid, knnDist) in knn_p do
          knnDist_p := max(knnDist_p, dist(p,r));
        // setze neue knnDist bei p
        p.knnDist := max(knnDist_p, dist(p,q));

      // aktualisiere maxDist im Knoten
      newMaxDist := max(newMaxDist, p.knnDist);

  //n ist innerer Knoten
  else
    // B aufsteigend nach Distanz minDist(B.mbr,q)
    for each Branch B = (ptr, mbr, maxDist) in n do
      if minDist(B.mbr,q) < B.maxDist or
         minDist(B.mbr,q) < knn_q.maxDist then
        // setze neue maxDist bei B
        B.maxDist := preInsert(ptr, q, knn_q);

      // aktualisiere maxDist im Knoten
      newMaxDist := max(newMaxDist, B.maxDist);

  return newMaxDist;
```

Listing 6.3: RDkNN-Baum: Prozedur `preInsert`

Beim Löschen eines Punktes q aus einem bestehenden RDkNN-Baum muss die Menge der Reverse-k-Nearest-Neighbor von q bestimmt werden, um die k-Nächste-Nachbarn-Distanzen *knnDist* dieser Punkte zu aktualisieren. Das bedeutet wiederum, dass für jeden Punkt aus $RNN_q(k)$ eine k-Nearest-Neighbor-Anfrage durchgeführt werden muss. Da die Objekte aus $RNN_q(k)$ im Regelfall nahe beieinander liegen, können diese Anfragen zu einem Schritt zusammengefasst werden. Es wird eine sogenannte Batch-k-Nearest-Neighbor-Suche durchgeführt, die die k-Nächsten-Nachbarn für eine Menge von Anfrageobjekten $q_1, \ldots, q_n$ berechnet. Dabei werden rekursiv die Knoten mit der minimalen Distanz zu jedem q_i besucht. Listing 6.5 zeigt den Pseudocode für die Prozedur `batchNN`. Um den Punkt q physikalisch aus dem RDkNN-Baum zu löschen, wird der Standard R*-Baum-Algorithmus zum Löschen eines Punktes (vgl. [BKSS90]) verwendet. Der Pseudocode für das Löschen eines Objekts und das Aktualisieren der k-Nächste-Nachbarn-Distanzen *knnDist* der dadurch betroffenen Punkte ist in den Listings 6.4 und 6.6 dargestellt.

```
delete(Point q)
  // Input : zu löschender Punkt q,

  // R*-Baum-Algorithmus um q zu löschen
  r*.delete(q);

  // Reverse k Nearest Neighbors von q
  Point[] rnn = RkNN(q);

  // Batch-kNN Anfrage, um die k Nearest Neighbors
  // für alle RkNN(q) zu finden
  KNNList[] knnList = batch_kNN(root, rnn);

  // Anpassung der knnDist in den Objekten
  for i=0 to knnList.size do
    KNNList knn = knnList[i];
    rnn[i].knnDist := knn.maxDist;

  // Anpassung der maxDist in den Knoten
  adjustDnn(root);
```

Listing 6.4: RDkNN-Baum: Prozedur `delete`

```
batchNN(Node n , Point[] qList, KNNList[] result)
  // Input : Knoten n, in dem gesucht wird
  //         Liste qList der Anfrageobjekte zu denen
  //         die kNN ermittelt werden sollen
  //         Kandidatenliste der bislang gefundenen kNN
  // Output: k-Nearest-Neighbors zu jedem r ∈ rnn
  //         in der Ergebnisliste result

  //n ist Blatt
  if n.isLeaf() then
    for each Point p = (p_oid, p_knnDist) in n do
      for each Point q = (q_oid, q_knnDist) in qList do
        KNNList knn_q = result[q];
        // p ist näher an q als der am weitesten
        // entfernte kNN-Kandidat von q
        // => p ist Kandidat für kNN(q)
        if p != r and
            dist(p,q) < knn_q.maxDist then
            knn_q.add(p);

  //n ist innerer Knoten
  else
    // sortiere B aufsteigend nach minimaler
    // minDist(B.mbr, q) zu jedem q ∈ qList
    for each Branch B = (ptr, mbr, maxDist) in n do
      double minDist = MAXVALUE;
      for each Point q = (q_oid, q_knnDist) in rnn do
        minDist = min(minDist , minDist(B.mbr, q));
      pq.add(minDist, B.ptr);

    while pq != ∅ do
      double dist = pq.firstPriority();
      Node current = pq.removeFirst();
      // durchsuche nur Knoten mit der jeweils kleinsten
      // Distanz zu jedem q ∈ qList
      for each Point q = (q_oid, q_knnDist) in qList do
        KNNList knn_q = result[q];
        if dist <= knn_q.maxDist then
          batchNN(current, rnn, result);
          break;
```

Listing 6.5: RDkNN-Baum: Prozedur `batchNN`

```
adjust_kNNDist(Node n)
  // Input  : Knoten n für den die knnDist
  //          angepasst werden soll
  // Output: maximale knnDist im Knoten n;

  double n_maxDist = Double.MIN_VALUE;

  //n ist Blatt
  if (n.isLeaf()) then
    for each Point p = (oid, knnDist) in n do
      n_maxDist := max(maxDist, p.knnDist);

  //n ist innerer Knoten
  else
    for each Branch B = (ptr, mbr, maxDist) in n do
      B.maxDist := adjust_kNNDist(B.ptr);
      n_maxDist := max(n_maxDist, B.maxDist);

