*Diplomarbeit*

# Mehrdimensionale Zugriffspfad–strukturen für das ähnlichkeitsbasierte Retrieval von Fällen

*Hannes Öchsner*

AG Künstliche Intelligenz – Expertensysteme
Prof. Dr. Michael M. Richter

Sonderforschungsbereich 314
"Künstliche Intelligenz – Wissensbasierte Systeme"
Projekt X9

September 1992

<u>*Betreuer:*</u>

Dipl. Inform. Stefan Wess
Dipl. Inform. Jürgen Paulokat

Universität Kaiserslautern

# Inhaltsverzeichnis


**I Einführung** **1**

**1 Übersicht** **2**
1.1 Aufgabenstellung . . . 2
1.2 Aufbau der Arbeit . . . 3

**2 Fallbasiertes Schließen — eine Einordnung** **4**

**II Grundlagen** **9**

**3 Der Prozeß des fallbasierten Schließens** **10**
3.1 Ein Modell . . . 10
3.2 Retrieval von Fällen aus der Fallbasis . . . 12
3.2.1 Organisationsstrukturen für die Fallbasis . . . 12
3.2.2 Repräsentation der Fallbeispiele . . . 12

**4 Kognitionswissenschaftliche Grundlagen des fallbasierten Schließens** **15**
4.1 Das MAC/FAC–Modell . . . 15
4.1.1 Kognitionspsychologische Aspekte . . . 15
4.1.2 Das Modell . . . 16
4.1.3 Einordnung der Arbeit . . . 17
4.2 Der Ähnlichkeitsbegriff . . . 18
4.3 Bewertung der Ähnlichkeit und Eignung bekannter Fälle . . . 19
4.3.1 Ähnlichkeitsbestimmung über das Contrast–Modell . . . 20
4.3.2 Eignungsbewertung . . . 21

**5 Assoziative Suche in Retrieval–Systemen** **22**
5.1 Anfragetypen . . . 22
5.1.1 Intersection Queries . . . 23
5.1.2 Best Match Query . . . 24
5.2 Mehrdimensionale Zugriffspfade . . . 25

**6 Nearest–Neighbor–Suche** **28**
6.1 Grundbegriffe und Notationen . . . 28
6.2 Problemstellung . . . 29
6.3 Untere Schranken . . . 30
6.4 Aufwandsaspekte . . . 30

**III Ansatz** **32**

**7 Der k-d-Baum** **33**
7.1 Definition . . . 33

7.2 Aufwandsbetrachtungen . . . 35
7.2.1 Mittlere Konstruktionskosten . . . 35
7.2.2 Aufbau idealer Bäume . . . 36
7.3 Einfaches Suchen, Einfügen und Löschen . . . 37

**8 Starre NN–Suche 40**
8.1 Optimierter Aufbau eines k-d-Baumes . . . 40
8.1.1 Bestimmung der Partitionswerte . . . 41
8.1.2 Wahl des Diskriminators . . . 41
8.1.3 Die Prozedur . . . 43
8.1.4 Beispiel . . . 44
8.2 Der Algorithmus . . . 45
8.2.1 Rekursive Baumsuche . . . 45
8.2.2 Obere und untere Schranken der Knoten . . . 48
8.3 Die Bounds–Tests . . . 48
8.3.1 Grundlegende Idee . . . 49
8.3.2 Bounds–Overlap–Ball . . . 50
8.3.3 Ball–Within–Bounds . . . 51
8.4 Die Ähnlichkeitsmaße . . . 52
8.5 Suchbeispiel . . . 53
8.6 Aufwandsbetrachtungen . . . 55
8.7 k.o.–Kriterien . . . 57

**9 Inkrementelle NN–Suche 58**
9.1 Einfache Vorgehensweise . . . 58
9.2 Ein effizienter inkrementeller Algorithmus . . . 59
9.3 Beispiel . . . 62
9.4 Performance–Verbesserung . . . 63

**10 Typkonzept zur Realisierung der Ähnlichkeitssuche in k-d-Bäumen 64**
10.1 Benutzerdefinierte Typen . . . 64
10.1.1 Spezifikation . . . 64
10.1.2 Verwendung der Typen in Attributen . . . 66
10.2 Typhierarchien . . . 66

**11 Zusammenfassung und Ausblick 68**

**IV Implementierung 70**

**12 Implementierungsbeschreibung 71**
12.1 Übersicht . . . 71
12.2 Die Fallbasis . . . 71
12.2.1 Konzeption . . . 71
12.2.2 Generierung einer Fallbasis . . . 72
12.2.3 Aktualisieren einer Fallbasis . . . 73
12.2.4 Einfaches Retrieval von Fällen . . . 74
12.3 k-d-Bäume . . . 74
12.3.1 Die Klasse 'KDTree' . . . 74
12.3.2 Erzeugung eines k-d-Baumes . . . 75
12.3.3 Rebalancierung von k-d-Bäumen . . . 76
12.3.4 Best–Match–Suche . . . 77
12.4 Das Typkonzept . . . 78
12.4.1 Wertebereiche der Repräsentationsattribute . . . 78

12.4.2 Benutzerdefinierte Typen . . . . . 81
12.4.3 Die globale Typen–Liste . . . . . 82
12.4.4 Implementierung von Koordinatenähnlichkeiten . . . . . 82
12.5 Ähnlichkeitsmaße für Fälle . . . . . 84

**Anhang** **85**

**A Implementierungsbeispiele** **86**
A.1 Ähnlichkeitsmaße . . . . . 86
A.2 ApplicationTypes . . . . . 87

**B Hinweise zur Benutzung des Tools** **89**
B.1 Laden der OPAL–Codefiles . . . . . 89
B.2 Benutzung der GSI–Schnittstelle zu Smalltalk-80 . . . . . 90

**Literaturverzeichnis** **92**

# Teil I

# Einführung

# Kapitel 1

# Übersicht

## 1.1 Aufgabenstellung

Diese Diplomarbeit ist Teil des CAbPlan–Projektes im SFB 314, dessen Ziel die Erstellung eines fallbasierten Expertensystems zur Unterstützung der Fertigungsplanung für rotationssymmetrische Drehteile ist ([PPW92]).

Grob skizziert soll das System in der Lage sein, aus einer vorgegebenen Konstruktionszeichnung eines Drehteils einen Plan für die maschinelle Fertigung dieses Teils zu erstellen. Ausgehend vom Ansatz des *fallbasierten Schließens* besteht die Aufgabe des Systems darin, aus einer Menge bekannter Drehteile, für die bereits ein Fertigungsplan erstellt worden ist, das Teil zu finden, dessen Darstellung zu der des eingegebenen Teils am ähnlichsten ist. Der Plan dieses ähnlichsten Teils ist dann so zu modifizieren und anzupaßen, daß damit das vorgegebene Teil gefertigt werden kann.

Ein zentrales Problem ist hierbei die Definition des Ähnlichkeitsbegriffes, der auf jeden Fall den fertigungstechnischen Aspekt berücksichtigen muß.

Aufgabe dieser Diplomarbeit ist es, für das Planungssystem ein Basismodul zu erstellen, das sowohl die Speicherung von Fällen als auch die Bereitstellung von Mechanismen zur Unterstützung des ähnlichkeitsbasierten Fallretrievals umfaßt. Die Fallbasis soll dabei auf einer objektorientierten Datenbank implementiert werden. Die in der Fallbasis zu speichernden Fälle liegen in einer auf Attributen basierenden Repräsentation vor, die die Fälle hauptsächlich auf syntaktischer Ebene beschreibt. Durch die bereitzustellenden Retrievalmechanismen soll die Ähnlichkeitssuche in der Fallbasis auf der niedrigen Abstraktionsstufe der syntaktischen Fallbeschreibungen realisiert werden.

Solch ein ähnlichkeitsbasiertes Fallretrieval kann als *Best–Match*–Suche aufgefaßt werden, einem bekannten Problem aus dem Bereich des Information Retrieval. Der in dieser Diplomarbeit verfolgte Lösungsansatz realisiert das Retrieval von Fällen durch eine assoziative Suche in dem mehrdimensionalen Datenraum der Fallrepräsentationen, wobei die Fälle in der Fallbasis durch eine mehrdimensionale Datenstruktur entsprechend organisiert werden.

Darüber hinaus sollte ein Konzept entwickelt und implementiert werden, das die Definition benutzerdefinierter Wertebereiche für die Fallrepräsentationsattribute ermöglicht, auf denen anwendungsabhängige Ähnlichskeitsmaße spezifiziert werden können.

Die Implementierung dieser Arbeit erfolgte in Form eines generischen Tools auf dem Datenbanksystem *GemStone*. Über die Smalltalk-80–Schnittstelle von GemStone wurde das Tool an ein graphisches Entwurfssystem für rotationssymmetrische Drehteile angebunden, das als Eingabekomponente des Planungssystems im Rahmen einer weiteren Diplomarbeit ([Kie92]) entwickelt wurde.

## 1.2 Aufbau der Arbeit

Die Diplomarbeit gliedert sich in vier Teile. Der einführende erste Teil nimmt im folgenden Kapitel zunächst eine Einordnung des *fallbasierten Schließens* vor, wobei Beziehungen dieses Ansatzes zu anderen Bereichen aufgezeigt werden.

Der zweite Teil beschäftigt sich mit den theoretischen Grundlagen, auf denen die Diplomarbeit aufbaut. Nach einer Modellierung des fallbasierten Schließens in Kapitel 3 werden in Kapitel 4 die kognitionswissenschaftlichen Grundlagen für das ähnlichkeitsbasierte Fallretrieval erläutert und der in dieser Arbeit verfolgte Ansatz entsprechend eingeordnet. Die beiden nachfolgenden Kapitel betrachten das Problem der Ähnlichkeitssuche dann vom Standpunkt der Information–Retrieval–Systeme aus. Kapitel 5 stellt hierzu in herkömmlichen Retrieval– bzw. Datenbanksystemen verwendete Ansätze für die assoziative Suche vor, während in Kapitel 6 das Problem der *Nearest–Neighbor*–Suche definiert wird.

Im dritten Teil wird der Ansatz zur Lösung des Problems des ähnlichkeitsbasierten Fallretrievals dargestellt. Kapitel 7 beschreibt die verwendete mehrdimensionale Datenstruktur des *k-d-Baumes*. In den Kapiteln 8 und 9 werden dann die verwendeten Algorithmen zur Unterstützung der Nearest–Neighbor–Suche auf dieser Datenstruktur erläutert. In Kapitel 10 erfolgt schließlich die Spezifizierung eines Typkonzeptes zur Realisierung der Ähnlichkeitssuche in $k$-d-Bäumen, das die Definition und Verwendung anwendungsabhängiger Ähnlichkeitsmaße erlaubt. Kapitel 11 schließt diesen Teil mit einer kurzen Zusammenfassung und einem Ausblick ab.

Der letzte Teil dieser Arbeit widmet sich der Beschreibung der im Rahmen dieser Diplomarbeit vorgenommenen Implementierung der vorgestellten Konzepte (Fallbasis, $k$-d-Bäume, benutzerdefinierte Typen). Im Anhang sind dann noch einige Beispiele zur Implementierungsbeschreibung aufgeführt.

# Kapitel 2

# Fallbasiertes Schließen — eine Einordnung

Das fallbasierte Schließen (*Case–Based Reasoning*, CBR) wird als Teilgebiet des Maschinellen Lernens den analogen Lernverfahren zugeordnet. Diese Verfahren versuchen neue Problemsituationen dadurch zu lösen, daß sie auf bereits gelerntes Wissen zurückgreifen und dieses entsprechend auf neue Situationen übertragen.

Ein solches Vorgehen entspricht häufig den menschlichen Verhaltensweisen bei der Lösung von Problemen. Der Problemlösungsprozeß wird oft sehr stark von bereits in ähnlichen Situationen gemachten Erfahrungen beeinflußt, an die man sich bei der Konfrontation mit einem neuen unbekannten Problem wieder erinnert.

Das Gebiet des fallbasierten Schließens verknüpft Erkenntnisse und Ansätze mehrerer verschiedener Bereiche miteinander (Abb. 2.1).

Kognitionswissenschaftliche Aspekte fließen hier genauso mit ein wie Ansätze aus den Bereichen der Expertensysteme und des Maschinellen Lernens. Daneben finden bei der Realisierung fallbasierter Systeme auch Methoden und Techniken des Information Retrieval und der Datenbanksysteme Verwendung. Im folgenden werden die Einflüsse der verschiedenen Bereiche kurz skizziert. Einen allgemeinen Überblick über fallbasiertes Schließen gibt z.B. [WPA92].

### CBR und Kognitionswissenschaften

Aus kognitionspsychologischer Sicht stellt das fallbasierte Schließen ein Modell für menschliches Problemlösen dar. Die Fälle entsprechen dabei vorhandenem Erfahrungswissen. Sie abstrahieren von Ereignissen oder Prozessen, die über einen räumlichen und zeitlichen Bezug verfügen (vgl. [AWB+92]).

Grundlegende Aspekte bei der Modellierung des menschlichen Problemlösungsprozesses durch fallbasiertes Schließen sind die Repräsentation des Fallgedächtnisses, das Erinnern an gemachte Erfahrungen sowie die Übertragung bekannter Lösungen auf das aktuelle Problem.

Hierbei spielt der Begriff der *Ähnlichkeit* zwischen Fällen eine zentrale Rolle. So kann der menschliche Erinnerungsprozeß als Bereitstellung (*Retrieval*) von Fällen aus der Fallbasis angesehen werden, die zu einem gegebenen Problem möglichst ähnlich sind. Die Lösung des Problems basiert dann auf Analogien zu diesen bereitgestellten Fällen (*Lösungstransfer*).

Das Gebiet der Kognitionswissenschaften liefert hierzu auf psychologischen Erkenntnissen basierende Modelle für die Bestimmung und Bewertung von Ähnlichkeiten zwischen Fällen.

### CBR und Expertensysteme

Fallbasiertes Schließen ist derzeit insbesondere im Zusammenhang mit Expertensystemen Gegenstand intensiver Forschungsaktivitäten.

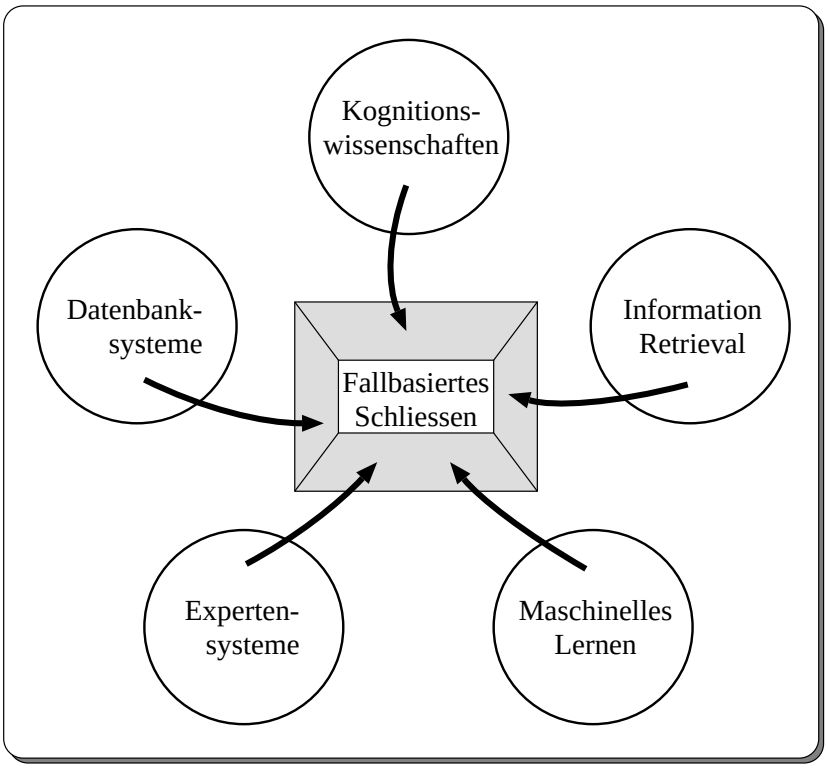


Abbildung 2.1: Einordnung des fallbasiertes Schließens

Das Verhalten von Experten bei der Lösung von Problemen ist sehr stark auch von den Erfahrungen geprägt, die sie bei der Bearbeitung früherer Probleme gewinnen konnten. Das Ziel sind somit wissensbasierte Informationssysteme, in denen das repräsentierte Wissen um die Erfahrungskomponente erweitert werden kann. Es hat sich gezeigt, daß dieses Erfahrungswissen durch die herkömmlichen logikorientierten Schemata wie Regeln, Klauseln etc. nicht ohne Verlust dargestellt werden kann.