  return n_maxDist;
```

Listing 6.6: RDkNN-Baum: Prozedur `adjust_kNNDist`

6.2 Algorithmus Inkrementelles OPTICS

Der im folgenden vorgestellte Algorithmus unterstützt die dynamische Aktualisierung einer Clusterordnung, die mit Hilfe des Algorithmus OPTICS aus Kapitel 5.4 berechnet wurde. Das Einfügen und Löschen von Objekten erfolgt dabei inkrementell, ohne dass die Clusterordnung komplett neu erstellt werden muss.

Durch das Einfügen oder Löschen eines Objekts können sich die Kerndistanzen und damit auch die Erreichbarkeitswerte einiger Objekte ändern. Diese Objekte können dann unter Umständen Bedingung 1 von Definition 5.10 verletzen. Sie weisen, verglichen mit allen ihnen nachgeordneten Objekten keine minimale Erreichbarkeitsdistanz relativ zu einem vor ihnen in der Clusterordnung enthaltenem Objekt auf. Dies impliziert, dass diese Objekte keine gültige Position mehr in der Clusterordnung besitzen. Infolge dessen muss die Clusterordnung reorganisiert werden, so dass oben genannte Bedingung für eine gültige Clusterordnung wieder erfüllt ist. Der Grundgedanke des inkrementellen OPTICS ist es, nicht für jedes Objekt diese Bedingung zu überprüfen, sondern die Reorganisation auf eine begrenzte Teilmenge der Objektmenge zu beschränken und die von der Reorganisation nicht betroffenen Objekte aus der alten Clusterordnung zu übernehmen.

Um den Algorithmus zu formulieren, werden die Begriffe Reverse-Neighbors bzgl. ε und $MinPts$, sowie Vorgänger, Nachfolger und rekursive Nachfolger eines Objekts benötigt, die im folgenden definiert werden.

6.2.1 Definitionen

Definition 6.4 (Reverse Neighbors bzgl. ε und $MinPts$). Sei q ein Objekt und bezeichne CO eine Clusterordnung bzgl. ε und $MinPts$. Dann bezeichnet die Menge der Reverse Neighbors bzgl. ε und $MinPts$ von q, kurz $RN_{\varepsilon,MinPts}(q)$ die Menge der Objekte in CO, die bedingt durch das Einfügen bzw. Löschen von q ihre Kerndistanz ändern können. $RN_{\varepsilon,MinPts}(q)$ ist dabei wie folgt definiert:

$$RN_{\varepsilon,MinPts}(q) = \{o \in RNN_q(MinPts) \mid \text{dist}(o,q) \leq \varepsilon\}$$

Wird in die ε-Umgebung eines Objekts o ein Objekt q eingefügt bzw. wird aus der ε-Umgebung eines Objekts o ein Objekt q gelöscht, so kann dies zur Folge haben, dass sich die Kerndistanz des Objekts o verringert bzw. erhöht. Ist dies der Fall, dann folgt aus Definition 5.8, dass es sich bei dem eingefügten bzw. gelöschten Objekt q um einen $MinPts$-Nächsten-Nachbarn von o handelt. Um die Menge der Objekte $RN_{\varepsilon,MinPts}$ zu erhalten, die infolge einer Updateoperation ihre Kerndistanz ändern können, muss demnach eine Reverse-$MinPts$-Nearest-Neighbor-Anfrage durchgeführt werden, die alle diejenigen

Objekte o zurückliefert, die in $RNN_q(MinPts)$ enthalten sind und deren Distanz $\text{dist}(o, q)$ kleiner gleich ε ist. Zur effizienten Durchführung einer solchen Anfrage wird der in Kapitel 6.1 vorgestellte RDkNN-Baum eingesetzt.

Definition 6.5 (Vorgänger). Sei CO eine Clusterordnung bzgl. ε und $MinPts$ und o ein Objekt mit $o \in CO$. Für jedes Element $p \in CO$ ist der Vorgänger, kurz $\text{pre}(p)$ wie folgt definiert:

$$\text{pre}(p) = \begin{cases} o & \text{, falls } p.reach = \text{reachDist}_{\varepsilon,MinPts}(o,p) \\ \text{UNDEFINED} & \text{, falls } p.reach = \infty \end{cases}$$

Der Vorgänger eines Objekts p in einer Clusterordnung CO ist dasjenige Objekt o in der Clusterordnung, von dem aus p erreicht wurde, dabei gilt $o.pos < p.pos$. Wurde p von keinem Objekt erreicht, dann ist der Erreichbarkeitswert von p unendlich. Im anderen Fall entspricht der Erreichbarkeitswert von p der Erreichbarkeitsdistanz von p relativ zu o.

Definition 6.6 (Nachfolger). Sei CO eine Clusterordnung bzgl. ε und $MinPts$. Für jedes Objekt $p \in CO$ ist die Menge seiner Nachfolger, kurz $\text{suc}(p)$ wie folgt definiert:

$$\text{suc}(p) = \{o \in CO \mid \text{pre}(o) = p\}$$

Die Nachfolger $\text{suc}(p)$ eines Objektes p sind all diejenigen Objekte o in der Clusterordnung, die von p aus erreicht wurden, dabei gilt: $p.pos < o.pos$. Es können Objekte innerhalb einer Clusterordnung CO existieren, die keinen Vorgänger und/oder Nachfolger besitzen.

Definition 6.7 (Rekursive Nachfolger). Sei CO eine Clusterordnung bzgl. ε und $MinPts$. Für jedes Objekt $p \in CO$ ist die Menge seiner rekursiven Nachfolger, kurz $\text{suc}_{\text{rec}}(p)$ wie folgt rekursiv definiert:

- $o \in \text{suc}(p) \Rightarrow o \in \text{suc}_{\text{rec}}(p)$
- $o \in \text{suc}_{\text{rec}}(p)$ und $q \in \text{suc}(o) \Rightarrow q \in \text{suc}_{\text{rec}}(p)$

6.2.2 Einfügen eines neuen Objekts

Im folgenden bezeichnet CO_{old} die alte Clusterordnung vor dem Einfügen eines neuen Objekts q in die Objektmenge $\mathcal{O}$, CO_{new} die neue Clusterordnung nach dem Einfügen von q, die mittels der Prozedur `insert` berechnet wird. Der Pseudocode für das Einfügen eines neuen Objekts q ist in Listing 6.7 dargestellt.

Um den Vorgänger pre(o) bzw. die Nachfolger suc(o) eines Objekts o zu erhalten, erhält jedes Objekt $o \in CO$ die zusätzlichen Attribute *o.pre* bzw. *o.suc*, die den Vorgänger bzw. die Menge der Nachfolger von o bezeichnen. Um diese Attribute konsistent zu halten, wird die Prozedur *OrderedSeeds* :: *update* aus Listing 5.5 wie in Listing 6.8 angegeben modifiziert. Die Modifikation besteht darin, dass bei Objekten n, die in die Liste `orderedSeeds` eingefügt werden, das Objekt o, aus dessen ε-Umgebung n stammt, als (potentieller) Vorgänger gesetzt wird. Ist o nicht spezifiziert, dann erhält n den Erreichbarkeitswert ∞. Erhält ein Objekt n schließlich seine endgültige Position in der Clusterordnung, wird zur Menge der Nachfolger *o.succ* seines Vorgängers o das Objekt n hinzugefügt. Auf diese Weise ist gewährleistet, dass sowohl nach Ablauf des inkrementellen Algorithmus als auch nach Ablauf des originalen OPTICS-Algorithmus die Attribute *o.pre* und *o.succ* bei jedem Objekt o der Clusterordnung konsistent sind.

Wird in die ε-Umgebung eines Objekts o das Objekt q eingefügt, so kann sich dadurch die Kerndistanz des Objekts o verringern. Deshalb wird in einem Vorbereitungsschritt für jedes Element o aus der Menge der Reverse Neighbors $RN_{\varepsilon,MinPts}(q)$ des einzufügenden Objekts q die Kerndistanz coreDist(o) aktualisiert. Danach wird q in die Priority Queue `orderedSeeds` mit einem Erreichbarkeitswert von ∞ eingefügt, da noch unbekannt ist, von welchem Objekt $o \in CO_{old}$ das neue Objekt q in der neuen Clusterordnung CO_{new} erreicht werden wird.

Im Anschluss daran beginnt die eigentliche Reorganisation der Clusterordnung, die den Algorithmus OPTICS imitiert. Dazu wird in jedem Schleifendurchlauf das nächste noch nicht bearbeiteten Objekt aus der alten Clusterordnung CO_{old} mit dem ersten Objekt in der Priority Queue `orderedSeeds` verglichen. Das Objekt mit dem kleineren Erreichbarkeitswert wird dann in die nächste freie Position der neuen Clusterordnung CO_{new} eingefügt. Sind die beiden Erreichbarkeitswerte identisch, wird das Objekt mit der kleineren Objekt-ID gewählt, da im OPTICS-Algorithmus die Objekte in der Priority Queue `orderedSeeds` bei gleichem Erreichbarkeitswert ebenfalls nach der Objekt-ID geordnet sind. Für den Fall, dass die beiden Objekte sowohl in ihren Erreichbarkeitswerten als auch in ihrer Objekt-ID übereinstimmen, wird das Objekt, dessen Vorgänger die kleinere Objekt-ID besitzt in die neue Clusterordnung eingefügt, da dieses Objekt auch vom OPTICS-Algorithmus ausgewählt werden würde.

Nachdem das nächste Objekt p in die Clusterordnung CO_{new} eingefügt wurde, muss die Priority Queue `orderedSeeds` aktualisiert werden. Wurde p aus der alten Clusterordnung CO_{old} übernommen, ist zu überprüfen, ob das Updateobjekt q bereits bearbeitet wurde. Wurde q noch nicht bearbeitet, dann kann p ein potentieller Vorgänger von q werden, falls das Objekt q in der ε-

Umgebung von p liegt und diese mehr als $MinPts$ Objekte beinhaltet. In diesem Fall ist die Priority Queue entsprechend zu aktualisieren. War das zuletzt eingefügte Objekt p aus der Priority Queue oder ein Reverse Neighbor des Updateobjekts q, so müssen die potentiellen Nachfolger von p in die Priority Queue eingefügt werden. Zu den potentiellen Nachfolgern von p zählen alle diejenigen noch nicht bearbeiteten Objekte aus der ε-Umgebung von p, die keinen Vorgänger besitzen oder deren Vorgänger noch nicht in die neue Clusterordnung eingefügt wurden oder deren Erreichbarkeitswerte größer gleich der Kerndistanz von p sind. Wurde der Vorgänger eines Nachbarobjekts von p bereits in CO_{new} eingefügt und ist dessen Erreichbarkeitswert kleiner als die Kerndistanz von p, so kann dieses Objekt kein Nachfolger von p werden und muss nicht in die Priority Queue aufgenommen werden.

Die Reorganisation terminiert, wenn die alte Clusterordnung CO_{old} keine unbearbeiteten Objekte mehr beinhaltet und die Priority Queue `orderedSeeds` leer ist. Die auf diese Weise berechnete neue Clusterordnung CO_{new} ist dann identisch zur Clusterordnung, die der originale OPTICS-Algorithmus aus der Objektmenge $\mathcal{O} \cup \{q\}$ berechnet.