Mit Hilfe des fallbasierten Schließens soll es Expertensystemen nun ermöglicht werden, frühere Fälle zur Lösung neuer Aufgaben heranzuziehen und aus gemachten Erfahrungen für zukünftige Situationen zu lernen (vgl. z.B. [AW92]).

Der Einsatz fallbasierter Systeme bietet sich vor allem dort an, wo Wissen bereits in Form von Fallbeispielen vorliegt. Dies ist insbesondere in Klassifikations– und Diagnoseanwendungen gegeben (vgl. z.B. [AKM+89], [AMTW91], [AW91] und [Wes91]). Daneben werden auch Einsatzmöglichkeiten fallbasierter Techniken in Entscheidungs– und Planungssystemen untersucht.

Bekannte Anwendungsgebiete des fallbasierten Schließens sind z.B. die Rechtsprechnug, die Medizin, Wirtschaftswissenschaften, die Architektur und auch die Mathematik.

### CBR und Maschinelles Lernen

Aus der Sicht des Maschinellen Lernens stellt fallbasiertes Schließen eine Art induktives Inferenzverfahren dar, durch das Folgen von Hypothesen erzeugt werden. Es kann als Spezialisierung des analogen Schließens aufgefaßt werden. Der Unterschied zwischen beiden Ansätzen besteht hauptsächlich darin, daß fallbasiertes Schließen i.a. versucht, Probleme *einer* Domäne durch Heranziehen ähnlicher Beispiele aus dieser Domäne zu lösen, während beim analogen Schließen existierendes Wissen aus einer bekannten in eine andere, noch unbekannte Domäne transferiert wird.

Fallbasierte Lernverfahren schließen wie auch induktive Verfahren von Beispielen auf globale Zusammenhänge, wobei die erzeugten Hypothesen nicht notwendigerweise korrekt sind. Beim induktiven Lernen werden vorwiegend logische Begriffsbeschreibungen gelernt, beim fallbasierten Lernen vor allem analytische. Im Gegensatz zu induktiven Lernverfahren, die innerhalb eines Lernschrittes meist mehrere Fälle vergleichen, ist fallbasiertes Lernen jedoch grundsätzlich inkrementell, da hier stets nur aus dem Vergleich zweier Fälle gelernt wird.

Ein zentraler Punkt des fallbasierten Schließens ist die Suche und Bereitstellung von Fällen aus der Fallbasis, die dem vorliegenden Problem am ähnlichsten sind. Ziel des fallbasierten Lernens ist es somit, die Fallbasis in der Trainingsphase so zu modifizieren und zu strukturieren, daß diese Fälle möglichst schnell gefunden werden können.

Vom Standpunkt des Maschinellen Lernens aus kann man dies mit dem Problem des begrifflichen Gruppierens (*concept clustering*) vergleichen. Die Fälle sind dabei so in konzeptuelle Klassen einzuteilen, daß jede Klasse eine Menge von Fällen enthält, die zueinander „am ähnlichsten“ sind, d.h. ähnlicher als zu Fällen anderer Klassen. Es handelt sich hier um *Lernen durch Beobachten*, bei dem die Klassen und ihre deskriptiven Beschreibungen erst entdeckt werden müssen (im Gegensatz zum Lernen aus Beispielen, bei dem ein Lehrer die Objekte in bekannte Klassen einteilt). Zur Bestimmung der Klassen und ihrer zugehörigen Konzeptbeschreibungen wird eine Evaluierungsfunktion (z.B. in Form von Ähnlichkeits– oder Distanzmaßen) verwendet.

Bekannte Conceptual–Clustering–Systeme sind z.B. UNIMEM [Leb86] und COBWEB [Fis87]. Das COBWEB–System baut aus den zu klassifizierenden Objekten einen hierarchischen Klassifikationsbaum auf, so daß das Lernen der Konzepte hier mit einer baumorientierten Suche verknüpft ist.

Auf der Grundlage einer solchen konzeptuellen Organisation der Fallbasis würde sich das ähnlichkeitsbasierte Fallretrieval auf die Aufgabe reduzieren, zu einem gegebenen Problemfall dessen Klasse beispielsweise über ein Ähnlichkeitsmaß zu bestimmen und die Fälle dieser Klasse auszugeben.

### CBR und Information Retrieval

Das Retrieval von Fällen aus der Fallbasis ist ein grundlegender Schritt für den Prozeß des fallbasierten Schließens. Hier müssen die für den Problemlösungsprozeß relevanten Fälle bereitgestellt werden. Bei der Realisierung fallbasierter Systeme spielt zudem die Effizienz des Fallretrievals eine sehr große Rolle.

Das Gebiet des Information Retrieval bietet hierfür einige interessante Ansätze. So kann das Problem, zu einem vorgegebenen Problemfall die ähnlichsten Fälle zu finden, als ein Partial–Matching–Problem angesehen werden, bei dem die Problembeschreibung mit den bekannten Fällen verglichen werden muß. Diese Fragestellung ist unter den Begriffen *assoziatives Retrieval* und *Nearest–Neighbor–Classification* Gegenstand intensiver Forschungsaktivitäten innerhalb des Information Retrieval. Dies beinhaltet insbesondere die Suche nach geeigneten Datenstrukturen und Algorithmen zur effizienten Lösung dieser Probleme.

In dieser Diplomarbeit untersuchen wir diesbezügliche Ansätze zur Unterstützung der assoziativen (bzw. mehrdimensionalen) Suche.

### CBR und Datenbanksysteme

Die Realisierung fallbasierter Systeme für komplexe Anwendungsbereiche erfordert immer mehr die Verknüpfung zweier Teilgebiete in der Informatik, die bislang meist getrennte Wege gingen: Wissensbasierte Systeme (*Knowledge Based Systems*, KBS) aus dem Bereich der Künstlichen Intelligenz und Datenbanksysteme (DBS).

Auf der abstrakten Ebene der Datenhaltung und des Datenzugriffs stellen sich in beiden Gebieten vergleichbare Probleme (siehe Abb. 2.2). In beiden Bereichen sind große Mengen an Informationen zu repräsentieren und zu verarbeiten. Bei KBS spricht man in diesem Zusammenhang

von *Wissensbasen*, in DBS von *Datenbanken*. Die in beiden Gebieten gesetzten Schwerpunkte unterscheiden sich jedoch wesentlich voneinander.

Während in DBS bisher hauptsächlich Probleme der Datensicherheit und der Effizienz im Vordergrund standen, versuchte man in KBS vorrangig, „Wissen“ geeignet darzustellen und zu verwenden.

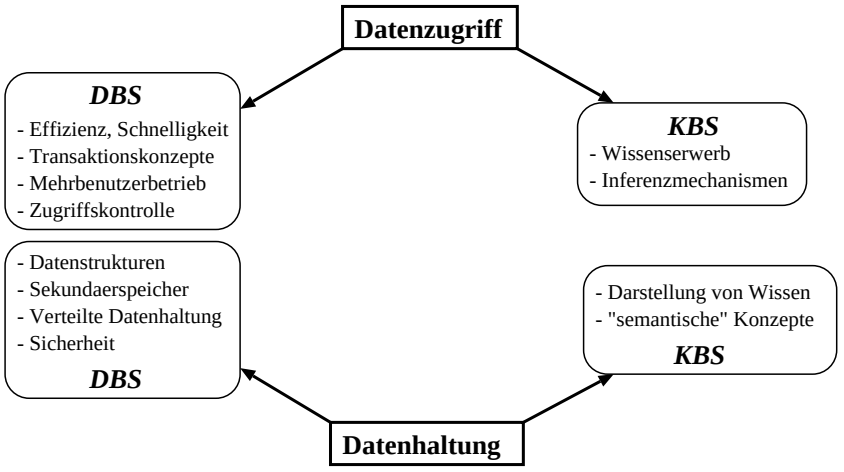


Abbildung 2.2: Vergleich der Bereiche DBS und KBS

In den letzten Jahren wurden jedoch in immer stärkeren Maße Ansätze untersucht, in denen Methoden und Konzepte beider Teilgebiete miteinander verknüpft werden (siehe z.B. [BM86]). Nonstandard–Datenbanksysteme (NDBS) sind hierfür ein Beispiel aus dem Bereich der DBS. Sie versuchen z.B., semantische Konzepte in Datenmodellen so zu integrieren, daß „Wissen“ repräsentiert werden kann.

Andererseits eröffnet die Entwicklung neuer Technologien (Client–Server–Konzepte, objektorientierte Datenbanken) im Bereich der DBS z.B. auch die Möglichkeit, Anwendungen wissensbasierter (und insbesondere auch fallbasierter) Systeme aufgesetzt auf Datenbanksysteme zu realisieren.

Bei der Verknüpfung von CBR und DBS stellt sich die Frage, welche Konzepte und Techniken aus dem Datenbank–Bereich in einer sinnvollen Art und Weise auf fallbasierte Systeme übertragbar sind. Hierbei gilt es zu unterscheiden zwischen Konzepten, die in fallbasierten Systemen

- *ohne Änderungen* übernommen werden können (z.B. verteilte Datenhaltung, Sekundärspeicherverwaltung, Mehrbenutzerbetrieb, Datensicherheit)
- *nicht* oder nur sehr schwer übertragbar sind (z.B. relationale Datenmodelle)
- nur *partiell* anwendbar sind und somit modifiziert werden müssen
  (z.B. Distanzmaße → Ähnlichkeitsmaße, Speicherungsstrukturen → „Wissens“strukturen, Datenmodellierung → Wissensmodellierung)

Wir untersuchen in dieser Diplomarbeit die Verwendung von Techniken des CBR in Verbindung mit einem objektorientierten DBS. Die Realisierung der Fallbasis erfolgt dabei auf der Datenbank. Für das ähnlichkeitsbasierte Fallretrieval sind dann z.B. die in DBS gebräuchlichen Zugriffsstrukturen so zu modifizieren, daß sie auch „fallbasiertes Wissen“ repräsentieren können. Die folgende Tabelle zeigt die für unseren Ansatz wichtigsten Verbindungspunkte zwischen den beiden unterschiedlichen Terminologien des CBR und der DBS:

| Fallbasiertes Schließen | Datenbanksysteme |
|---|---|
| Fallbasis | Datenbank |
| Fall | Entity/Objekt |
| „Wissens“strukturen | Zugriffsstrukturen |
| ähnlichkeitsbasiertes Fallretrieval | Best–Match–Suche |
| gegebener Problemfall | DB–Anfrage (Query) |

**Warum objektorientierte Datenbanken?**

Bei der Realisierung neuerer komplexer Anwendungen (z.B. CAD/CAM, VLSI, wissensbasierte Systeme), für die die konventionellen DBS (hierarchische, netzwerkorientierte oder relationale DBS) als nicht mehr ausreichend angesehen werden, spielen objektorientierte Datenbanken eine immer größere Rolle. Auch die Implementierung des in dieser Arbeit vorgestellten Ansatzes erfolgte auf einem solchen System.

Die konventionellen DBS sind nur für eine begrenzte Klasse von Anwendungen geeignet, die dadurch charakterisiert sind, daß zwar große Datenmengen, aber nur relativ einfache Datenmanipulations- und Retrievaloperationen auftreten ([Ull88]). Dabei ist die Effizienz des Datenzugriffs ein vorrangiges Ziel (vgl. auch vorangegangenen Abschnitt). Aus diesem Grund besteht in konventionellen DBS auch die Trennung zwischen einer Datendefinitionssprache (DDL), einer Datenmanipulationssprache (DML) mit effizienten DB–Zugriffsoperatoren aber sehr beschränkter Ausdruckskraft, und zwischen einer sog. Host–Sprache, die zur Erstellung von Anwendungsprogrammen alle üblichen Operationen bereitstellt, die sich nicht auf die DB beziehen.

Konventionelle DBS stellen zudem nur eine endliche Menge von Datentypen und Operatoren (insbesondere auch Zugriffsmechanismen) zur Verfügung. Darüber hinaus besteht oft der Zwang zu einer normalisierten Datenmodellierung. Dies bedeutet einen erheblichen Nachteil für komplexe Anwendungen, die neben schnellem Retrieval und effizienter Datenmanipulation noch durch folgende Anforderungen charakterisiert sind ([Ull88]):

- Bereitstellung *mächtigerer Operationen* auf den Daten (z.B. die Berechnung transitiver Hüllen von Relationen und Rekursion, die in konventionellen DBS nicht möglich sind)
- Unterstützung *semantischer Konzepte* (Aggregation, Generalisierung)
- Modellierung *komplexer Objekte* aus der realen Welt

Diese Probleme sind mit den alten Paradigmen nicht oder nur sehr schwer lösbar. Deshalb wurde ein Ansatz entwickelt, der die schnellen Zugriffsmechanismen der DML und die Funktionalität der Host–Sprache miteinander kombiniert: der *objektorientierte* Ansatz.

Hierbei wird eine Sprache bereitgestellt, die die Definition abstrakter Datentypen oder Klassen erlaubt. Dadurch kann sich der Benutzer die Datentypen, die er benötigt, und auch die Zugriffsstrukturen für diese Typen selbst definieren.

*Objektorientierte DBS* (ooDBS) verknüpfen die Konzepte des objektorientierten Sprachansatzes mit der Funktionalität konventioneller DBS. Dadurch bieten ooDBS folgende Möglichkeiten:

1. Darstellung *komplexer Objekte* und Strukturen durch benutzerdefinierte Datentypen mit hierarchischen Strukturen (Objekthierarchien)

2. *Kapselungsprinzip*: Definition von Operationen auf den Datenstrukturen

3. *Identität von Objekten*: Unterscheidung zwischen zwei verschiedenen Objekten, obwohl sie „gleich aussehen“, d.h., die Existenz eines Objektes ist unabhängig von seinem Wert

Der 1. und der 2. Punkt entsprechen der Definition *abstrakter Datentypen*. Das Prinzip der Objektidentität wurde bereits in hierarchischen und netzwerkorientierten DBS unterstützt, bevor es in den *wertorientierten* relationalen DBS dann aufgegeben wurde.

# Teil II

# Grundlagen

# Kapitel 3

# Der Prozeß des fallbasierten Schließens

## 3.1 Ein Modell

In fallbasierten Systemen stehen die Erfahrungen in Form von *Fallbeispielen* zur Verfügung. Das gesamte Erfahrungswissen, die *Fallbasis*, setzt sich aus einer Menge von bekannten Fällen zusammen. Ein Fall besteht dabei mindestens aus einer Problembeschreibung, einer Lösung sowie gegebenenfalls einer Rechtfertigung für diese Lösung. Da die Repräsentation von Fällen in hohem Maße vom jeweiligen Anwendungsbereich abhängig sind und deshalb unterschiedliche Auffassungen darüber bestehen, wie ein Fall formal zu beschreiben ist, soll der Begriff des Falles hier nur sehr allgemein definiert werden (vgl. dazu auch [AWB+92]).

**Definition (Fall):** Ein *Fall* ist ein Tripel $(P, R, L)$ mit einer Problembeschreibung $P$, einer Rechtfertigung $R$ und einer Lösung $L$.

Beim fallbasierten Schließen wird ein neues Problem dadurch gelöst, daß Lösungen von bekannten Fällen, die eine gewisse Ähnlichkeit zur aktuellen Problemsituation aufweisen, bezüglich den gegebenen Anforderungen interpretiert, modifiziert und analog auf die neue Aufgabe übertragen werden. Neue Problemlösungen können sich dabei auch aus Teillösungen mehrerer bereits bekannter Fälle zusammensetzen.

Gelöste Probleme werden als neu inferiertes Wissen in die Fallbasis übernommen und stehen somit für zukünftige Aufgabenstellungen zur Verfügung. Dieser Schritt entspricht dem Vorgang des Lernens.

Für das fallbasierte Schließen wurden eine Reihe verschiedener Prozeßmodelle aufgestellt. Das im folgenden vorgestellte Modell von Rissland et al. [RKW89] beschreibt die grundlegenden Schritte des fallbasierten Schließens. Dabei wird auch eine explizite Lernphase berücksichtigt.