```
insert(Object q, Object[] rn)
  // Input:  einzufügendes Objekt q,
  //         rn: RN_{ε,MinPts}(q)

  // Kerndistanzen der RN(q) aktualisieren
  for each Object o in rn do
    updateCoreDistance(o);

  // Objekt q in orderedSeeds mit reachability MAXVALUE einfügen
  updateOrderedSeeds(q, null);

  while co_old.hasUnhandled() or orderedSeeds != ∅ do
    Object c := co_old.nextUnhandled();
    Object o := orderedSeeds.nextUnhandled();

    // nächstes Objekt für co_new bestimmen
    boolean addFromSeeds := false;
    if (o.reach < c.reach) or
       (o.reach = c.reach and o.id < c.id) then
      orderedSeeds.remove(o);
      co_new.add(o);
      addFromSeeds = true;

    else if (o.reach > c.reach) or
            (o.reach = c.reach and o.id > c.id) then
      co_new.add(c);

    else
      Object preOld = c.pre;
      Object preNew = o.pre;
      if (preOld.id < preNew.id)
        co_new.add(c);
      else co_new.add(o);

    Object p = co_new.last();
    p.handled := true;

    // orderedSeeds aktualisieren
    if addFromSeeds or rn.contains(p) then
      // potentielle Nachfolger von p
      orderedSeeds.update(potentialSuccs(p, epsilon),p);
    else if not q.handled and p.core <= epsilon and
            dist(q,p) <= epsilon then
      // p kann Vorgänger von q werden
      orderedSeeds.update(q,p);
```

Listing 6.7: Inkrementelles OPTICS: Prozedur `insert`

```
OrderedSeeds::update(Object[] objects, Object o)
  // Input: Liste objects der Objekte die eingefügt
  //        werden sollen,
  //        möglicher Vorgänger o dieser Objekte

  Real newReach;
  for each n in objects do
    if not n.handled then
      if o = null then newRech := MAX_VALUE;
      else newReach := max(o.core, dist(o, n));

      // n ist noch nicht in orderedSeeds enthalten
      if not orderedSeeds.contains(n) then
        n.reach := newReach;
        insert(n, newReach);
        if newReach != MAX_VALUE then
          n.pre := o;