Die Vorgehensweise des fallbasierten Schließens läßt sich nach Rissland in folgende fundamentale Schritte unterteilen:

1. **Vorauswahl und Bereitstellung relevanter Fälle:** Aus der gesamten Fallbasis wird zunächst eine handhabbare Menge von Fällen ausgewählt und als Grundlage für die weiteren Schritte des fallbasierten Schließens bereitgestellt. Die Relevanz der untersuchten Fälle ist dadurch charakterisiert, daß ihre Lösungen möglicherweise zur Lösung des gegebenen Problems herangezogen werden können. Diese Auswahl kann z.B. durch Vergleich der aktuellen Problembeschreibung mit den Beschreibungen der bekannten Fälle erfolgen. In vielen Systemen werden diese Vergleiche bezüglich bestimmter Falleigenschaften, die sich in der Vergangenheit für das Problemlösen als relevant erwiesen haben, durchgeführt.
   Ziel dieses initialen Schrittes ist eine Einschränkung der in den folgenden komplexeren Schritten zu betrachtenden Fallmenge.

2. **Auswahl der am besten geeigneten Fälle:** Aus der im ersten Schritt gewonnenen Menge potentieller Lösungskandidaten sollen nun einige wenige Fälle selektiert werden, die hinsichtlich der speziellen gegebenen Problemsituation als am besten geeignet erscheinen. Hierbei gehen vor allem Betrachtungen darüber mit ein, inwieweit ein Fall im weiteren Verlauf die Lösungskonstruktion unterstützen kann. Üblicherweise werden dazu Ranking–Methoden herangezogen, die z.B. Ähnlichkeitsmaße oder Parameter–Gewichtungen verwenden.

3. **Konstruktion einer Lösung:** In diesem Schritt wird durch eine Anpassung der Lösungen der ausgewählten Fälle eine neue Lösung für den aktuellen Problemfall konstruiert. Die Methoden, die bei der Anpassung an die gegebene Situation angewendet werden, sind ebenfalls sehr anwendungsabhängig, so daß man keine allgemeine Vorgehensweise angeben kann.

4. **Testen und kritisieren der gefundenen Lösung:** Die konstruierte Lösung wird vom System selbst überprüft. Dies kann auf mehrere Arten geschehen. Eine Möglichkeit ist z.B. das Überprüfen von Gegenbeispielen, um die Lösung auf ihre Robustheit hin zu testen. Oder es erfolgt eine Simulation der Lösung, wobei anschließend die dabei aufgetretenen Ergebnisse mit den zu erwartenden Ergebnissen verglichen werden. Des weiteren kann man noch die Erfüllung vorgegebener Randbedingungen überprüfen.

5. **Auswertung der Ergebnisse:** Da die durch den Problemlösungsprozeß erhaltenen Ergebnisse ähnlich wie bei induktiven Schlüssen nicht notwendigerweise korrekt sind, ist noch ihre Überprüfung in der realen Welt erforderlich. Das System erhält vom Benutzer ein positives oder negatives Feedback über die Ergebnisse, die bei der Anwendung der gefundenen Lösung in der Realität aufgetreten sind. Über das Feedback soll vor allem verhindert werden, daß sich in der Vergangenheit gemachte Fehler wiederholen.

6. **Übernahme des neuen Falles in die Fallbasis (Lernphase):** In diesem letzten Schritt wird die gegebene Problemsituation zusammen mit der gefundenen Lösung als neuer Fall in die Fallbasis eingebracht. Damit steht er für eine Wiederbenutzung bei zukünftigem Schließen zur Verfügung. Zu diesem Zweck muß bei der Übernahme des neuen Falles in den Fallspeicher die interne Organisationsstruktur, die den Prozeß des Schließens unterstützen soll, entsprechend geändert und angepaßt werden.
   Die Anpassung erfolgt dabei unter Berücksichtigung des analysierten Feedbacks aus dem 5. Schritt. Bei negativem Feedback soll eine zukünftige Wiederholung des Fehlers vermieden werden, während eine richtige Lösung als positiv zu kennzeichnen ist. In symbolischen Ansätzen führt dies zu einer Modifikation der internen Speicherungsstruktur, bei statistischen Ansätzen zu entsprechenden Verstärkungen bzw. Abschwächungen der jeweiligen Parameter.
   Durch den Lernprozeß werden die Fälle selbst unverändert in die Fallbasis übernommen. Darin unterscheidet sich fallbasiertes Lernen von den meisten induktiven Lernverfahren, die die Beispiele, die sie zur Bildung einer Hypothese herangezogen haben, anschließend wieder vergessen.

Zusammenfassend lassen sich also im wesentlichen folgende Grundprobleme des fallbasierten Schließens charakterisieren:

1. die geeignete Auswahl von Organisationsstrukturen für die Fallbasis, um eine schnelle Bereitstellung der relevanten Fälle zu gewährleisten
2. die Erstellung entsprechend effizienter Suchalgorithmen auf diesen Strukturen
3. Transfer und Anpassung der Lösungen bekannter Fälle an die neue Situation

Das Problem des analogen Lösungstransfers ist dabei ein sehr zentraler Punkt. Es müssen hier insbesondere Abbildungen gefunden werden, die zum einen die Zusammenhänge zwischen zwei Problemsituationen beschreiben und die zum anderen dann auch zur Modifikation der bekannten Lösungen herangezogen werden.

In dieser Diplomarbeit liegt das Hauptaugenmerk jedoch vor allem auf den beiden ersten Aspekten, die die Organisation der Fallbasis sowie geeignete Suchalgorithmen betreffen.

## 3.2 Retrieval von Fällen aus der Fallbasis

Ein sehr wichtiger Schritt des fallbasierten Schließens ist in dem oben beschriebenen Prozeßmodell von Rissland die Vorauswahl von möglicherweise relevanten Fällen aus der Fallbasis. Als Grundlage für die weiteren Prozeßschritte haben diese ausgewählten Fälle großen Einfluß auf die Anpassungs– und Auswertungsprozesse des fallbasierten Schließens.

Das Retrieval kann als ein Best–Match–Problem angesehen werden, bei dem Beschreibungen bekannter Fälle gegen eine vorliegende Situationsbeschreibung zu matchen sind. Es sollen die Fälle zurückgeliefert werden, die zu der gegebenen Problemsituation die besten Matches liefern. Die Best–Match–Suche an sich ist ein Problem von nicht–polynomialer Komplexität, so daß sich bei umfangreichen Fallbasen vor allem die Forderung nach effizienten Retrievalmechanismen stellt, um möglichst nur die in Frage kommenden Fälle auf ihre Relevanz hin zu untersuchen und ein vollständiges Durchsuchen der gesamten Fallbasis zu vermeiden.

Dabei spielt die Organisation der Fallbasis eine entscheidende Rolle. Hier müssen Speicherungsstrukturen bereitgestellt werden, die durch entsprechende Suchalgorithmen einen schnellen Zugriff auf die Fälle unterstützen.

### 3.2.1 Organisationsstrukturen für die Fallbasis

In vielen bekannten Systemen wird der Ansatz der *Indexierung* verfolgt, in dem Indexstrukturen für die gespeicherten Fälle aufgebaut werden. Üblicherweise sind dabei die Fälle durch sog. *Featurevektoren* beschrieben, deren Komponenten bestimmte Falleigenschaften repräsentieren. Die Indexstrukturen werden für eine Teilmenge dieser Eigenschaften aufgebaut. Hierbei stellt sich die Frage, welche der Features als wichtig und geeignet angesehen werden können. Lebowitz verwendet z.B. induktive Lernmethoden, die die Eigenschaften auswählen, die sich in der Vergangenheit bereits als relevant erwiesen haben [Leb87]. Ein Ansatz von Barletta und Mark hingegen bestimmt die Indizes mit Hilfe erklärungsbasierter Techniken [BM88].

Bei großen Fallbasen reicht die Indexierung alleine jedoch für ein effizientes und schnelles Retrieval nicht aus. Zusätzlich ist es oft erforderlich, die Fälle in Speicherungsstrukturen zu organisieren, auf denen schnelle Suchalgorithmen ablaufen können. Ansätze in dieser Richtung bieten beispielsweise die Discrimination Networks von Feigenbaum [Fei63] und darauf aufsetzende Erweiterungen. Neuere Ansätze basieren u.a. auf dem von Bentley entwickelten $k$-d-Baum [Ben75], einer mehrdimensionalen Zugriffspfadstruktur, die auch zentraler Gegenstand dieser Diplomarbeit ist, und auf konzeptuellem Retrieval in semantischen Netzwerken, in denen die Fälle in Konzepthierarchien angeordnet werden [Lev90].

Darüber hinaus müssen Techniken und Mechanismen untersucht werden, die solche Zugriffsstrukturen bei der Aufnahme neuer Fälle in die Fallbasis *dynamisch* reorganisieren und anpassen.

### 3.2.2 Repräsentation der Fallbeispiele

Zusammen mit dem Problem der Organisation der Fallbasis durch geeignete Speicherungsstrukturen stellt sich die Frage nach der Art und Weise, wie die Fallbeispiele selbst in der Fallbasis repräsentiert werden sollen.

Hier gibt es zum einen die Möglichkeit, die Fälle jeweils als ganzes an einer Stelle im Speicher abzulegen. Das ist dann von Vorteil, wenn einzelne Fälle eventuell schon nahezu komplette Lösungen für ein Problem anbieten. Allerdings können zur Lösung eines neuen Problems nur sehr schwer einzelne Teillösungen mehrerer bekannter Fälle verwendet werden.

Eine alternative Repräsentation zerlegt die Fälle in mehrere Teile und legt diese an verschiedenen Stellen im Speicher ab. Die Rekonstruktion eines kompletten Falles kann über eine Verzeigerung erfolgen, die zwischen den einzelnen Teilen besteht. In einer solchen Darstellung ist es leichter, auf bekannte Teillösungen zuzugreifen. Gerade bei komplexeren Problemen ist es oft von Vorteil, wenn eine Gesamtlösung aus mehreren Teillösungen verschiedener Fälle zusammengesetzt werden kann. Ein Nachteil liegt jedoch in dem Mehraufwand für die erforderlichen Rekonstruktionen der Fallbeispiele, wenn sie komplett zur Lösung eines neuen Problems verwendet werden sollen.

In dieser Diplomarbeit stellen wir die Fälle als ganze Objekte dar. Sie werden durch *Attribute* repräsentiert, die ähnlich den oben erwähnten Features bestimmte Eigenschaften der Fälle beschreiben. Jedes Repräsentationsattribut kann dabei einen *Wert* aus einem ihm zugeordneten Wertebereich annehmen.

Im folgenden wird nun in formaler Weise spezifiziert, wie solche aus Attribut–Wert–Paaren bestehenden Fallbeschreibungen aussehen.

**Definition (Fallrepräsentation):** Wird ein Fall durch $n$ Attribute beschrieben, so ist seine zugehörige *Fallrepräsentation FR* ein $n$-Tupel der Form

$$FR \ = (A_1, A_2, \ldots, A_n) \qquad \text{mit } A_i \neq A_j \text{ für } i \neq j.$$

Sei weiterhin $\mathcal{W}=\{W_1, \ldots, W_s\}$ $(s \leq n)$ die Menge aller Wertebereiche, die den $n$ Attributen zugeordnet sind. Für ein $W_i \in \mathcal{W}$ müssen folgende Bedingungen erfüllt sein:

1. auf den Elementen von $W_i$ ist eine 2-stellige Ordnungsrelation $R$ definiert, so daß für zwei Elemente $x, y \in W_i$ gilt: $(x, y) \in R \Leftrightarrow x <_R y$
2. auf $W_i$ ist ein Ähnlichkeitsmaß $\mu_i : W_i \times W_i \to [0, 1]$ definiert (der Begriff des Ähnlichkeitsmaßes wird in Kapitel 4, Abschnitt 4.2 noch definiert werden)

Sei die Fallbasis $\mathcal{FB}$ eine Menge solcher aus Attributen zusammengesetzter Fallbeschreibungen. Das $j$-te Attribut der Repräsentationen kann dann formal als eine Funktion aufgefaßt werden, die eine Fallbeschreibung auf einen der $s$ Wertebereiche abbildet:

$$A_j : \mathcal{FB} \to W_i \cup \{\perp\}, \qquad 1 \leq i \leq s,\ 1 \leq j \leq n.$$

$\perp$ bezeichne hierbei den undefinierten Wert. Sei $a_j^{(FR)}$ der Wert des $j$-ten Attributes in der Fallbeschreibung $FR$. Dann gilt

$$a_j^{(FR)} := \begin{cases} \perp & , \text{ falls für } FR \text{ im } j\text{-ten Attribut kein Wert spezifiziert ist} \\ A_j(FR) & , \text{ sonst.} \end{cases} \tag{3.1}$$

Die Werte, mit denen die Repräsentationsattribute belegt werden können, lassen sich einteilen in *einfache* und in *komplexe* Werte. (Im folgenden sind für die verschiedenen Arten von Attributwerten in Klammern jeweils noch Bezeichnungen angegeben, die in Kapitel 10 dann verwendet werden.)

**Einfache** Werte sind

- reelle Zahlen (*Number*)
  - ganze Zahlen (*Integer*)
  - rationale Zahlen (*Float*)
- boolesche Werte (*Boolean*)
- Zeichenketten (*String*)
- Symbole (*Symbol*).

Die **komplexen** Werte sind Mengen, deren Elemente wiederum einfache oder komplexe Werte sein können. Hierbei wird unterschieden zwischen *ungeordneten* und *geordneten* Mengen.

- eine ungeordnete Menge (*UnorderedSet*) ist eine Menge, auf deren Elementen keine Ordnungsrelation definiert ist
- für die Elemente einer geordneten Menge (*OrderedSet*) besteht eine Ordnungsrelation
  - Listen (*List*) sind besondere geordnete Mengen. Ihre Ordnung ist durch die Reihenfolge ihrer Elemente implizit vorgegeben. Ein Listenelement ist kleiner als alle Elemente, die in dieser Liste nach ihm enthalten sind.
  - Intervalle (*Interval*) schließlich sind spezielle Listen, die genau zwei Elemente enthalten, wobei das erste Element die Intervalluntergrenze und das zweite die Intervallobergrenze angibt. Ein Intervall stellt somit implizit eine Liste dar, die alle Elemente zwischen den beiden Intervallgrenzen enthält.

# Kapitel 4

# Kognitionswissenschaftliche Grundlagen des fallbasierten Schließens

## 4.1 Das MAC/FAC–Modell

Das MAC/FAC–Modell ("*m*any *a*re *c*alled but *f*ew *a*re *c*hosen") ist ein von D. Gentner und K.D. Forbus entwickeltes Modell für das ähnlichkeitsbasierte Retrieval [GF91]. Es versucht, den Prozeß des ähnlichkeitsbasierten Erinnerns unter Berücksichtigung der bei Menschen in diesem Zusammenhang beobachteten psychologischen Phänomene und Verhaltensweisen zu modellieren. Als Grundlage dienen hierbei kognitionspsychologische Erfahrungen, die in diesbezüglichen Experimenten mit Menschen gemacht werden konnten.

### 4.1.1 Kognitionspsychologische Aspekte

Ähnlichkeitsbasiertes Erinnern kann auch aus psychologischer Sicht in mehrere Teilprozesse aufgegliedert werden. Die Lösung einer gegebenen Problemsituation durch Erfahrungen, die in ähnlichen Situationen bereits gemacht worden sind, beinhaltet folgende Schritte:

1. das Erinnern an eine ähnliche Situation im Langzeitgedächtnis
2. Bildung einer Mapping–Funktion zur Erfassung der Ähnlichkeitsbeziehungen zwischen der Problemsituation und der bekannten Situation
3. Auswertung der Mapping–Funktion und analoge Anpassung der bekannten Problemlösung an die neue Situation

Je nach Abstraktionsstufe der betrachteten Gemeinsamkeiten unterscheidet man verschiedene Arten von Matches zwischen zwei Situationen:

**Analogy Match:** basiert auf einer Menge gemeinsamer *struktureller* Beziehungen und Relationen hoher Abstraktionsstufen

**Literal Similarity Match:** berücksichtigt werden sowohl Gemeinsamkeiten struktureller Art als auch Ähnlichkeiten, die sich aufgrund einfacher Situationsbeschreibungen ergeben

**Surface Match:** berücksichtigt werden hauptsächlich Gemeinsamkeiten *syntaktischer* Art, die sich aus den Situationsbeschreibungen ableiten lassen; hier werden Ähnlichkeitsbeziehungen auf einer nur sehr niedrigen Abstraktionsstufe betrachtet (beispielsweise auf einer Menge von Eigenschaften, die die Situationen beschreiben)

Versuche aus der Psychologie haben gezeigt, daß bei Menschen der Prozeß des ähnlichkeitsbasierten Erinnerns an Situationen aus dem Langzeitgedächtnis in einem erheblichen Maße von Ähnlichkeitsbeziehungen auf niedrigeren Abstraktionsstufen beeinflußt wird. Hierbei spielen insbesondere die Surface Matches eine dominante Rolle.