      // n ist schon in orderedSeeds enthalten
      // => orderedSeeds ggf. aktualisieren
      else if newReach < n.reach then
        n.reach := newReach;
        decrease(n, newReach);
        n.pre := o;
```

Listing 6.8: Inkrementelles OPTICS: Prozedur `OrderedSeeds::update`

6.2.3 Löschen eines bestehenden Objekts

Der Pseudocode für das Löschen eines bestehenden Objekts q ist in Listing 6.7 dargestellt. Zur Aktualisierung der Priority Queue wird eine neue Prozedur `OrderedSeeds::updateAll` benötigt, deren Pseudocode in Listing 6.10 angegeben ist. Dabei wird für alle Objekte n in der Priority Queue überprüft, ob sie von dem übergebenen Objekt o aus dichte-erreichbar sind. Ist die Erreichbarkeitsdistanz von n relativ zu o kleiner als der bisher gefundene Erreichbarkeitswert von n, dann wird der Eintrag für das Objekt n in der Priority Queue entsprechend aktualisiert.

Das zu löschende Objekt q wird zu Beginn als bereits bearbeitet markiert, damit es nicht von der alten Clusterordnung CO_{old} in die neue Clusterordnung CO_{new} übernommen wird. Durch das Löschen des Objekts q aus der ε-Umgebung eines Objekts o kann sich die Kerndistanz des Objekts o erhöhen. Deshalb wird analog zum Einfügen für jedes Objekt $o \in RN_{\varepsilon,MinPts}(q)$ die Kerndistanz coreDist(o) aktualisiert. Aufgrund der (eventuell) geänderten Kerndistanzen der Reverse Neighbors von q können alle rekursiven Nachfolger

der Objekte der Menge $RN_{\varepsilon,MinPts}(q)$ ihren Erreichbarkeitswert ändern und ihre Position in der neuen Clusterordnung CO_{new} verschieben. Daher werden alle rekursiven Nachfolger der Reverse Neighbors $RN_{\varepsilon,MinPts}(q)$ des Updateobjekts q in die Priority Queue mit einem Erreichbarkeitswert von ∞ eingetragen, da noch nicht bekannt ist, von welchen Objekten sie zukünftig erreicht werden.

In jedem Schleifendurchlauf der Reorganisationsphase wird das nächste noch nicht bearbeiteten Objekt c aus der alten Clusterordnung CO_{old}, das nicht in den rekursiven Nachfolgern von q enthalten ist, mit dem nächsten Objekt o in der Priority Queue `orderedSeeds` verglichen. Ist der Vorgänger des Objekts c noch nicht bearbeitet, dann kann auch c nicht aus der alten Clusterordnung übernommen werden und das nächste Objekt p für die neue Clusterordnung wird aus der Priority Queue gewählt. Ansonsten wird dasjenige Objekt p als nächstes in CO_{new} eingefügt, das den kleinsten Erreichbarkeitswert besitzt.

Im Anschluss daran muss die Priority Queue `orderedSeeds` in Abhängigkeit von p aktualisiert werden. Handelt es sich bei dem zuletzt eingefügten Objekt p um ein Kernobjekt, dann kann p zum einen ein potentieller Vorgänger aller noch in der Priority Queue vorhandenen Objekte sein. Zum anderen müssen auch alle bisherigen direkten Nachfolger von p in die Priority Queue eingefügt werden, denn auch für diese Objekte kann gelten, dass sie aufgrund geänderter Kerndistanzen ihren Erreichbarkeitswert und damit ihre Position in der Clusterordnung ändern.

Die Reorganisation terminiert, wenn die alte Clusterordnung CO_{old} keine unbearbeiteten Objekte mehr beinhaltet und die Priority Queue `orderedSeeds` leer ist. Die auf diese Weise berechnete neue Clusterordnung CO_{new} ist dann identisch zur Clusterordnung, die der originale OPTICS-Algorithmus aus der Objektmenge $\mathcal{O}\backslash\{q\}$ berechnet.

```
delete(Object q, Object[] rn)
  // Input:  zu löschendes Objekt q,
  //         rn: RN_{ε,MinPts}(q)

  q.handled := true;

  // Kerndistanzen der RN(q) aktualisieren
  for each Object o in rn do
    updateCoreDistance(o);

  // alle rekursiven Nachfolger von rn berechnen und
  // in orderedSeeds mit reachability MAXVALUE einfügen
  Object[] recSuccs = getRecSuccessors(rn);
  orderedSeeds.update(recSuccs, null);

  while co_old.hasUnhandled() or orderedSeeds != ∅ do
    // nächstes unbearbeitetes Objekt, das nicht
    // in recSuccs enthalten ist
    Object c := co_old.nextUnhandled(recSuccs);
    boolean addFromSeeds := false;
    Object preC := c.pre;
    if not preC.handled then
      addFromSeeds := true;

    Object o := orderedSeeds.nextUnhandled();

    // nächstes Objekt für co_new bestimmen
    if o.reach <= c.reach or addFromSeeds then
      orderedSeeds.remove(o);
      co_new.add(o);
      addFromSeeds = true;
    else if c.reach < o.reach then
      co_new.add(c);

    Object p = co_new.last();
    p.handled := true;

    // orderedSeeds aktualisieren
    if p.core <= epsilon then
      orderedSeeds.updateAll(p, epsilon);
      // alle Nachfolger von p einfügen
      orderedSeeds.update(p.successor, p);
```

Listing 6.9: Inkrementelles OPTICS: Prozedur `delete`

```
OrderedSeeds::updateAll(Object o, Real epsilon)
  // Input: möglicher Vorgänger o aller Einträge in orderedSeeds
  //        Parameter ε

    for each n in orderedSeeds do
      if dist(o, n) <= epsilon then
        Real newReach := max(o.core, dist(o, n));
        if newReach < n.reach then
          n.reach := newReach;
          decrease(n, newReach);
          n.pre := o;
```

Listing 6.10: Inkrementelles OPTICS: Prozedur `OrderedSeeds::updateAll`

6.2.4 Einfügen und Löschen mehrere Objekte

Die in den vorangegangenen Kapiteln vorgestellten Algorithmen zum Einfügen und Löschen eines einzelnen Objekts können relativ einfach auf das Einfügen und Löschen mehrerer Objekte erweitert werden. Die Pseudocodes für ein sogenanntes Bulk-Insert bzw. Bulk-Delete sind in den Listings 6.11 und 6.12 angegeben. Der Unterschied zu den Prozeduren `insert` bzw. `delete` besteht lediglich darin, dass nicht ein einzelnes Objekt q übergeben wird, sondern eine Liste von Objekten, die in die Clusterordnung eingefügt bzw. aus ihr gelöscht werden sollen. Entsprechend dazu werden die Kerndistanzen aller Reverse Neighbors dieser Updateobjekte aktualisiert. Anschließend werden beim Einfügen anstatt eines einzelnen Objets alle Updateobjekte in die Priority Queue eingefügt, der Rest der Prozedur `bulkInsert` verläuft identisch zur Prozedur `insert`. Beim Löschen werden analog die rekursiven Nachfolger aller Reverse Neighbors der zu löschenden Objekte in die Priority Queue `orderedSeeds` eingetragen, die Reorganisationsschleife verläuft auch hier identisch zur Prozedur `delete`.

```
insertBulk(Object[] insertions, Object[] rn)
  // Input:   insertions: einzufügende Objekte,
  //          rn: RN_{ε,MinPts}(q) aller q ∈ insertions

  // Kerndistanzen der RN aktualisieren
  for each Object o in rn do
    updateCoreDistance(o);

  // alle Objekte ∈ insertions in orderedSeeds
  // mit reachability MAXVALUE einfügen
  updateOrderedSeeds(insertions, null);

  while co_old.hasUnhandled() or orderedSeeds != ∅ do
    // analog zur Prozedur insert ...
```

Listing 6.11: Inkrementelles OPTICS: Prozedur `bulkInsert`

```
deleteBulk(Object deletions, Object[] rn)
  // Input:   deletions: zu löschende Objekte
  //          rn: RN_{ε,MinPts}(q) aller q ∈ deletions

  for each Object q in deletions do
    q.handled := true;

  // Kerndistanzen der RN(q) aktualisieren
  for each Object o in rn do
    updateCoreDistance(o);