Im Gegensatz dazu gibt es psychologische Erfahrungswerte, die belegen, daß Menschen andererseits sehr schnell strukturelle Ähnlichkeiten auf höheren Abstraktionsstufen zwischen zwei vorliegenden Situationen erkennen können. Darüber hinaus konnte festgestellt werden, daß sie sowohl zur Erkennung dieser Ähnlichkeiten als auch zur Beurteilung der Qualität eines Matches zwischen zwei Situationen vorrangig Relationen höherer Ordnungen heranziehen. Dagegen haben die Gemeinsamkeiten, die sich aufgrund der einfachen Situationsbeschreibungen ergeben, darauf weniger Einfluß.

Ein adäquates Modell für den komplexen Prozeß des ähnlichkeitsbasierten Retrievals muß deshalb auch die menschliche Wahrnehmungsfähigkeit bezüglich struktureller Ähnlichkeitsbeziehungen auf höheren Stufen berücksichtigen. Daraus ergibt sich zum einen die Forderung nach strukturellen Repräsentationen, die die Situationen bzw. Fälle auf höheren Abstraktionsebenen beschreiben können, und zum anderen nach Prozessen, die das Erkennen von Gemeinsamkeiten zwischen solchen Repräsentationen ermöglichen.

Der Implementierung eines solchen Modells sind jedoch Einschränkungen vorgegeben. Da ist zunächst die große Anzahl von Fällen, die in der Regel in der Fallbasis gespeichert werden sollen. Die Erinnerung an Situationen im Langzeitgedächtnis bei Menschen entspricht im Modell dann Zugriffen auf den Fallspeicher, wobei es durch die Schnelligkeit des menschlichen Erinnerungsvorganges naheliegt, diese Zugriffe durch einen schnellen Prozeß, d.h. einen Prozeß, der wenig Berechnungsaufwand erfordert, zu simulieren. Die anschließende Beurteilung und Auswertung der gefundenen Matches erfordert jedoch einen komplexeren und somit teureren Prozeß.

Das MAC/FAC–Modell verfolgt diese Strategie durch ein zweistufiges Vorgehen. In der ersten Stufe, der *MAC*–Stufe, die dem Erinnerungsprozeß entspricht, werden aus der Menge aller gespeicherten Fälle die potentiellen Lösungskandidaten mittels eines „billigen“, schnellen Matchers ermittelt, der die Ähnlichkeiten auf der untersten Abstraktionsebene syntaktischer Fallbeschreibungen betrachtet. Die zweite Stufe (*FAC*) wählt aus den in der MAC–Stufe gewonnenen Fällen den besten Lösungskandidaten aus, indem komplexere Matches gebildet werden, die strukturelle Gemeinsamkeiten von Fallrepräsentationen auf höheren Abstraktionsebenen berücksichtigen.

### 4.1.2 Das Modell

Abbildung 4.1 zeigt die Komponenten der beiden Stufen des MAC/FAC–Modells. Als Eingabe dienen dabei eine Menge von Situations– bzw. Fallbeschreibungen aus dem Fallspeicher sowie eine Problembeschreibung, für die ein Match gefunden werden soll. Ausgegeben wird die Beschreibung des Falles aus dem Speicher, der den besten Match zum gegebenen Problem liefert.

Beide Stufen enthalten je einen *Matcher*, der die eingegebenen Fallbeschreibungen mit dem vorliegenden Problem vergleicht, und einen *Selektor*, der anhand der Ergebnisse des Matchers eine entsprechende Auswahl aus den Eingabebeschreibungen trifft.

Die Eingabe der *FAC*–Stufe besteht aus einer kleinen Menge ausgewählter Beschreibungen von bekannten Fällen aus dem Speicher. Der komplexe Matcher berechnet für jede dieser Beschreibungen einen Wert für die Qualität des Matches mit der gegebenen Problembeschreibung. Hier müssen Ähnlichkeiten auf der strukturellen Ebene ausgewertet werden, d.h. Gemeinsamkeiten auf höheren Abstraktionsstufen (*literal similarity*). Da die Berechnung dieses Matches relativ aufwendig ist, ist sie effizient nur auf eine kleine Menge von Beschreibungen anwendbar.

Genau diese kleine Menge an Kandidaten soll die *MAC*–Stufe liefern. Die auf dieser Stufe betrachtete Menge von Fällen ist in der Regel sehr groß, so daß hier ein extrem schneller Matcher erforderlich ist, der zunächst nur Ähnlichkeiten auf der syntaktischen Ebene der Fallbeschreibungen berücksichtigt (*surface similarity*).

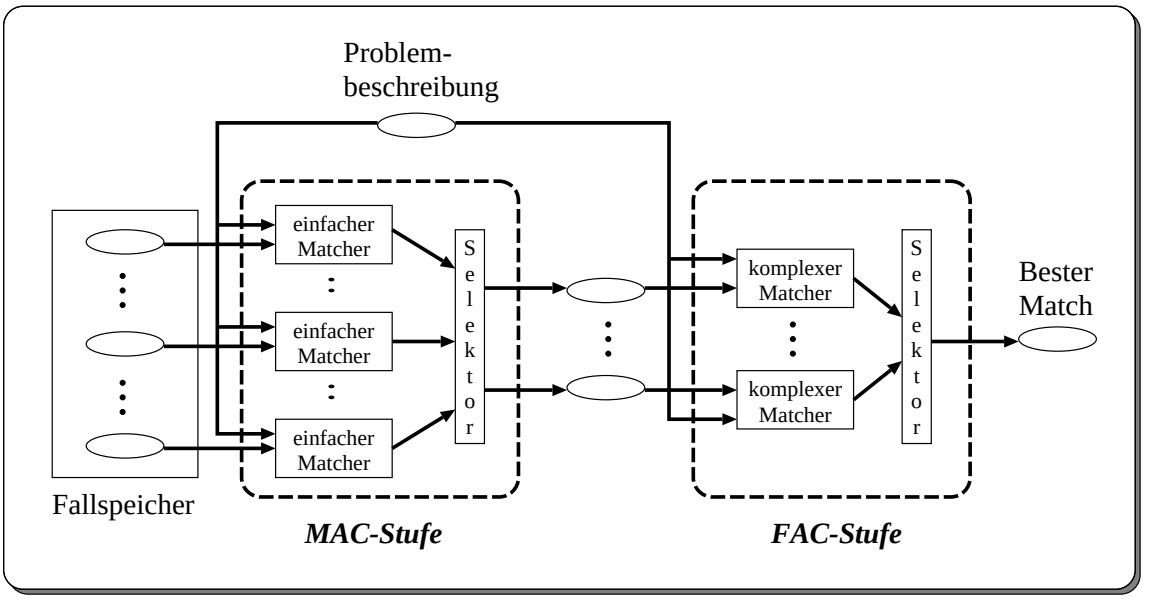


Abbildung 4.1: Das MAC/FAC–Modell

### 4.1.3 Einordnung der Arbeit

Geht man von dem in Kapitel 3 beschriebenen Prozeßmodell für das fallbasierte Schließen aus, so läßt sich sehr schnell ein Bezug dieses Modells zu MAC/FAC herstellen. Die MAC–Stufe unterstüzt ja gerade den initialen Schritt des fallbasierten Schließens, nämlich die Bereitstellung einer Vorauswahl potentiell relevanter Fälle aus der Fallbasis, während die FAC–Stufe die weiteren Schritte der Auswahl, Anpassung und Interpretation des am besten geeigneten Falles umfaßt.

Da sich diese Diplomarbeit insbesondere mit dem initialen Prozeßschritt befaßt, kann sie in die erste Stufe des MAC/FAC–Modells eingeordnet werden.

In dieser Arbeit gehen wir von einer Fallbasis aus, die eine (große) Menge bekannter Fälle enthält. Die Fälle werden durch Attribute dargestellt, die die syntaktischen Eigenschaften beschreiben, die bzgl. der Ähnlichkeitsbeziehungen zwischen Fällen eine Rolle spielen. Der Fallspeicher enthält also Repräsentationen einer niedrigen Abstraktionsstufe.

Das Ziel ist es nun, aufgrund dieser Fallbeschreibungen zu einem vorgegebenen Fall schnell und effizient die ähnlichsten Fälle zu bestimmen und für die weitere Verarbeitung zur Verfügung zu stellen. Aus Sicht des MAC/FAC–Modells heißt das nichts anderes, als daß ein einfacher Matcher bereitzustellen ist, der diese Auswahl treffen kann.

Zur Unterstützung des fallbasierten Schließens sind noch weitere Probleme zu beachten. Zunächst muß eine geeignete Fallrepräsentation gefunden werden, die den Anforderungen genügt, wie sie sich durch das MAC/FAC–Modell ergeben; d.h., die Ähnlichkeitsbeziehungen auf dieser Abstraktionsstufe müssen aussagekräftig genug sein im Hinblick auf das letztendliche Ziel der jeweiligen Anwendung, den in *struktureller* Hinsicht ähnlichsten Fall zu finden. Dazu muß für jede spezielle Anwendung ein entsprechender Ähnlichkeitsbegriff geeignet spezifiziert werden. Des weiteren ist ein effizienter Mechanismus erforderlich, der eine schnelle Bestimmung der besten Matches und somit der ähnlichsten Fälle ermöglicht.

In dieser Diplomarbeit wurde die Fallbasis im Rahmen eines objektorientierten Datenbanksystems implementiert. Die Auswahl der Fälle erfolgt über Datenbank–Anfragen, in denen die Problemfälle beschrieben werden. Somit besteht die Aufgabe darin, zur Unterstützung des ähn-

lichkeitsbasierten Retrievals das DB–System um geeignete Datenstrukturen zur Organisation der Fallbasis und um effiziente Such– und Matching–Algorithmen zu erweitern.

## 4.2 Der Ähnlichkeitsbegriff

Die Bestimmung der Relevanz bekannter Fälle für die Lösung eines gegebenen Problems ist eng verknüpft mit dem Begriff der *Ähnlichkeit*, der bei Analogieschlüssen eine zentrale Rolle einnimmt. Deshalb soll hier zunächst der Ähnlichkeitsbegriff präzisiert und formalisiert werden. Ähnlichkeit zwischen Fällen hängt im wesentlichen jedoch vom jeweiligen Verwendungszweck der Fälle ab, der sich aus einer bestimmten Anwendung heraus ergibt. Das heißt, daß zwischen Problemstellungen immer nur eine Ähnlichkeit *bezüglich verwendungsabhängiger Aspekte* besteht und man deshalb nicht von einer absoluten Ähnlichkeit sprechen kann.

Ähnlichkeit wird zumeist durch die Verwendung von Ähnlichkeitsmaßen oder den dazu dualen Distanzfunktionen modelliert. Dabei ergibt sich das Problem, eine „gute“, d.h. für den bestimmten Anwendungszweck geeignete Ähnlichkeit zu definieren. Da diese Fragestellung im allgemeinen nicht in einem einzigen Schritt gelöst werden kann, entspricht die Suche nach einem guten Ähnlichkeitsmaß ebenfalls einem Lernprozeß ([Ric92]).

Die im folgenden aufgeführte Formalisierung der Begriffe des *Ähnlichkeitsmaßes* und der *Distanzfunktion* basiert auf einer Arbeit von Richter und Wess [RW91]. Dabei sei $\mathcal{R}$ die Menge der reellen Zahlen.

**Definition (Ähnlichkeitsmaß):** Ein *Ähnlichkeitsmaß* auf einer Menge M ist eine reellwertige Funktion $\mu$: $\mathrm{M}^2 \to [0,1]$ mit

1. $\mu(x,x) = 1$ (Reflexivität)
2. $\mu(x,y) = \mu(y,x)$ (Symmetrie)

Dual dazu können *Unähnlichkeiten* zwischen Fällen in Form von Distanzfunktionen modelliert werden.

**Definition (Distanzfunktion):** Eine *Distanzfunktion* auf einer Menge M ist eine nicht–negative Funktion $d$: $\mathrm{M}^2 \to \mathcal{R}$ mit

1. $d(x,x) = 0$ (Reflexivität)
2. $d(x,y) = d(y,x)$ (Symmetrie)
3. $d(x,y) \geq 0$

Um zu einem gegebenen Fall $x$ einen „ähnlichsten Fall $y$“ zu finden, wird beim analogen Schließen häufig die vierstellige Relation $R(x,y,u,v)$ benutzt, die die Aussage “$x$ ist ähnlicher zu $y$ als $u$ zu $v$“ ausdrücken soll. Diese Relation kann sowohl über eine Distanzfunktion $d$ als auch über ein Ähnlichkeitsmaß $\mu$ definiert werden:

$$R_d(x,y,u,v) \quad :\Longleftrightarrow \quad d(x,y) \leq d(u,v) \tag{4.1}$$

$$R_\mu(x,y,u,v) \quad :\Longleftrightarrow \quad \mu(x,y) \geq \mu(u,v) \tag{4.2}$$

Man sagt dann, eine Distanzfunktion $d$ und ein Ähnlichkeitsmaß $\mu$ sind *kompatibel*, falls

$$R_d(x,y,u,v) \Longleftrightarrow R_\mu(x,y,u,v) \tag{4.3}$$

Es gilt, daß $d$ und $\mu$ kompatibel sind, wenn es eine bijektive ordnungsinvertierende Abbildung $f$:Wertebereich($d$) $\to$ Wertebereich($\mu$) gibt, so daß $f(0) = 1$ und $f(d(x,y)) = \mu(x,y)$.

Solche Abbildungen können auch dazu benutzt werden, zu einem gegebenen $d$ bzw. $\mu$ ein kompatibles Gegenstück zu bestimmen. Bekannte Funktionen sind $f(x) = 1 - \frac{x}{x+1}$ für einen unbeschränkten Wertebereich von $d$ oder $f(x) = 1 - \frac{x}{max}$, falls der Wertebereich von $d$ ein größtes Element $max$ besitzt.

Mit Hilfe der Relation $R$ kann man nun die Aussage "$y$ ist ähnlicher zu $x$ als $z$" folgendermaßen ausdrücken:

$$S_d(x, y, z) \quad :\Longleftrightarrow \quad R_d(x, y, x, z) \tag{4.4}$$

$$S_\mu(x, y, z) \quad :\Longleftrightarrow \quad R_\mu(x, y, x, z) \tag{4.5}$$

Für einen zu $x$ ähnlichsten Fall $y$ aus einer Menge M gilt also somit $S_d(x, y, z)$ bzw. $S_\mu(x, y, z)$ für alle $z$ aus M.

Üblicherweise werden die Ähnlichkeitsmaße bzw. die Distanzfunktionen auf Fälle angewendet, die durch Attribute beschrieben und deren Repräsentationen durch geordnete Tupel fester Länge gegeben sind, wobei jede Tupelkomponente einem Attribut entspricht und den entsprechenden Wert enthält.

## 4.3 Bewertung der Ähnlichkeit und Eignung bekannter Fälle

Damit der Prozeß des fallbasierten Schließens möglichst schnell ablaufen kann, kommt es vor allem darauf an, die „richtigen" Fälle aus der Fallbasis auszuwählen. Dies betrifft sowohl den initialen Schritt, in dem die Vorauswahl einer kleineren Menge potentiell relevanter Fälle getroffen wird, als auch die anschließende Bestimmung des zur Problemlösung am besten geeigneten Falles aus dieser Menge. In [Jan92] wird ein Modell beschrieben, das die Auswahl von Fällen unter diesen beiden Aspekten betrachtet.