  // alle rekursiven Nachfolger von rn berechnen und
  // in orderedSeeds mit reachability MAXVALUE einfügen
  Object[] recSuccs = getRecSuccessors(rn);
  orderedSeeds.update(recSuccs, null);

  while co_old.hasUnhandled() or orderedSeeds != ∅ do
    // analog zur Prozedur delete ...
```

Listing 6.12: Inkrementelles OPTICS: Prozedur `bulkDelete`

6.3 Experimentelle Untersuchung

In diesem Abschnitt werden die Ergebnisse der experimentellen Untersuchung des inkrementellen OPTICS Algorithmus präsentiert. Alle Experimente wurden auf einer Workstation mit einem Intel Xeon Prozessor mit 3.2 GHz Taktfrequenz und einem Arbeitsspeicher von 3,5 GB durchgeführt.

Die Evaluierung des inkrementellen OPTICS Algorithmus wurde sowohl für synthetische Daten als auch für Realdaten vorgenommen. Die fünf synthetischen Datensätze bestanden aus 100.000, 200.000, 300.000, 400.000 und 500.000 zweidimensionalen Punkten. Die Realdaten bestanden aus 112.361 TV-Bildern, die in 64-dimensionalen Farbhistogrammen abgelegt waren.

6.3.1 Einfügen und Löschen einzelner Punkte

Für jeden Datensatz wurde eine Clusterordnung mit dem originalen OPTICS-Algorithmus erzeugt. Danach wurden pro Datensatz nacheinander 500 zufällig ausgewählte Punkte in die Clusterordnung eingefügt und anschließend wieder gelöscht. Die durchschnittliche und maximale Laufzeit der Einfüge- bzw. Löschvorgangs des inkrementellen Algorithmus wurde mit der Gesamtlaufzeit des OPTICS Algorithmus verglichen und ist für die synthetischen Datensätze in den Abbildungen 6.4 und 6.5 dargestellt. In die Laufzeitbetrachtung ging dabei jeweils nur die Laufzeit des OPTICS-Algorithmus ein, d.h. die Zeit für den Aufbau des RDkNN-Baums wurde sowohl beim Original-Algorithmus als auch beim inkrementellen Algorithmus nicht betrachtet.

Für die synthetischen Datensätze wurde ein durchschnittlicher Beschleunigungsfaktor in der Laufzeit beim Einfügen eines Objekts von 1.300 (bei 100.000 Objekten) bis hin zu 8.500 (bei 500.000 Objekten) ermittelt. Beim Löschen eines Objekts benötigt der OPTICS Algorithmus als durchschnittliche Laufzeit das 330- bis 1.000-fache des inkrementellen Ansatzes. Selbst im worst-case Fall betrugen die Laufzeiten des OPTICS Algorithmus noch das 170- bis 700-fache des inkrementellen Ansatzes.

Die deutlich langsameren durchschnittliche Laufzeiten beim Löschen eines Objekts im Vergleich zum Einfügen eines Objekts lassen sich dadurch erklären, dass beim Löschen unbekannt ist, von welchem Objekt die (rekursiven) Nachfolger des zu löschenden Objekts nun erreicht werden. D.h. es muss für wesentlich mehr Objekte als beim Einfügen ein neuer Vorgänger bestimmt werden. Wie die Test in Kapitel 6.3.2 zeigen, relativieren sich diese Unterschiede in der Laufzeit aber bei einer höheren Anzahl von Updateobjekten. Dort ist sogar zu beobachten, dass das Löschen in einem Bulk-Update eine schneller Laufzeit als das Einfügen besitzt.

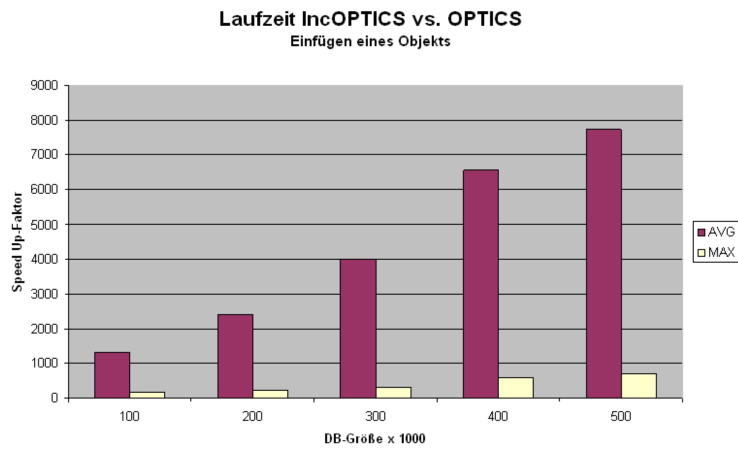


Abbildung 6.4: Laufzeitvergleich: Einfügen eines Objekts

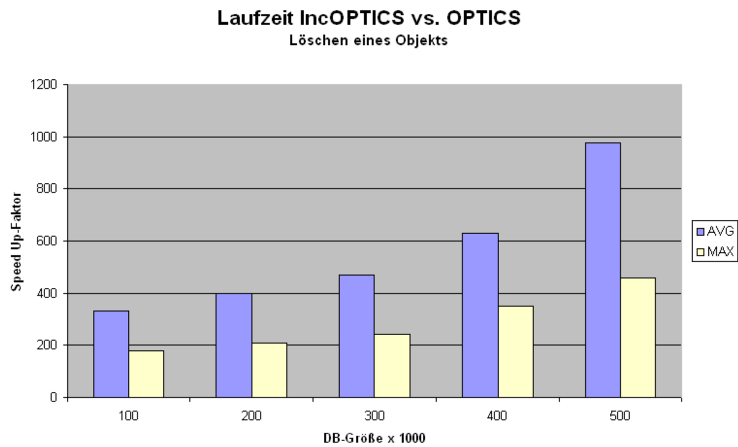


Abbildung 6.5: Laufzeitvergleich: Löschen eines Objekts

Ein wesentlicher Grund für die hohe Laufzeitsteigerung ist die Einsparung von Bereichsanfragen in der inkrementellen Version. Es wurden die durchschnittliche Anzahl von Bereichsanfragen des originalen OPTICS-Algorithmus mit der durchschnittlichen Anzahl von Bereichsanfragen in der inkrementellen Version verglichen. Dabei wurde eine Einsparung an Bereichsanfragen um das 2.000 - 10-000-fache gegenüber dem Original-Algorithmus festgestellt. Das Ergebnis dieses Vergleichs ist in Abbildung 6.6 dargestellt.

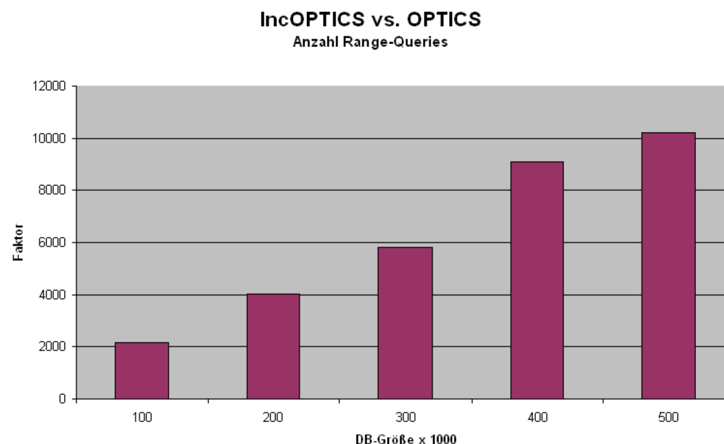


Abbildung 6.6: Vergleich Bereichsanfragen

Ähnliche Beobachtungen können auch bei der Anwendung des inkrementellen Algorithmus auf Realdaten festgestellt werden. Dort ergab sich eine durchschnittliche Einsparung an Bereichsanfragen um das 7.500-fache. Die Laufzeit wurde beim Einfügen eines Objekts um das 6.000-fache gesteigert, beim Löschen eines Objekts wurde eine Laufzeitsteigerung um das 1.800-fache festgestellt. Die Ergebnisse dieser Evaluation sind in Abbildung 6.7 dargestellt.

6.3.2 Einfügen und Löschen mehrerer Punkte

Für einen synthetischen Datensatz bestehend aus 200.000 zweidimensionalen Punkten wurde eine Clusterordnung mit dem originalen OPTICS-Algorithmus erzeugt. Danach wurden sogenannte Bulk-Updates ausgeführt. Dazu wurden zufällig ausgewählte Updateobjekte kumuliert in die Clusterordnung eingefügt und wieder gelöscht. Die Anzahl der Updateobjekte wurde dabei schrittweise um jeweils 500 Objekte ausgehend von 500 bis zu 4000 Objekten erhöht.

Die durchschnittliche Laufzeit der Einfüge- bzw. Löschvorgangs des inkrementellen Algorithmus wurde mit der Gesamtlaufzeit des OPTICS Algorith-

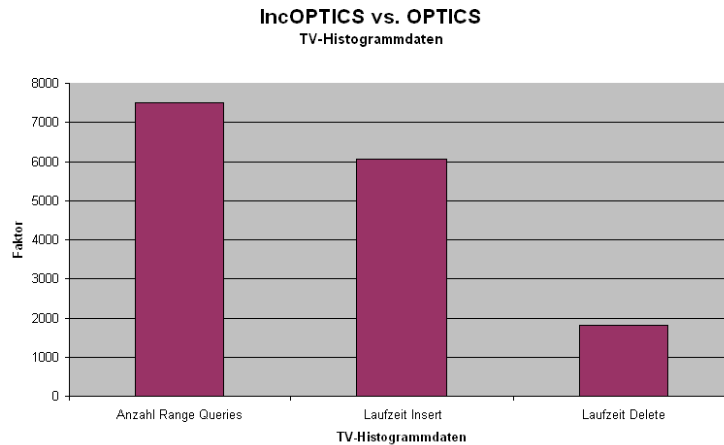


Abbildung 6.7: Laufzeitvergleich und Vergleich Bereichsanfragen für Farbhistogramme

mus verglichen und ist in der Abbildungen 6.8 dargestellt. In die Laufzeitbetrachtung ging dabei bei beiden Algorithmen sowohl die Laufzeit des OPTICS-Algorithmus als auch die Zeit für den Aufbau des RDkNN-Baums ein. Wie die Grafik verdeutlicht, ist selbst bei einer Updategröße von 20% der Originaldatenmenge der inkrementelle Algorithmus immer noch schneller als der Original-Algorithmus.

Im Gegensatz zum Einzel-Update ist beim Bulk-Update zu beobachten, dass das Löschen schnellere Laufzeiten als das Einfügen besitzt. Die liegt darin begründet, dass beim Einfügen untersucht wird, ob das zuletzt in die neue Clusterordnung eingefügte Objekt o in den Reverse Neighbors aller Updateobjekte war. Ist dies der Fall, dann werden die potentiellen Nachfolger von o in die Priority Queue eingefügt. Bei einer hohen Anzahl an Updateobjekten erhöht sich die Wahrscheinlichkeit, dass o in der Menge der Reverse Neighbors enthalten ist und somit werden tendenziell mehr Elemente in die Priority Queue eingefügt.

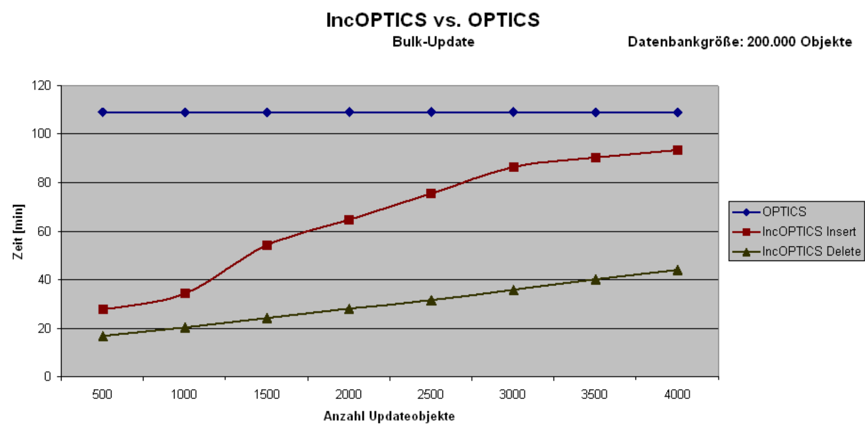


Abbildung 6.8: Laufzeitvergleich Bulk-Update

Kapitel 7

Zusammenfassung

7.1 Ergebnis

In einem sogenannten Data Warehouse werden alle in einem Unternehmen oder einer Organisation vorhandenen Informationen gespeichert. Ziel der Clusteranalyse ist es, in einem Data Warehouse Daten mit ähnlichen Merkmalen (semi-) automatisch zu identifizieren und anhand dieser in Gruppen (Cluster) einzuteilen, die sich möglichst stark voneinander unterscheiden. Im Rahmen dieser Diplomarbeit wurden inkrementell arbeitende Versionen der hierarchischen Clusteringverfahren Single-Link und OPTICS entwickelt.

Der inkrementelle Single-Link-Algorithmus baut auf dem Verfahren SLINK [Sib73] auf und ermöglicht es, in ein bestehendes Single-Link-Dendrogramm Objekte einzufügen bzw. aus einem bestehendem Dendrogramm Objekte zu löschen. Für die Darstellung der hierarchischen Baumstruktur eines Dendrogramms wird dabei eine sehr kompakte Form, die sogenannte Pointer-Repräsentation gewählt. Obwohl die Pointer-Repräsentation aus Anwendersicht nicht sehr übersichtlich ist, besitzt sie den Vorteil, dass sie sehr effizient berechnet werden kann. Zur Überführung der Pointer-Repräsentation in eine anwenderfreundliche Baumstruktur existiert eine entsprechende Methode.

Das Einfügen eines neuen Objekts o_{n+1} in ein aus n Objekten bestehendes Dendrogramm entspricht im wesentlichen dem $n+1$-ten Rekursionsschritt des SLINK-Algorithmus und besitzt eine Laufzeitkomplexität von $O(n)$. Das Löschen eines Objekts gestaltet sich komplexer, da ein Rückwärtsschritt bedingt durch den Rekursionsansatz nicht möglich ist. Es werden zunächst die Cluster gesucht, deren Level kleiner ist als eine kritische Distanz λ_{crit}. Für diese Cluster gilt, dass sie durch das Löschen nicht verändert werden. Im nachfolgenden Schritt wird ausgehend von diesen Clustern mit SLINK sukzessive das neue Dendrogramm aufgebaut. Das Löschen eines Objekts aus einem Dendrogramm besitzt bedingt durch den Single-Link-Ansatz eine Laufzeitkomplexität

von $O(n^2)$.

Der inkrementelle OPTICS Algorithmus unterstützt die dynamische Aktualisierung einer bestehenden Clusterordnung. Das Einfügen und Löschen von Objekten erfolgt dabei inkrementell, ohne dass die Clusterordnung komplett neu erstellt werden muss. Die Grundidee des inkrementellen Algorithmus ist es, die Reorganisation der Clusterordnung auf eine begrenzte Teilmenge der Objektmenge zu beschränken und die von der Reorganisation nicht betroffenen Objekte aus der alten Clusterordnung zu übernehmen.

Um die Menge der Objekte zu berechnen, die von einer Updateoperation betroffen sind, muss eine sogenannte Reverse-$MinPts$-Nearest-Neighbor-Anfrage durchgeführt werden, die alle diejenigen Objekte zurückliefert, zu denen das Updateobjekt $MinPts$-Nächster-Nachbar ist. Zur effizienten Durchführung einer solchen Anfrage wird der RDkNN-Baum eingesetzt. Mit Hilfe dieser Reverse-$MinPts$-Nearest-Neighbor kann die Reorganisation der Clusterordnung effizient durchgeführt werden. In der experimentelle Untersuchung des inkrementellen OPTICS Algorithmus wurden signifikante Laufzeitverbesserungen gegenüber dem originalen OPTICS Algorithmus festgestellt.