Die Fallauswahl ist dabei ein Prozeß, der aus zwei Phasen besteht:

**1. Phase:** Durchsuchen der Fallbasis nach der Menge der ähnlichsten Fälle

**2. Phase:** Bewertung der ausgewählten Fälle hinsichtlich ihrer Eignung als Lösungskandidaten und Auswahl des am besten geeigneten Falles

Diese zwei Phasen sind vergleichbar mit den beiden Stufen des MAC/FAC–Modells, die diese Vorgehensweise ebenfalls modellieren. Hier wird die *such–orientierte Ähnlichkeitsbestimmung* mit einer anschließenden *wissensbasierten Bewertung der Brauchbarkeit bzw. Eignung* von Fällen verknüpft.

Die Motivation dieses Modells ergibt sich aus den möglichen Fehlern, die bei der Auswahl von Fällen aus einer Fallbasis auftreten können:

**Definition:**

**$\alpha$-Fehler:** Ein Fall, der zur Lösung des gegebenen Problems *geeignet* ist, wird *nicht* ausgewählt.

**$\beta$-Fehler:** Ein Fall, der zur Lösung des gegebenen Problems *nicht geeignet* ist, wird trotzdem *ausgewählt*.

Jedes Modell zur Beschreibung des Fallauswahl–Prozesses muß diese beiden Fehlerarten berücksichtigen.

Das im folgenden vorgestellte Modell wählt die Fälle zur Lösung eines Problems derart aus, daß in einer Vorauswahl (vergleichbar mit dem initialen Prozeßschritt) zunächst eine Teilmenge von Fällen bestimmt wird, die vom *statistischen* Standpunkt aus gesehen geeignete Lösungskandidaten sein könnten.

Danach erfolgt eine Bewertung jedes einzelnen der ausgewählten Kandidaten um zu überprüfen, ob ein auf das gegebene Problem am besten passender Fall darunter ist. Anhand dieser Bewertung wird dann auch entschieden, welche Strategie im weiteren Problemlösungsprozeß verfolgt wird: die des fallbasierten Schließens, die eine analoge Anpassung der Lösung eines bekannten Falles beinhaltet, oder die, die eine Lösung von Grund auf neu konstruiert. Die zweite Phase der Bewertung verwendet vor allem vorhandenes Domänenwissen.

### 4.3.1 Ähnlichkeitsbestimmung über das Contrast–Modell

Der folgende Ansatz zur Ähnlichkeitsbestimmung bei der Vorauswahl bekannter Fälle geht auf ein mengentheoretisches Ähnlichkeitsmodell von Tversky zurück, das sog. *Contrast–Modell* [Tve77].

Motiviert wurde dieses formale Modell durch das Ziel, Erkenntnisse, die in psychologischen Experimenten gewonnen werden konnten, zu formalisieren. Das Contrast–Modell stellt somit eine Alternative zu den geometrischen Ähnlichkeitsmodellen (z.B. das euklidische Modell) dar; diese gehen meist von Axiomen aus, die nicht durch experimentell gewonnene Erkenntnisse untermauert sind.

In dem Modell wird die Ähnlichkeit zweier Objekte durch eine lineare Kombination (Gegenüberstellung, Vergleich; engl. *contrast*) von gewichteten Differenzen ausgedrückt, die zwischen ihren gemeinsamen und ihren unterscheidenden Merkmalen bestehen.

Sei nun $\mathcal{B} = \{a, b, c, \ldots\}$ eine Menge von Objekten bzw. Fällen, zwischen denen Ähnlichkeiten bestimmt werden sollen. Jedes Element von $\mathcal{B}$ ist dabei durch eine Menge von Attributen repräsentiert, die das Objekt beschreiben. Die Objektbeschreibungen der Elemente $a$, $b$, $c$,... seien entsprechend mit $A$, $B$, $C$,... bezeichnet.

Sei weiterhin $\Phi$ die Menge aller Features, die in den Objekten aus $\mathcal{B}$ vorkommen. Dabei wird angenommen, daß $\Phi$ endlich ist.

Den Ähnlichkeitsberechnungen für zwei Fälle $a$ und $b$ aus $\mathcal{B}$ liegen in Tverskys Contrast–Modell drei gewichtete Attributmengen $X, Y, Z \subseteq \Phi$ zugrunde:

$$\begin{aligned} A \cap B &= \{x \mid x \in A \ \wedge \ x \in B\} = X \\ A \setminus B &= \{x \mid x \in A \ \wedge \ x \notin B\} = Y \\ B \setminus A &= \{x \mid x \notin A \ \wedge \ x \in B\} = Z \end{aligned}$$

Nach den Axiomen, die Tversky in [Tve77] spezifiziert hat, gibt es ein Ähnlichkeitsmaß $S$, so daß die Ähnlichkeit zwischen zwei Objekten $a$ und $b$ aus $\mathcal{B}$ bezüglich ihrer Featuremengen $A$ und $B$ bestimmt werden kann durch

$$S(a, b) = \vartheta f(A \cap B) - \alpha f(A \setminus B) - \beta f(B \setminus A) \tag{4.6}$$

In diesem Maß ist die Ähnlichkeit eine Funktion, die in Abhängigkeit von der Menge der gemeinsamen Merkmale wächst und in Abhängigkeit von den unterscheidenden Merkmalen abnimmt. Da $f$ eine Funktion ist, die Featuremengen auf nicht–negative reelle Zahlen abbildet, kann sie als Gewichtsfunktion verwendet werden, die je nach Variation den Einfluß, den die Mengen $X$, $Y$ und $Z$ auf die Ähnlichkeit haben, modelliert.

Das Contrast–Modell spezifiziert demnach nicht nur ein einziges festes Ähnlichkeitsmaß, sondern eine ganze Schar solcher Maße, die von den Werten der Parameter $\vartheta$, $\alpha$ und $\beta$ abhängen. Bei der Wahl von $\alpha = \beta = 0$ ist die Ähnlichkeit zwischen zwei Fällen eine Funktion ihrer gemeinsamen Merkmale. Ist dagegen nur $\vartheta = 0$, so entspricht $S$ einem Unähnlichkeitsmaß. Wählt man für $\alpha$ und $\beta$ unterschiedliche Werte, wird ein *asymmetrischer* Ähnlichkeitsbegriff modelliert.

Die wichtigste Eigenschaft des Contrast–Modells ist die, daß durch das spezifizierte Ähnlichkeitsmaß das Auftreten von $\alpha$-Fehlern bei der Fallauswahl reduziert werden kann. Dazu ist es jedoch erforderlich, das Durchsuchen der Fallbasis durch on–line–Ähnlichkeitsberechnungen zu steuern.

Eine solche ähnlichkeitsgesteuerte Suche liefert als Ergebnis eine Teilmenge von Fällen, die dann als Eingabe für den Prozeß der Brauchbarkeitsbewertung möglicher Lösungskandidaten dienen. Da nur diese ausgewählten Fälle zur Lösung des gegebenen Problems herangezogen werden können, entspricht das Contrast–Modell also quasi einem „0/1–Filter“, der entscheidet, ob ein Fall für die Lösung relevant sein kann oder nicht.

### 4.3.2 Eignungsbewertung

Ziel dieser zweiten Phase des Auswahlprozesses ist es, auf der Grundlage der in der ersten Phase ausgewählten Fälle zu entscheiden,

1. welche Strategie im weiteren Verlauf des Lösungsprozesses verfolgt werden soll (*case–based reasoning* oder *from–scratch–reasoning*) und
2. welcher der Fälle für das fallbasierte Schließen am besten geeignet ist.

Sei $\mathcal{C}$ eine $n$-elementige Menge von Fällen aus $\mathcal{B}$, die in der ersten Phase der Ähnlichkeitsbestimmung durch das Contrast–Modell von Tversky ausgewählt worden sind. Dann sind folgende drei Situationen zu unterscheiden:

1. die Menge $\mathcal{C}$ ist leer, d.h., es wurden *keine ähnlichen Fälle* gefunden:
   das Problem muß durch from–scratch–reasoning gelöst werden
2. $\mathcal{C}$ ist nicht leer (d.h. $n > 0$), und $\mathcal{C}$ enthält einen mit dem gegebenen Problem *identischen Fall* sowie $n - 1$ weitere ähnliche Fälle:
   die Lösung des identischen Falles wird für den Problemfall übernommen
3. es werden nur *ähnliche*, aber kein identischer Fall gefunden ($n \geq 1$):
   Die ausgewählten Fälle werden ihren berechneten Ähnlichkeiten nach geordnet. Über die Distanz jedes Falles aus $\mathcal{C}$ zum gegebenen Problem werden die Kosten abgeschätzt, die bei der Anpassung der Lösung an den Problemfall entstehen würden.
   Die Distanz ist jedoch ein ziemlich einfaches Maß für diese Kosten. Um eine bessere Abschätzung zu bekommen, ist es erforderlich, den Aufwand mitzuberücksichtigen, der bei der Angleichung der Unterschiede zwischen einem bekannten Fall und dem Problem anfällt. Eine Anordnung der Fälle nach den Kosten für die Anpassung ihrer Lösungen ist deshalb aussagekräftiger. Dabei können Fälle, deren Anpassungskosten eine gewisse Schranke überschreiten, von vorneherein schon ausgeschlossen werden.
   Wird $\mathcal{C}$ dadurch auf die leere Menge reduziert, muß die Problemlösung von Grund auf neu konstruiert werden.
   Bei $|\mathcal{C}| = 1$ wird der noch verbliebene Fall weiter betrachtet. Ist $|\mathcal{C}| > 1$, dann werden nur die Fälle weiter betrachtet, deren Distanz kleiner als eine feste Obergrenze ist.

Im letzten Schritt der Brauchbarkeitsbewertung muß für jeden Fall, der entsprechend geringe Anpassungskosten verspricht, noch festgestellt werden, ob die Unterschiede zwischen ihm und dem gegebenen Problem zu vernachlässigen sind oder ob er aufgrund dieser Unterschiede doch nicht zur Lösung herangezogen wird.

Es ist offensichtlich, daß ein rein statistisches Vorgehen hier nicht ausreichend sein kann. Auf der einen Seite kann der Unterschied in nur einem Merkmal trotz einer eventuell hohen statistischen Ähnlichkeit so groß sein, daß der Fall für die Problemlösung ungeeignet ist. Auf der anderen Seite ist es möglich, daß zwar die statistische Ähnlichkeit aufgrund mehrerer Unterschiede zwischen beiden Fällen relativ klein ist, andererseits diese Unterschiede jedoch so gering sind, daß sie auf die Problemlösung keinen Einfluß haben.

Hier ist deshalb eine *wissensbasierte* Bewertung der Eignung eines Falles notwendig, in die auch Domänenwissen mit einfließen muß. Das Ziel dieser zweiten Phase ist somit eine Reduzierung der $\beta$-Fehler.

# Kapitel 5

# Assoziative Suche in Retrieval–Systemen

Die in dieser Diplomarbeit betrachtete Problematik des Retrievals von Fällen, die über Attribut–Wert–Paare repräsentiert werden, ist eng verknüpft mit dem Gebiet der *mehrdimensionalen Suchprobleme*, das in den letzten Jahren vor allem in der Theorie der Datenstrukturen und Algorithmen sehr an Bedeutung gewonnen hat. Mehrdimensional heißt hier, daß die Datensätze durch Schlüssel identifiziert werden, die sich aus mehreren Komponenten zusammensetzen (in unserem Falle Attribute).

Mehrdimensionale Suchprobleme treten bei fast allen modernen Information–Retrieval– bzw. Datenbank–Systemen auf, zu deren Hauptaufgaben es gehört, einem Benutzer aus einer umfangreichen Menge an Daten schnell die gewünschte Information bereitzustellen. Der Benutzer formuliert dabei Anfragen an das System, in denen er Bedingungen für die zu suchenden Daten über Attributwerte spezifiziert. Das System sucht aus dem vorhandenen Datenbestand alle Daten, die diese Bedingungen erfüllen, und liefert sie als Ergebnis zurück. Werden in den Anfragen Bedingungen für mehr als eines der Repräsentationsattribute angegeben, so spricht man auch von Mehrattributsuche oder *assoziativem Retrieval*.

Um bei Suchanfragen nicht jeweils den kompletten Datenbestand durchsuchen zu müssen, sind effiziente Zugriffspfadstrukturen erforderlich, die die gegebenen Daten geeignet organisieren, sowie optimale Suchalgorithmen, die auf diesen Strukturen ablaufen können. Einen guten Überblick über das große Gebiet der mehrdimensionalen Datenstrukturen und Algorithmen gibt K.J. Jacquemain in [Jac88].

Auch wir gehen in dieser Diplomarbeit davon aus, daß die Fälle durch eine Menge von Attributen beschrieben sind. Ist $k$ die Anzahl der Suchattribute, so entspricht jeder Fall einem geordneten $k$-Tupel, dessen Komponenten gerade die $k$ Suchschlüsselwerte dieses Falles enthalten. Geometrisch gesprochen können die Fälle somit als Punkte eines $k$-dimensionalen Datenraumes angesehen werden, der durch die $k$ Attribute aufgespannt wird. Die Attribute selbst entsprechen den Koordinaten der Punkte.

## 5.1 Anfragetypen

In Information–Retrieval–Systemen können mehrere Typen von Retrieval–Anfragen (*Queries*) auftreten, die verschiedene Ergebnismengen erwarten. Die Anfragetypen unterscheiden sich in der Spezifikation der Bedingungen an die Suchschlüsselattribute. Man erwartet von Retrieval–Sytemen, daß sie je nach Art der an sie gestellten Anfrage einen entsprechenden Suchvorgang auf der vorhandenen Datenmenge starten.
Grundsätzlich unterscheidet man zwei Klassen von Anfragen (vgl. dazu auch [Ben75]):

- *Intersection Queries*: die gesuchten Daten bilden einen Durchschnitt mit der vorhandenen Datenmenge
- *Best Match Queries*: der gesuchte Datensatz muß nicht unbedingt vorhanden sein; hier wird nach möglichst ähnlichen Daten gesucht

Bei der Klassifikation der Queries gehen wir im weiteren von folgender Notation aus.

**Gegeben:**

- $B$ als die Menge der vorhandenen Datensätze, die alle durch $n$ Attribute $A_1, A_2, \ldots, A_n$ beschrieben seien (mit $A_i \neq A_j$ für $i \neq j$)
- $l$ verschiedene geordnete Wertebereiche, die den Attributen zugeordnet sind ($l \leq n$); es bezeichne $W_j := W(A_j)$ den Wertebereich des Attributes $A_j$
- jeder Datensatz ist ein geordnetes $n$-Tupel der Form $(a_1, a_2, \ldots, a_n)$ mit $a_j \in W_j$
- die Attribute $A_1, A_2, \ldots, A_k$ seien die Suchschlüssel, d.h. die Attribute, für die in Anfragen Bedingungen spezifiziert werden können; somit entspricht $k$ der Dimensionalität des zu durchsuchenden Datenraumes und jeder Datensatz $R \in B$ einem Punkt in diesem Raum

**Eingabe:** Anfrage $Q$, in der Bedingungen für die Suchschlüssel in Form von booleschen Termen angegeben sind

**Ausgabe:** die Menge $Y_Q := \{R \in B \mid P_Q(R)\}$ als Ergebnis der Anfrage $Q$;
dabei sei $P_Q(R)$ ein einstelliges Prädikat auf der Menge $B$, das genau dann wahr ist, wenn die Suchschlüsselwerte des Satzes $R$ die in $Q$ spezifizierten Bedingungen erfüllen

### 5.1.1 Intersection Queries

Diese Art von Queries ist der wohl gebräuchlichste Anfragetyp in Retrieval–Systemen. Man unterscheidet dabei folgende in ihrer Komplexität wachsende Queries:

1. **Exact Match Query (Point Query):** einfachster Anfragetyp, der nach einem bestimmten Datensatz sucht, indem für jeden Suchschlüssel ein Wert spezifiziert wird;

$$Q = (A_1 = a_1) \wedge (A_2 = a_2) \wedge \ldots \wedge (A_k = a_k)$$

2. **Partial Match Query:** spezifiziert $s < k$ Schlüsselwerte;

$$Q = (A_{i1} = a_{i1}) \wedge (A_{i2} = a_{i2}) \wedge \ldots \wedge (A_{is} = a_{is})$$

mit $1 \leq s < k$ und $1 \leq i_1 < i_2 < \ldots < i_s \leq k$

3. **Range Query:** für jeden Schlüssel $A_j$ wird ein Bereich $r_j \subseteq W_j$ der Form $r_j = [l_j \leq a_j \leq u_j]$ spezifiziert;

$$\begin{aligned} Q &= (A_1 \in r_1) \wedge (A_2 \in r_2) \wedge \ldots \wedge (A_k \in r_k) \\ &= (A_1 \geq l_1) \wedge (A_1 \leq u_1) \wedge \ldots \wedge (A_k \geq l_k) \wedge (A_k \leq u_k) \end{aligned}$$

4. **Partial Range Query:** spezifiziert für $s < k$ Schlüssel jeweils einen Bereich;

$$Q = (A_{i1} \in r_{i1}) \wedge \ldots \wedge (A_{is} \in r_{is})$$

mit $1 \leq s < k$, $1 \leq i_1 < \ldots < i_s \leq k$ und $r_{ij} = [l_{ij} \leq a_{ij} \leq u_{ij}]$ $(1 \leq j \leq s)$

Die Range Queries sind die allgemeinste Form der Intersection Queries. Die angegebenen Bereiche definieren dabei Regionen innerhalb des Datenraumes. Eine Point Query kann als Spezialfall dieses Anfragetyps angesehen werden, bei der die Regionen gerade Punkte sind, während eine Partial Match Query mit $s$ Schlüsselwerten ($k$-$s$)–dimensionale Hyperebenen spezifiziert. Abbildung 5.1 zeigt eine 2-dimensionale Veranschaulichung der vier Anfragetypen für 2 Attribute $A_1$ und $A_2$.

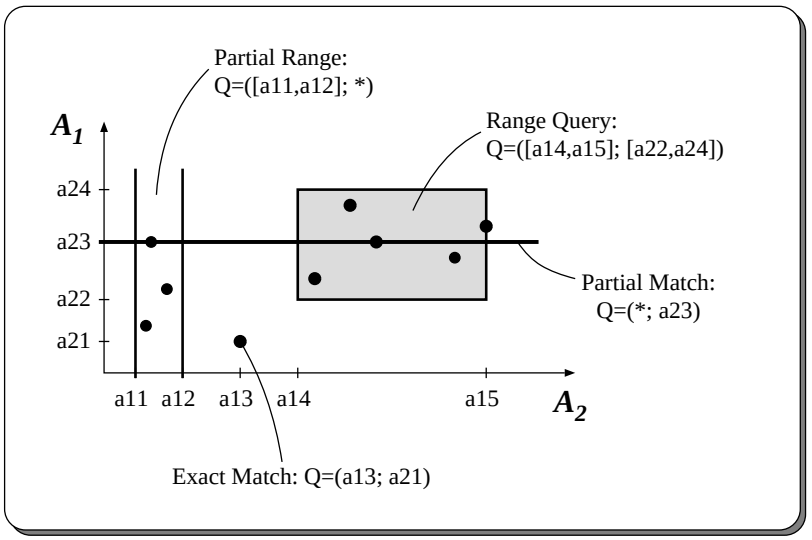


Abbildung 5.1: Intersection Queries

### 5.1.2 Best Match Query

Dieser Anfragetyp, der auch als *Nearest Neighbor Query* bezeichnet wird, erfordert im Gegensatz zu den Intersection Queries eine unscharfe Suche, da der durch die Anfrage spezifizierte Datensatz i.a. nicht im Datenbestand vorhanden sein muß und somit nach den *ähnlichsten* Sätzen zu suchen ist.

Die Ähnlichkeitsbeziehung zwischen zwei Punkten ergibt sich dabei durch ihre Nachbarschaft im Datenraum (Abb. 5.2). Sie kann mit Hilfe von Ähnlichkeitsmaßen oder Distanzfunktionen ausgedrückt werden. Der nächste Nachbar zu einem gegebenen Anfragepunkt entspricht somit dem Datensatz, dessen Suchschlüsselwerte den „besten Match“ für die Werte des gesuchten Objektes liefern.

**gegeben:** eine Distanzfunktion $D$ bzw. ein Ähnlichkeitsmaß $S$
$B$: Sammlung von Datensätzen, interpretiert als Punkte in einem $k$-dimensionalen Raum
$q$: durch Anfrage $Q$ spezifizierter Datensatz

**gesucht:** $p$: nächster Nachbar zu $q$, wobei gelte
1. $p \in B$
2. $\forall r \in B : (r \neq p \Rightarrow [D(r,q) \geq D(p,q)$ bzw. $S(r,q) \leq S(p,q)])$

Die Nearest Neighbor Query ist für das in dieser Diplomarbeit untersuchte Problem des ähnlichkeitsbasierten Fallretrievals von zentraler Bedeutung. In Kapitel 6 erfolgt deshalb eine eingehendere Betrachtung dieses Aspektes.

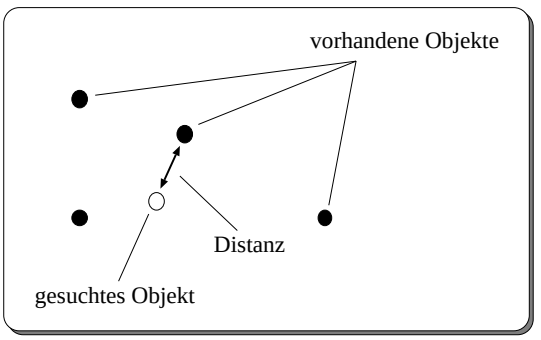


Abbildung 5.2: Best Match Query

## 5.2 Mehrdimensionale Zugriffspfade

Bei der enormen Größe des Datenbestandes in heutigen Informationssystemen sind Suchverfahren, die die beschriebenen Anfragetypen mit einer Laufzeit proportional zur Anzahl der gespeicherten Daten unterstützen, nicht tragbar.

Deshalb mußten Datenstrukturen gefunden werden, in denen die Datensätze so abgespeichert werden können, daß sich die unterschiedlichen Anfragen jeweils effizient realisieren lassen. Dabei sind im mehrdimensionalen Fall alle $k$ Suchattribute ($k > 1$) zu berücksichtigen, die in den zu erwartenden Anfragen für die Datensätze spezifiziert werden können. Für analytische Betrachtungen ist es ausreichend, wenn man sich nur auf diese $k$ Schlüsselwerte beschränkt, so daß die Datensätze als $k$-Tupel angesehen werden können. Die Tupel entsprechen dann den Datensatz–Schlüsseln, die sich aus $k$ verschiedenen und unabhängigen Komponenten zusammensetzen.

Dieser Abschnitt soll eine kurze Übersicht über einige mehrdimensionale Zugriffspfadstrukturen geben.

Die wichtigsten Eigenschaften mehrdimensionaler Zugriffspfade ergeben sich aus den Grundproblemen und den daraus abgeleiteten Forderungen, die an eine anfrageunterstützende Datenstruktur zu stellen sind. Folgende Grundprobleme sind in diesem Zusammenhang zu berücksichtigen (vgl. auch [LS87]):

1. **Erhaltung der topologischen Struktur des Datenraumes:** Mehrdimensionale Zugriffspfadstrukturen unterteilen den Raum durch eine mehrdimensionale Gitterstruktur in mehrere disjunkte Regionen.
   Die erhobene Forderung nach Erhaltung der topologischen Struktur bedeutet hier eine Clusterbildung bei der externen Speicherung der Datensätze. Dabei sollen die Sätze, die in einem gemeinsamen durch die Partitionierung des Raumes entstandenen Bereich liegen, auch extern zusammen in einem sog. *Bucket* abgelegt werden (Abb. 5.3). Buckets entsprechen auf Externspeicherseite jeweils den Bereichen, in die der Raum durch die Datenstruktur partitioniert wird.

2. **Stark variierende Dichte der Objekte:** Im allgemeinen schwankt die Dichte der im Datenraum verteilten Objekte sehr stark. Neben Bereichen, in denen sehr viele Objekte liegen, kann es auch Regionen geben, die keine oder nur wenige Objekte enthalten. In solchen Fällen, in denen keine gleichmäßige Verteilung der Daten vorliegt, stellt sich die Forderung nach einer Anpassungsfähigkeit der Datenstruktur an eine gegebene Objektdichte. Dies betrifft vor allem die Aufteilung des Raumes. Sie sollte nicht von vornherein fest vorgegeben sein, sondern sich in Abhängigkeit von der jeweiligen Dichte ergeben.

3. **Dynamische Reorganisation der Struktur:** Ein Zugriffspfad sollte auf die durch Einfügen und Löschen von Objekten entstehenden Änderungen des Datenraumzustandes reagieren können und somit jederzeit dynamisch aktualisierbar sein (Abb. 5.4).

4. **Balancierte Zugriffspfadstruktur:** Für einen gleichförmigen Zugriff auf den Externspeicher ist eine balancierte Zugriffsstruktur erforderlich, die gewährleistet, daß bei einer Anfrage in der Regel nicht mehr als 2 oder 3 Externspeicherzugriffe benötigt werden, um auf ein Objekt zuzugreifen.

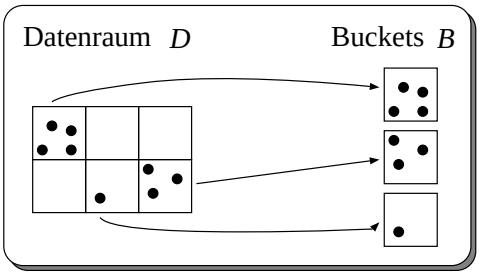


Abbildung 5.3: Topologieerhaltung

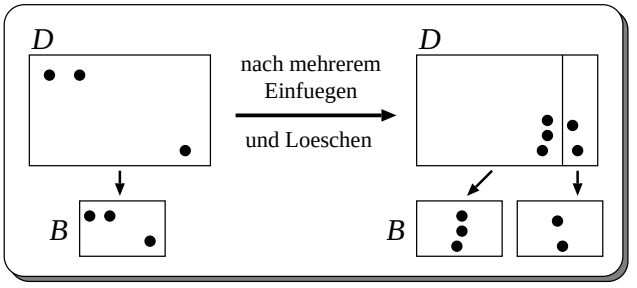


Abbildung 5.4: Dynamische Reorganisation

Im folgenden sind nun einige bekannte mehrdimensionale Zugriffspfadstrukturen aufgeführt. Der $k$-d-Baum ist, wie schon erwähnt, zentraler Gegenstand dieser Diplomarbeit. Er wird in Kapitel 7 noch ausführlich beschrieben, während die folgenden Strukturen hier nur übersichtsartig vorgestellt werden sollen. Zu vertieften Betrachtungen dieser Strukturen wird deshalb jeweils auf weiterführende Literatur verwiesen.

**Mehrdimensionale Zugriffspfadstrukturen:**

- **$k$-d-Baum:**
  - mehrdimensionaler binärer Suchbaum als Verallgemeinerung des eindimensionalen Suchbaums (vgl. hierzu z.B. [Meh84a])
  - jeder Baumknoten repräsentiert eine Menge von Datensätzen
  - die Wurzel des Baumes repräsentiert die gesamte Datenmenge
  - innere Knoten partitionieren diese Menge jeweils nach einem der Suchschlüsselattribute
  - Suchvorgänge innerhalb des Baumes bestehen aus Vergleichsoperationen bezüglich der Attribute, durch die die Daten an den inneren Knoten jeweils partitioniert werden
- **Grid File** (Abb. 5.5, vgl. [NHS84]):
  - der Datenraum $D$ wird durch ein orthogonales Raster (engl. *grid*) partitioniert
  - es wird jeweils nur in einer Dimension gesplittet
  - das Raster wird auf dem $k$-dimensionalen Raum $D$ durch $k$ Skalierungsvektoren $\{S_i\}_{i=1}^{k}$ (*Scales*) definiert
  - zwischen einem Gridblock $GB$ (definiert durch die $S_i$) und einer Komponente des $k$-dimensionalen dynamischen Vektors $GD$ (Grid Directory) besteht eine 1:1-Beziehung
  - ein Element von $GD$ enthält einen Zeiger auf ein Bucket $B$
  - ein Bucket $B$ enthält alle Datensätze des zugehörigen Gridblockes $GB$

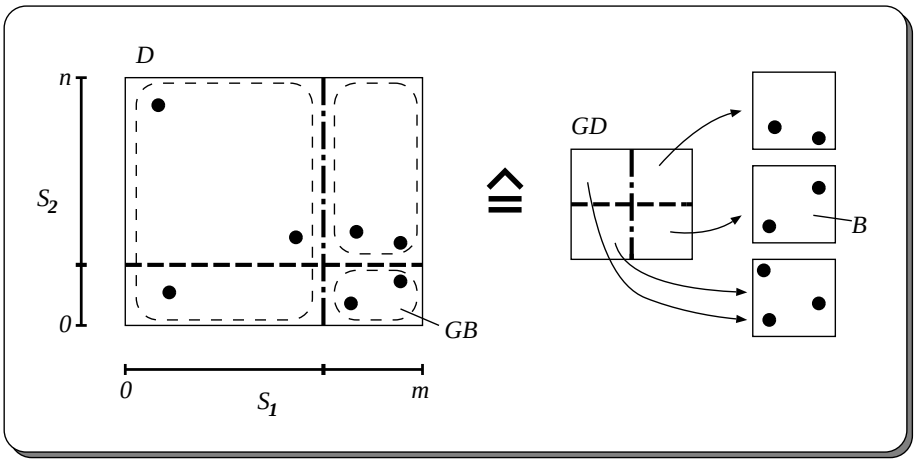


Abbildung 5.5: Grid–File–Konzept

- **Quad–Tree** (Quadranten–Baum, vgl. [Jac88]):
  - ebenfalls eine Verallgemeinerung des binären Suchbaums
  - ein $k$-dimensionaler Quad–Tree ist ein $2^k$-närer Wurzelbaum, bei dem jeder Knoten den Schlüssel eines Satzes sowie $2^k$ Zeiger auf ebenso viele Unterbäume (Quadranten) enthält
  - die Wurzel teilt den $k$-dimensionalen Raum in $2^k$ Quadranten auf; der $i$-te Unterbaum eines Knotens enthält die Datensätze, die im $i$-ten Quadranten liegen
  - rekursive Aufteilung jedes Quadranten durch die Wurzel des entsprechenden Unterbaumes

Tabelle 5.1 zeigt abschließend eine Zusammenstellung, die die erwähnten Datenstrukturen auf die Erfüllung der verschiedenen Anforderungen hin vergleicht. Die Erprobung der praktischen Tauglichkeit dieser mehrdimensionalen Strukturen in Information–Retrieval–Systemen steht jedoch zum Teil noch aus.

| **Vergleich** | **Quad–Tree** | **k-d-Baum** | **Grid File** |
|---|---|---|---|
| Dynamische Reorganisation | nein | ja | ja |
| Clusterbildung | nein | ja | ja |
| Range Queries, Best Match Queries | rekursive Baumsuche | rekursive Baumsuche | Berechnung des Buckets |
| Eignung für Sekundärspeicher | nein | ja | ja |
| Exact Match Queries | $\log_k n$ | $\log_2 n$ | 2 |
| optimale Dateigröße | klein | mittel – groß | beliebig groß |
| Balancierung | nein | aufwendig | ja |

Tabelle 5.1: Vergleich mehrdimensionaler ZP–Strukturen

# Kapitel 6

# Nearest–Neighbor–Suche

Der in dieser Diplomarbeit verfolgte Lösungsansatz geht von der Interpretation der Fälle als Punkte in einem mehrdimensionalen Datenraum aus, der durch die Wertebereiche der fallbeschreibenden Attribute aufgespannt wird. Somit kann man die Ähnlichkeitssuche als mehrdimensionale, assoziative Suche nach den nächsten Nachbarn (*nearest neighbors*) bzw. besten Matches eines gegebenen Problemfalles auffassen.

## 6.1 Grundbegriffe und Notationen

Zunächst werden einige grundlegende Begriffe eingeführt, die wir im folgenden verwenden wollen (vgl. hierzu auch [Jac88]).

Es bezeichne $\mathcal{N}$ die Menge der natürlichen Zahlen einschließlich der Null, $\mathcal{Z}$ die Menge der ganzen Zahlen und $\mathcal{R}$ die reellen Zahlen. Weiterhin sei $\mathcal{N}_+ := \mathcal{N} \setminus \{0\}$.

Die Logarithmusfunktion ohne Angabe der Basis, also einfach $\log x$ für $x \in (0, \infty)$, bezeichne den Logarithmus zur Basis 2.