7.2 Ausblick

Ein Ziel für zukünftige Arbeiten ist ein formaler Korrektheitsbeweis des inkrementellen OPTICS Algorithmus. Es wurde zwar sowohl anhand der Überlegungen gezeigt, als auch in den anschließenden Tests bestätigt, dass der inkrementellen OPTICS-Algorithmus eine zum Original-Algorithmus identische Clusterordnung erzeugt, ein mathematischer Beweis steht aber noch aus.

Desweiteren wird für höher dimensionale Daten eine geeignete Indexstruktur benötigt, die ähnlich zum RDkNN-Baum für niedrig dimensionale Daten Reverse-k-Nearest-Neighbor-Anfragen effizient unterstützt. Ein möglicher Lösungsansatz hierbei wäre die kanonische Übertragung des Grundkonzepts des RDkNN-Baums auf einen X-Baum.

Einen weiterer Aspekt für zukünftige Arbeiten stellt die Weiterentwicklung der im Rahmen dieser Diplomarbeit vorgestellten inkrementellen Algorithmen für das Konzept der Data Bubbles dar. Data Bubbles [BKKS01] sind speziell für hierarchische Clusteringalgorithmen entwickelte Repräsentanten, auf die anstelle der Originaldatenmenge ein hierarchisches Clusteringverfahren angewendet wird, um die Skalierbarkeit der Clustering-Anwendung zu verbessern.

Abbildungsverzeichnis

2.1 Die Schritte des KDD-Prozesses 7

3.1 Dendrogramm: Zerlegung der Datenmenge in vier Cluster 11
3.2 Single-Link, Complete-Link und Average-Link Distanz 14
3.3 Single-Link bzw. Complete-Link Cluster 16
3.4 Parameter in der Lance-Williams Rekursionsformel 17

4.1 Dendrogramm mit korrespondierender Pointer-Repräsentation . 20
4.2 Einfügen der neuen Objekt-ID 5 24
4.3 Dendrogramm vor dem Löschen der Objekt-ID 2 28
4.4 Ermitteln der gültigen Cluster 28
4.5 Resultierendes Dendrogramm nach dem Löschen der Objekts mit der ID 2 29

5.1 Kernobjekte und direkte Dichte-Erreichbarkeit für $MinPts = 3$ 32
5.2 Dichte-Erreichbarkeit für $MinPts = 3$ 33
5.3 Dichte-Verbundenheit für $MinPts = 3$ 33
5.4 Ineinander enthaltene, unterschiedlich dichte Cluster 36
5.5 k-Nächste-Nachbarn-Distanz $\mathrm{nnDist}(o, k)$ für $k = 4$ 37
5.6 Kerndistanz und Erreichbarkeitsdistanz für $MinPts = 4$ 38
5.7 Kern- und Erreichbarkeitsdistanzen in der Clusterordnung . . . 42
5.8 Erreichbarkeitsdiagramme für verschiedene Datenmengen 42

6.1 k-Nearest-Neighbor bzw. Reverse-k-Nearest-Neighbor von q für $k = 1$ 45
6.2 Struktur eines RDkNN-Baums 45
6.3 Definition von minDist 47
6.4 Laufzeitvergleich: Einfügen eines Objekts 65
6.5 Laufzeitvergleich: Löschen eines Objekts 65
6.6 Vergleich Bereichsanfragen 66
6.7 Laufzeitvergleich und Vergleich Bereichsanfragen für Farbhistogramme 67

6.8 Laufzeitvergleich Bulk-Update 68

Quellcodeverzeichnis

4.1 Überführung der Pointer-Repräsentation in eine Baumstruktur . 21
4.2 Inkrementelles Single-Link: Einfügen eines neuen Objekts 23
4.3 Inkrementelles Single-Link: Löschen eines Objekts 26
4.4 Inkrementelles Single-Link: Berechnung der gültigen Cluster . . 27
5.1 Algorithmus DBSCAN 34
5.2 DBSCAN: Prozedur `expandCluster` 35
5.3 Algorithmus OPTICS 39
5.4 OPTICS: Prozedur `expandClusterOrder` 40
5.5 OPTICS: Prozedur `OrderedSeeds::update` 41
6.1 RDkNN-Baum: Reverse-k-Nearest-Neighbor-Anfrage 47
6.2 RDkNN-Baum: k-Nearest-Neighbor-Anfrage 48
6.3 RDkNN-Baum: Prozedur `preInsert` 50
6.4 RDkNN-Baum: Prozedur `delete` 51
6.5 RDkNN-Baum: Prozedur `batchNN` 52
6.6 RDkNN-Baum: Prozedur `adjust_kNNDist` 53
6.7 Inkrementelles OPTICS: Prozedur `insert` 58
6.8 Inkrementelles OPTICS: Prozedur `OrderedSeeds::update` . . . 59
6.9 Inkrementelles OPTICS: Prozedur `delete` 61
6.10 Inkrementelles OPTICS: Prozedur `OrderedSeeds::updateAll` . 62
6.11 Inkrementelles OPTICS: Prozedur `bulkInsert` 63
6.12 Inkrementelles OPTICS: Prozedur `bulkDelete` 63

Literaturverzeichnis

[ABKS99] ANKERST, M. ; BREUNIG, M. M. ; KRIEGEL, H.-P. ; SANDER, J.: OPTICS: Ordering Points to Identify the Clustering Structure. In: *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'99)*, ACM Press, New York, 1999, S. 49–60

[And73] ANDERBERG, M. R.: *Cluster analysis for applications*. New York and London : Academic Press, 1973

[BKKS01] BREUNIG, M. M. ; KRIEGEL, H.-P. ; KRÖGER, P. ; SANDER, J.: Data Bubbles: Quality Preserving Performance Boosting for Hierarchical Clustering. In: *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'01)*, ACM Press, New York, 2001, S. ??

[BKSS90] BECKMANN, N. ; KRIEGEL, H.-P. ; SCHNEIDER, R. ; SEEGER, B.: The R*-Tree: an efficient and robust access method for points and rectangles. In: *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'90)*, ACM Press, New York, 1990, S. 322–331