Des weiteren werden die Größenordnungssymbole $O$, $\Omega$ und $\Theta$ verwendet, die für Funktionen $f,\ g : \mathcal{N} \to [0, \infty)$ wie folgt definiert sind:

1. $f(n) \in O(g(n)) :\Longleftrightarrow (\exists c > 0)(\exists n' \in \mathcal{N})(\forall n \geq n')\ f(n) \leq c \cdot g(n)$
   ($f(n)$ ist höchstens von der Größenordnung $g(n)$)
2. $f(n) \in \Omega(g(n)) :\Longleftrightarrow g(n) \in O(f(n))$
   ($f(n)$ ist mindestens von der Größenordnung $g(n)$)
3. $f(n) \in \Theta(g(n)) :\Longleftrightarrow f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n))$
   ($f(n)$ ist genau von der Größenordnung $g(n)$)

Dazu ergänzend werden folgende Begriffe benötigt.

**Definition (r-närer Suchbaum):** Ein geordneter *r-närer Suchbaum* ($r \geq 2$) ist ein Wurzelbaum, bei dem jeder Knoten höchstens $r$ Nachfolgeknoten (Söhne) hat.
Alle Söhne eines Knotens sind dabei eindeutig identifizierbar, so daß man stets eindeutig vom $j$-ten Sohn ($1 \leq j \leq r$) sprechen kann.

Im Zusammenhang mit Suchbäumen wird außerdem noch die Bezeichnung *Level* (oder auch Höhe) eines Knotens verwendet.

**Definition (Level):** Sei $v'$ die Wurzel eines Suchbaums. Dann ist $v'$ auf *Level* 1, und für alle Knoten $v \neq v'$ gilt: $v$ ist auf Level $i + 1$ genau dann, wenn der Vater von $v$ auf Level $i$ ist.

Die Höhe eines Baumes beschreibt die Länge des längsten Weges im Baum, somit also den höchsten Level, den ein Knoten in diesem Baum besitzt.

**Definition(balancierte Bäume):** Eine Klasse von Bäumen heißt *balanciert* zur Höhe $f(n)$, wenn die Höhe jedes Baumes mit $n$ Knoten aus dieser Klasse höchstens $f(n)$ ist und Such–, Einfüge– und Löschoperationen stets in $O(f(n))$ Schritten durchgeführt werden können und der resultierende Baum wieder zu dieser Klasse gehört.

## 6.2 Problemstellung

Die Nearest–Neighbor–Suche ist eines der bekanntesten *Closest–Point*–Probleme, einer Klasse von geometrischen Problemen, die in den letzten Jahren immer mehr zum Gegenstand umfangreicher Forschungstätigkeiten auch in der Informatik geworden ist. Hier steht vor allem die Suche nach geeigneten Verfahren und schnellen Algorithmen zur Lösung dieser geometrischen Probleme im Vordergrund. Dies beinhaltet natürlich auch den Entwurf entsprechender Datenstrukturen. Einige diesbezügliche Ansätze wurden bereits in Kapitel 5 erwähnt.

Das *Nearest–Neighbor*–Problem (kurz *NN–Problem*) läßt sich formal wie folgt definieren (vgl. auch [Jac88]):

**Definition (NN–Suche):** Gegeben sei eine $n$-elementige Punktmenge $B$ aus einem $k$-dimensionalen Datenraum $\mathcal{R}^k$ ($k$, $n \in \mathcal{N}$) und ein Distanzmaß $d : \mathcal{R}^k \times \mathcal{R}^k \rightarrow [0, \infty)$. Dann versteht man unter dem Problem der *NN–Suche* die Bestimmung eines Punktes $p$ aus $B$ mit der Eigenschaft, daß $p$ von allen Punkten aus $B$ zu einem Anfragepunkt $q \in \mathcal{R}^k$ am nächsten gelegen (und somit am ähnlichsten) ist, also

$$p \in B \quad \text{mit} \quad d(p,q) = min\{d(r,q) |\ r \in B\}.$$

Es ist offensichtlich, daß $p$ nicht eindeutig bestimmt ist.

Neben diesem NN–Problem gibt es noch zwei weitere „klassische“ Suchprobleme der in der Literatur als Closest–Point–Searching bezeichneten Problemklasse.

**Definition (MNN–Problem):** Das *MNN–Problem* bestimmt zu einem Anfragepunkt $q \in \mathcal{R}^k$ die $m$ nächsten Nachbarn ($m \in \mathcal{N}$). Gesucht ist somit eine $m$-elementige Teilmenge

$$MNN(q,m) := \{p_1, p_2, \ldots, p_m\} \subseteq B$$

mit $d(p_i, q) \leq d(r, q)$ für alle $i$=$1, 2, \ldots, m$ und alle $r \in B \setminus MNN(q,m)$.
Auch $MNN(q,m)$ ist nicht eindeutig bestimmt.

**Definition (FRNN–Problem):** Das *Fixed-Radius-Near-Neighbor*–Problem (*FRNN–Problem*) bestimmt alle Punkte aus $B$, die zum Anfragepunkt $q$ einen Abstand von höchstens $max > 0$ haben. Gesucht ist die Menge

$$FRNN(q, max) := \{r \in B \mid d(r,q) \leq max\}.$$

Sowohl beim MNN– als auch beim FRNN–Problem unterscheidet man jeweils noch zwischen einer starren und einer flexiblen Version; starr heißt, daß lediglich Anfragen mit einem einzigen festen $m \in \mathcal{N}$ (MNN–Problem) bzw. einem festen $max > 0$ (FRNN–Problem) beantwortet werden können. In vielen Fällen sind jedoch flexible Strukturen und Algorithmen wünschenswert, die eine Variation dieser Parameter erlauben.

Der Ansatz der $k$-d-Baumstruktur, der in dieser Diplomarbeit untersucht wird, berücksichtigt diese Aspekte in entsprechenden Algorithmen. Dieser Komplex ist in den Kapiteln 7–9 Gegenstand ausführlicher Betrachtungen.

## 6.3 Untere Schranken

K.J. Jacquemain gibt in [Jac88] für die drei beschriebenen Closest–Point–Probleme untere Grenzen bezüglich des Suchaufwandes in Abhängigkeit von der Anzahl $n$ der in $B$ enthaltenen Punkte an.

Sei $B \subset \mathcal{R}^k$ gegeben.

**Satz 1** *Für das NN–Problem ist $O(\log n)$ eine untere Schranke.*

**Beweis:** (indirekt)
Das NN–Problem kann auf das Membership–Problem reduziert werden. Soll festgestellt werden, ob ein Anfragepunkt $q \in \mathcal{R}^k$ zur Menge $B$ gehört, so kann man zunächst $NN(q)$ bestimmen und muß dann nur noch $q$ mit $NN(q)$ vergleichen, da $q \in B \Leftrightarrow NN(q) = q$.
Angenommen, die NN–Suche könnte in weniger als $\log(n)$ Schritten durchgeführt werden, so müßte dies auch für das Membership–Problem gelten. Da das jedoch nicht der Fall ist (vgl. z.B. [HS81]), ist $\log(n)$ eine untere Schranke für das NN–Problem.

**Satz 2** *Für das MNN–Problem ist $O(m + \log n)$ eine untere Schranke.*

**Beweis:**
Es gilt natürlich $\Omega(m + \log n) = \Omega(max(m, \log n))$ (die linke Schreibweise ist üblicher und wird deshalb auch hier verwendet, der Nachweis wird jedoch für die rechte Schreibweise geführt).

1. Fall: $m \geq \log n$
Hier ist die untere Schranke $\Omega(m)$ zu zeigen. Dies ist trivial, da ja $m$ Punkte ausgegeben werden sollen und somit auf jeden Fall $m$ Schritte notwendig sind.

2. Fall: $m < \log n$
In diesem Falle muß die Schranke $\Omega(\log n)$ gezeigt werden. Dies gilt nach Satz 1, da nach der Bestimmung von $MNN(q, m)$ nur $m$ Abstandsberechnungen und $m - 1$ Vergleiche notwendig sind, um $NN(q)$ zu ermitteln.

**Satz 3** *Für das FRNN–Problem ist $\Theta(t + \log n)$ eine untere Schranke.*

**Beweis:**
Der Beweis verläuft analog zu dem Beweis von Satz 2 (man setze überall $t$ statt $m$ ein und $FRNN(q, r)$ für $MNN(q, m)$).

Jacquemain zeigt darüber hinaus noch, daß alle drei Schranken sogar jeweils untere Grenzen, also größte untere Schranken für diese Probleme sind. Dazu gibt er in [Jac88] jeweils Algorithmen an, deren Laufzeiten diese Schranken erreichen.

## 6.4 Aufwandsaspekte

Bekannte algorithmische Ansätze zur Lösung des NN–Problems (vgl. z.B. [Das91]) berücksichtigen besonders auch Aspekte, die eine effiziente Implementierung betreffen.

Eines der Hauptaugenmerke liegt dabei auf einer Reduzierung des zur Bestimmung der nächsten Nachbarn erforderlichen Berechnungsaufwandes. Dies betrifft insbesondere den Aufwand, der bei der Berechnung von Ähnlichkeiten zwischen Punkten bzw. Fällen anfällt. Die erwähnten Ansätze versuchen vor allem, die Menge der Punkte, die mit dem gegebenen Anfragepunkt verglichen werden müssen, möglichst klein zu halten.

Der erforderliche Berechnungsaufwand läßt sich im wesentlichen zwei Phasen zuordnen:

1. der Aufwand, der beim Aufbau bzw. der Reorganisation der Datenstruktur anfällt, die ein effizientes Suchen unterstützen soll (*Preprocessing*– oder *Lernphase*)

2. der Aufwand des eigentlichen Suchvorganges im organisierten Datenraum (*operationale Phase*)

In sehr einfachen Verfahren, die keine Preprocessing–Phase enthalten, sind die Kosten der operationalen Phase sehr hoch. Bei etwas weiter entwickelten Ansätzen ist dieser Suchaufwand geringer, dafür fallen jedoch höhere Kosten bei der Organisation des Datenbestandes an.

Es offenbart sich hier das typische Problem eines Tradeoff zwischen dem Aufwand der Lernphase und dem der operationalen Phase. Es sind deshalb bei jeder Anwendung Überlegungen darüber anzustellen, bis zu welcher Höhe die Kosten in der Preprocessing–Phase noch akzeptabel sind. Dies hängt z.B. auch davon ab, wie groß der Datenbestand ist oder wie häufig sich dieser ändert.

Aufgrund dieser vielschichtigen Aspekte ist es oft auch erforderlich, die Anwendbarkeit und Effizienz solcher Ansätze im Hinblick auf eine spezielle Anwendung durch ausgedehnte experimentelle Auswertungen zu untersuchen und zu vergleichen.

# Teil III

# Ansatz

# Kapitel 7

# Der k-d-Baum

Die von J. Bentley 1975 eingeführte Datenstruktur des $k$-d-Baumes ([Ben75]) ist ein mehrdimensionaler binärer Suchbaum zur Unterstützung des assoziativen Retrievals; $k$ ist hierbei die Dimensionalität des Datenraumes, der durchsucht werden muß.

Der $k$-d-Baum kann als eine natürliche Verallgemeinerung des Konzepts der eindimensionalen binären Suchbäume aufgefaßt werden, wo jeder gespeicherte Datensatz durch genau einen Schlüsselwert charakterisiert ist und die Vergleiche während eines Suchvorganges nur auf diesem einen Schlüsselwert durchgeführt werden. Im mehrdimensionalen Fall sind dagegen Schlüssel zu betrachten, die aus mehreren Werten zusammengesetzt sind. Viele Aussagen, die für die 1-d-Bäume gelten, können jedoch auch auf den mehrdimensionalen Fall mit $k > 1$ übertragen werden (vgl. hierzu z.B. [Meh84a] bzw. [Meh84b]).

J.L. Bentley zeigt in [Ben75], daß diese Datenstruktur insbesondere auch zur Unterstützung der *Best–Match*–Suche herangezogen werden kann. Bevor wir jedoch in Kapitel 8 näher auf die Verwendung von $k$-d-Bäumen für die Nearest–Neighbor–Suche eingehen, soll in diesem Kapitel zunächst die allgemeine Struktur eines $k$-d-Baums beschrieben werden, auf die wir uns in dieser Diplomarbeit beziehen. Auch werden hier noch kurz Überlegungen angeführt, die den Aufwand einiger Grundoperationen auf dieser Datenstruktur betreffen.

## 7.1 Definition

Gegeben seien $k$ Wertebereiche als geordnete Mengen $\{W_j\}_{j=1}^{k}$ und ein entsprechender $k$-dimensionaler Datenraum $W = W_1 \times \ldots \times W_k$ .

**Definition (k-d-Baum):** Es sei $B \subseteq W$ mit $B = \{X \mid X = (x_1, x_2, \ldots, x_k)\}$, $|B| = n$. Weiter sei $b \in \mathcal{N}_+$ beliebig aber fest vorgegeben.
Dann ist ein *k-d-Baum* $T^{(i1)}(B)$ für die Menge $B$ definiert durch:

1. Ist $n \leq b$, so ist $T^{(i1)}(B)$ ein Blattknoten, der alle $X \in B$ enthält ($i_1$:=0).
2. Ist $n > b$, dann bezeichnet $i_1$ die Dimension des Datenraumes $W$ ($1 \leq i_1 \leq k$), in der die Menge $B$ am Wurzelknoten des Baumes partitioniert wird. Die Wurzel von $T^{(i1)}(B)$ enthält einen Wert $p_{i1} \in W_{i1}$ und je einen Zeiger auf die $k$-d-Bäume $T_{\leq}^{(i2)}(B_{\leq})$ und $T_{>}^{(i3)}(B_{>})$ $(1 \leq i_2,\ i_3 \leq k)$, wobei

   $$B_{\leq} := \{X \in B \mid x_{i1} \leq p_{i1}\} \quad \text{und} \quad B_{>} := \{X \in B \mid x_{i1} > p_{i1}\}.$$

Jeder Baumknoten repräsentiert eine Teilmenge der im $k$-d-Baum gespeicherten Datensätze.

- Die *Wurzel* des Baumes repräsentiert den gesamten $k$-dimensionalen Raum $W$, der die Menge aller Sätze umfaßt.

- Alle anderen Knoten beschreiben jeweils eine Teilregion aus $W$, die eine bestimmte Teilmenge der Sätze enthält.

- Jeder *innere Knoten* definiert für die repräsentierte Teilmenge eine Partitionierung der Sätze in zwei disjunkte Mengen. Somit hat ein innerer Knoten genau zwei Nachfolgeknoten, wobei jeder Sohn eine dieser beiden Partitionen enthält.
  Die beiden Söhne entstehen dadurch, daß die durch den Knoten beschriebene Teilregion mittels einer Hyperebene, die orthogonal zu einer der $k$ Koordinatenachsen ist, in zwei Teile aufgesplittet wird.
  Hierbei ist durch $i_1$ die Koordinatenachse (bzw. der *Diskriminator*) festzulegen, an der der Raum gesplittet werden soll, und zum anderen der *Partitionswert* $p_{i1}$, der die Position der Hyperebene auf dieser Achse angibt.
  Innere Knoten werden jedoch nur dann generiert, wenn die Anzahl der zu partitionierenden Datensätze größer ist als die sog. *Bucketgröße b*. Ansonsten wird ein Blattknoten erzeugt.

- Gespeichert werden die Sätze letztendlich nur in den *Blattknoten*, so daß die inneren Knoten quasi als „Wegweiser“ zu den Sätzen dienen. Die Blattknoten enthalten kleine, disjunkte Teilmengen (*Buckets*) aller im Baum organisierten Datensätze.

- Die Menge, die durch einen inneren Knoten repräsentiert wird, ergibt sich aus allen Sätzen, die in den Blattknoten unterhalb dieses Knotens im Baum gespeichert sind.

Der $k$-d-Baum ist folglich ein binärer Suchbaum, dessen Knoten entweder genau zwei oder keinen Nachfolgeknoten haben. Man kann ihn deshalb als Verallgemeinerung der eindimensionalen binären Suchbäume auffassen. Da im Gegensatz zu den 1-d-Bäumen für die Partitionierungen mehrere Attribute zur Verfügung stehen, spielt beim Aufbau der Bäume an den inneren Knoten neben der Bestimmung der Partitionswerte auch die Auswahl der Diskriminatoren eine große Rolle.