[Def77] DEFAYS, D.: An efficient algorithm for a complete link method. In: *The Computer Journal* 20 (1977), Nr. 4, S. 364–366

[EHW89] EL-HAMDOUCHI, A. ; WILLET, P.: Comparison of Hierarchic Agglomerative Clustering Methods for Document Retrieval. In: *The Computer Journal* 32 (1989), Nr. 3, S. 220–227

[EKSX96] ESTER, M. ; KRIEGEL, H.-P. ; SANDER, J. ; XU, X.: A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In: *Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining (KDD'96)*, AAAI Press, Menlo Park, 1996, S. 226–231

[ES00] Ester, M. ; Sander, J.: *Knowledge Discovery in Databases: Techniken und Anwendungen*. 1. Berlin Heidelberg : Springer-Verlag, 2000

[ESM01] Everitt, B. S. ; S., Landau ; M., Leese: *Cluster Analysis*. New York : Oxford University Press Inc., 2001

[FBY92] Frakes, W. B. ; Baeza-Yales, R.: *Information Retrieval*. Englewood Cliffs, New Jersey : Prentice Hall, 1992

[FHS96] Fayyad, U. M. ; Haussler, D. ; Stolorz, P.: KDD for Science Data Analysis: Issues and Examples. In: *Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining (KDD'96)*, AAAI Press, Menlo Park, 1996, S. 50–56

[FLP+51] Florek, K. ; Lukaszewicz, J. ; Perkal, J. ; H., Steinhaus ; Zubrzycki, S.: Sur la liaison et la division des points d'un ensemble fini. In: *Colloquium Math* 2 (1951), S. 182–185

[GR69] Gower, J. C. ; Ross, G. J. S.: Minimum spanning trees and single-linkage cluster analysis. In: *Applied Statistics* 18 (1969), S. 54–64

[JD88] Jain, A. K. ; Dubes, R. C.: *Algorithms for Clustering Data*. Englewood Cliffs, New Jersey : Prentice Hall, 1988

[JMF99] Jain, A. K. ; Murty, M. N. ; Flynn, P. J.: Data Clustering: A Review. In: *ACM Computing Surveys* 31 (1999), Nr. 3, S. 264–323

[JS77] Jardine, N. ; Sibson, R.: *Mathematical Taxonomy*. London, New York, Sidney, Toronto : John Wiley & Sons, 1977

[KKG03] Kriegel, H.-P. ; Kröger, P. ; Gotlibovich, I.: Incremental OPTICS: Efficient Computation of Updates in a Hierarchical Cluster Ordering. In: *Proceedings of 5th International Conference on Data Warehousing and Knowledge Discovery (DaWaK'03)*. Prague, Czech Republic, 2003, S. 224–233

[KR90] Kaufman, L. ; Rousseauw, P. J.: *Finding Groups in Data: An Introduction to Cluster Analysis*. London, New York, Sidney, Toronto : John Wiley & Sons, 1990

[LW67] Lance, G. N. ; Williams, W. T.: A general theory of classificatory sorting strategies: 1. Hierarchical systems. In: *The Computer Journal* 9 (1967), S. 173–380

[Mac67] MacQueen, J.: Some Methods for Classification and Analysis of Multivariate Observations. In: *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability* Bd. 1, University of California Press, Berkeley, 1967, S. 281–297

[NH94] Ng, R. T. ; Han, J.: Efficient and Effective Clustering Methods for Spatial Data Mining. In: *Proceedings of 20th International Conference on Very Large Data Bases (VLDB'94)*, Morgan Kaufmann Publishers, San Francisco, California, 1994, S. 144–155

[RKV95] Roussopoulos, N. ; Kelley, S. ; Vincent, F.: Nearest neighbor queries. In: *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'95)*, ACM Press, New York, 1995, S. 71–79

[Roh82] Rohlf, F. J.: Single-link Clustering Algorithms. In: *Krishnaiah P. R., Kanal L. N., eds. Handbook of Statistics* Bd. 2, North-Holland Publishing Company, 1982

[Sib73] Sibson, R.: SLINK: An optimally efficient algorithm for the single-link cluster method. In: *The Computer Journal* 16 (1973), Nr. 1, S. 30–34

[Sor48] Sorensen, T.: A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on Danish commons. In: *Biologiske Skifter* 5 (1948), S. 1–34

[SQL+03] Sander, J. ; Qin, X. ; Lu, Z. ; Niu, N. ; Kovarsky, A.: Automatic Extraction of Clusters from Hierarchical Clustering Representations. In: *Proceedings of 7th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'03)*, Springer, 2003, S. 75–87

[Voo86] Voorhees, E. M.: Implementing agglomerative hierarchic clustering algorithms for use in document retrieval. In: *Information Processing and Management* 22 (1986), Nr. 6, S. 465–576

[Wil88] Willet, P.: Recent trends in hierarchical document clustering: a critical review. In: *Information Processing and Management* 24 (1988), Nr. 5, S. 577–597

[Wil97] Wille, D.: *Repetitorium der Linearen Algebra - Teil 1*. Springe : Binomi Verlag, 1997

[YL01] Yang, C. ; Lin, K.-I.: An index structure for efficient reverse nearest neighbor queries. In: *Proceedings of 17th IEEE International Conference on Data Engineering.* Heidelberg, Germany, 2001, S. 485–492