In Bentleys ursprünglicher Definition des $k$-d-Baumes ([Ben75]) ist der Diskriminator eine natürliche Zahl zwischen 1 und $k$, der angibt, nach welcher Koordinate (bzw. nach welchem Attribut) an dem betreffenden Knoten verglichen wird. Bentley wählt die Diskriminatoren so, daß alle Knoten auf demselben Level im Baum den gleichen Diskriminator haben. Aus Gründen der Fairness soll dabei keine der Koordinaten bevorzugt werden. Deshalb erfolgt die Wahl des Diskriminators zyklisch in der Reihenfolge der Koordinaten. Die Wurzel hat den Diskriminator 1, ihre beiden Söhne den Diskriminator 2 und so weiter bis zum $k$-ten Level. Auf Level $k+1$ wird dann wieder nach der ersten Komponente verglichen. Alle Knoten auf Level $i$ besitzen somit den Diskriminator $((i-1) \bmod k)+1$.

Für den Fall, daß eine Menge in der $j$-ten Koordinate nicht gesplittet werden kann, da alle Tupel in dieser Komponente denselben Wert haben, wird die $(j+1)$–te Komponente herangezogen, bei nochmaliger Gleichheit die $(j+2)$–te Komponente und so fort, wobei nach der $k$-ten Komponente dann wieder die erste genommen wird.

Diese Art der Diskriminatorbestimmung ist jedoch nur eine von vielen Möglichkeiten. Wir werden in dieser Diplomarbeit eine andere Vorgehensweise wählen.

Schließlich sei noch angemerkt, daß man nicht von *der* $k$-d-Baum–Definition sprechen kann. Neben Bentleys Definition gibt es in der Literatur noch verschiedene Variationen des $k$-d-Baumes. Wir beziehen uns im weiteren auf die oben beschriebene Definition.

Abbildung 7.1 zeigt ein Beispiel für die Anordnung einer Menge von Punkten aus einem 2-dimensionalen Datenraum in einem $k$-d-Baum mit einer Bucketgröße von 2. In Abb. 7.1(a) ist der Datenraum in 5 disjunkte Teilräume aufgeteilt, die jeweils höchstens zwei Punkte der Menge $B = \{A, B, C, \ldots, I\}$ enthalten. Der entsprechende 2-d-Baum $T^{(1)}(B)$ ist in Abb. 7.1(b) dargestellt. Die Diskriminatoren seien hierbei für alle Knoten eines Levels jeweils gleich. Die Kreise bezeichnen innere Knoten, in denen der entsprechende Partitionswert angegeben ist. Auf die Wahl der Partitionswerte wird in Abschnitt 7.2.2 näher eingegangen. Die Rechtecke stellen die Blattknoten dar.

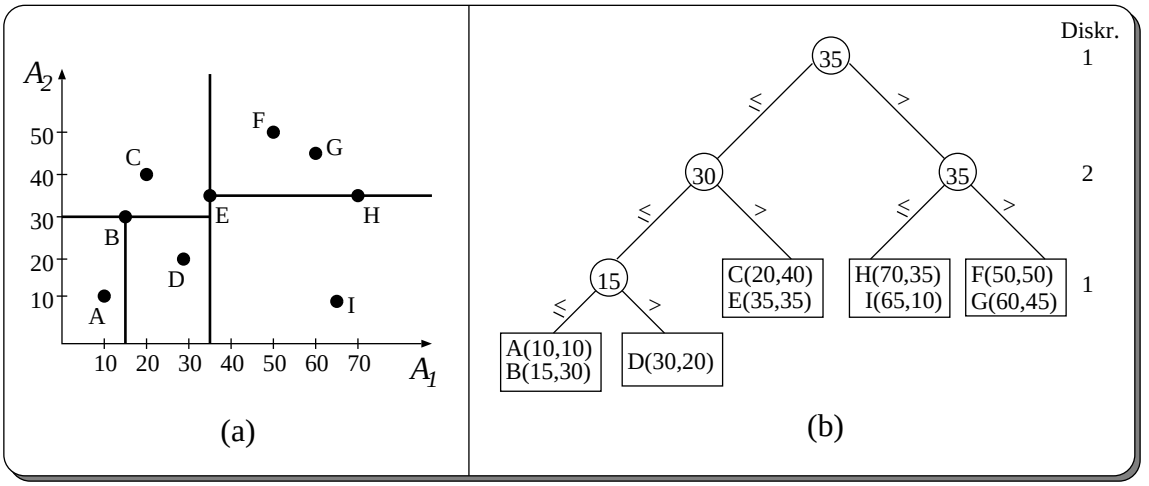


Abbildung 7.1: Beispiel für einen $k$-d-Baum

## 7.2 Aufwandsbetrachtungen

Im folgenden sollen die Kosten betrachtet werden, die zur Konstruktion eines $k$-d-Baumes erforderlich sind. Vorgegeben sei dabei eine Menge von $n$ $k$-Tupeln, die bei einer Bucketgröße $b = 1$ in einem $k$-d-Baum angeordnet werden sollen. Wir betrachten zunächst den Aufwand für einen Baumaufbau, bei dem die Auswahl der Partitionswerte an den inneren Knoten „zufällig“ erfolgt. Anschließend werden dann die Kosten für sog. *ideale* $k$-d-Bäume mit einer höhenbalancierten Struktur untersucht.

Um überhaupt von Kosten reden zu können, muß zuerst ein Maß für diese Kosten festgelegt werden. Wir folgen hier dem Ansatz von K.J. Jacquemain, der die Kosten über die Anzahl der Vergleiche bestimmt, die notwendig sind, um einen $k$-d-Baum für $n$ Tupel zu konstruieren.

### 7.2.1 Mittlere Konstruktionskosten

Im schlechtesten Fall wird beim zufälligen Aufbau des Baumes aus $n$ Tupeln an jedem Knoten gerade so partitioniert, daß immer genau ein einziges der Tupel aus der noch zu partitionierenden Menge einem der beiden Nachfolgeknoten zugeordnet wird und alle übrigen Tupel dem anderen Sohn des Knotens. Dadurch entsteht schließlich ein „entarteter“ Baum mit nahezu linearer Struktur und der Höhe $n$, wobei auf jedem Level $l > 1$ ein Blattknoten erzeugt wird. Die Worst–Case–Komplexität ist somit von der Größenordnung $\Theta(n^2)$.

Die Kosten, die im durchschnittlichen Fall anfallen, sind relativ einfach zu ermitteln, da man bei $k$-d-Bäumen im Prinzip die gleichen Verhältnisse vorfindet wie bei den einfachen binären Suchbäumen.

Gesucht ist ein Ausdruck $C_k(n)$ für die erwartete Anzahl von Vergleichen. Die Herleitung des Ergebnisses, die auch Bentley in [Ben75] beschreibt, soll hier kurz skizziert werden. Als Voraussetzung werden folgende Verteilungsannahmen gemacht:

1. Alle $k$ Komponenten sind voneinander unabhängig.

2. Alle $k$ Komponenten sind gleichverteilt, so daß gilt:
   Die Wahrscheinlichkeit, daß die $i$-te Komponente des $m$-ten Tupels ($1 \leq i \leq k$, $1 \leq m \leq n$)

in einer beliebigen Teilmenge $A$ aller $i$-ten Komponenten, also $A \subset \{a_i^{(m)} | \ 1 \leq m \leq n\}$, das $j$-kleinste Element ist, ist für alle $j$ $(1 \leq j \leq |A|)$ gleich groß, nämlich $1/|A|$.

3. Damit die Bezeichnung „$j$-kleinstes Element“ eindeutig ist, seien die $n$ Werte einer Komponente jeweils paarweise verschieden, also $|\{a_i^{(m)} | \ 1 \leq m \leq n\}| = n$ für alle $i$ $(1 \leq i \leq k)$.

Wird nun an einem inneren Knoten nach der $i$-ten Komponente partitioniert und als Partitionswert demzufolge $a_i^{(m)}$ $(1 \leq m \leq n)$ aus $A$ zufällig ausgewählt, so ist nach der oben gemachten Verteilungsannahme die Wahrscheinlichkeit $1/n$, daß dieses $a_i^{(m)}$ das $j$-kleinste Element $(1 \leq j \leq n)$ von allen $n$ Werten in der $i$-ten Komponente ist. Folglich werden mit der Wahrscheinlichkeit $1/n$ im linken Teilbaum des Knotens $j$ Tupel und im rechten Teilbaum $n-j$ Tupel gespeichert. Für diese $j$ bzw. $n-j$ Tupel gelten die Verteilungsannahmen ebenfalls. Damit ergibt sich für $C_k$ folgende Rekursionsgleichnug, wobei der Term $n$ noch die Vergleiche der $n$ Tupel mit dem Partitionswert des betrachteten Knotens berücksichtigt:

$$C_k(n) = n + \frac{1}{n}\sum_{j=1}^{n-1} C_k(j) + \frac{1}{n}\sum_{j=1}^{n-1} C_k(n-j), \qquad n \in \mathcal{N}_+, \ C_k(1) = 0$$

Wie man sieht, ist dieses Rekursionsschema *unabhängig von* $k$, so daß hier die gleichen Resultate wie im eindimensionalen Fall gelten. Man kann nun z.B. wie in [AHU74] vorgehen und obige Gleichung umformen zu

$$C_k(n) = n + \frac{2}{n}\sum_{j=2}^{n-1} C_k(j), \qquad n \in \mathcal{N}_+, \ C_k(1) = 0, \tag{7.1}$$

und dann durch Induktion über $n$ folgende Abschätzung zeigen

$$C_k(n) \leq \ln(4) \cdot n \cdot \log(n), \qquad n \in \mathcal{N}_+, \tag{7.2}$$

wobei $\ln(4)$ die kleinstmögliche Konstante mit dieser Eigenschaft ist.

Somit liegt die erwartete Anzahl von Vergleichen zur Konstruktion von $k$-d-Bäumen in der Größenordnung $\Theta(n \cdot \log n)$, und der zugehörige Proportionalitätsfaktor ist — in jeder Dimension $k \in \mathcal{N}_+$ — gleich $\ln(4)$.

### 7.2.2 Aufbau idealer Bäume

Die Laufzeit für Operationen in einem $k$-d-Baum hängt vor allem auch von der Höhe des Baumes ab. Deswegen ist es vorteilhaft, wenn die Struktur des Baumes relativ ausgeglichen ist, so daß alle Blätter möglichst auf benachbarten Levels liegen.

K. Mehlhorn definiert in [Meh84b] solche idealen $k$-d-Bäume als Verallgemeinerung der balancierten 1-dimensionalen Binärbäume.

**Definition (idealer $k$-d-Baum):** Sei $T$ ein $k$-d-Baum und $v$ ein Knoten in $T$.
(a) $S(v) := \{v' | \ v'$ ist ein Blattknoten im Teilbaum unterhalb des Knotens $v\}$
(b) $h(v) :=$ *Level* von $v$ in $T$
(c) $T$ ist ein *idealer k-d-Baum* $:\Longleftrightarrow$ $|S(v')| \leq |S(v)| \ / \ 2$ für alle Knoten $v \in T$ und alle Nachfolgeknoten $v'$ von $v$.

**Lemma 1** *Sei $T$ ein idealer k-d-Baum für die Menge $B$, $|B| = n$.*
*Dann gilt für jeden Knoten $v$ aus $T$:* $h(v) \leq \log(n) + 1$.
**Beweis:** (folgt aus der Definition eines idealen $k$-d-Baums).

**Satz 4** *Sei $B \subseteq W = W_1 \times \ldots \times W_k, \quad |B| = n$.*
*Der Aufwand für den Aufbau eines idealen k-d-Baumes für B ist von der Größenordnung $O(n \log n)$.*
**Beweis:** Der Beweis erfolgt durch die Angabe eines Algorithmus, der einen idealen $k$-d-Baum in $O(n \log n)$ Schritten erzeugt (vgl. hierzu [Meh84b]).

- Sei $B_j := \{x_j | \ (x_1, \ldots, x_k) \in B\}$ $(1 \leq j \leq k)$ eine *Multimenge*, die alle Werte in der $j$-ten Koordinate aus $B$ enthält.
- Ist $n = 1$, erzeuge für das Element von $B$ einen Blattknoten und gib diesen aus.
- Ist $n > 1$, dann erzeuge einen inneren Knoten und wähle als Diskriminator für diesen Knoten ein $i \in \{1, \ldots, k\}$. Bestimme den zugehörigen Partitionswert $p_i$ über den *Median* der Multimenge $B_i$ (die Bestimmung des Medians kann nach [Meh84a] in linearer Zeit erfolgen). Durch die Wahl des Medians gilt für die beiden Mengen $B_{\leq} := \{X \in B \mid x_i \leq p_i\}$ und $B_{>} := \{X \in B \mid x_i > p_i\}$ dann $|B_{\leq}| \leq |B|/2$ bzw. $|B_{>}| \leq |B|/2$ , so daß die Bedingung für ideale $k$-d-Bäume erfüllt ist.
  Wende diesen Algorithmus rekursiv auf die beiden Mengen $B_{\leq}$ und $B_{>}$ an und ordne die beiden dabei entstandenen idealen $k$-d-Bäume für $B_{\leq}$ und $B_{>}$ dem erzeugten inneren Knoten als dessen Teilbäume zu.

Durch diesen Algorithmus wird offensichtlich ein idealer $k$-d-Baum $T := T^{(i)}(B)$ generiert. Auch der in Abschnitt 7.1 konstruierte 2-d-Baum (Abbildung 7.1) ist ein solcher idealer Baum. Die Partitionswerte entsprechen jeweils gerade dem Median der in der zu partitionierenden Punktmenge auftretenden Werte des Diskriminatorattributes.

Die Laufzeit des Algorithmus läßt sich wie folgt abschätzen. An jedem Knoten $v$ aus $T$ müssen $O(|S(v)|)$ Schritte ausgeführt werden, um den Median einer Menge mit $|S(v)|$ Elementen zu bestimmen. Für zwei Baumknoten $v$ und $w$ gilt weiterhin $S(v) \cap S(w) = \emptyset$. Folglich ist $\sum_{h(v)=k} |S(v)| \leq n$ für jede Rekursionsstufe $k$ $(1 \leq k < \log(n) + 1)$. Die Laufzeit ist somit beschränkt durch

$$\begin{aligned} \sum_{v \in T} O(|S(v)|) &= O(\sum_{k=1}^{\log n} \sum_{h(v)=k} |S(v)|) \\ &= O(n \cdot \log n). \end{aligned} \tag{7.3}$$

## 7.3 Einfaches Suchen, Einfügen und Löschen

Die Grundoperationen auf $k$-d-Bäumen sind — wie bei anderen Suchstrukturen auch — die einfache Suche (Exact Match Query) sowie das Einfügen und Löschen von Tupeln. Auf die Realisierung von Best Match Queries in $k$-d-Bäumen, dem zentralen Gegenstand dieser Arbeit, gehen wir dann in den folgenden Kapiteln 8 und 9 näher ein. Algorithmen für Partial Match Queries und Region Queries sind in [Ben75] beschrieben. Sie werden in dieser Arbeit nicht näher betrachtet.

Gegeben sei wie in den vorangegangenen Abschnitten ein $k$-dimensionaler Datenraum $W$, eine $n$-elementige Menge $B \subseteq W$ und ein $k$-d-Baum $T := T^{(i)}(B)$ $(1 \leq i \leq k)$ mit der Bucketgröße $b$. Sei $X = (x_1, \ldots, x_k) \in B$ das Tupel, das in $T$ gesucht oder eingefügt bzw. aus $T$ gelöscht werden soll.

Bei allen drei Operationen muß zunächst der Baum $T$ durchlaufen werden, um das Bucket zu finden, in dessen Teilraum das Tupel $X$ liegt. Dies erfolgt durch rekursives Hinabsteigen im Baum. Ausgegeben wird die im aufgesuchten Bucket gespeicherte Tupelmenge.

Als erstes wird die Wurzel $v$ von $T^{(i)}(B)$ betrachtet. Ist sie ein Blattknoten, so werden alle Tupel aus ihrem Bucket ausgegeben. Ist $v$ ein innerer Knoten, dann wird die $i$-te Komponente von $X$ mit dem in $v$ gespeicherten Partitionswert $p_i$ verglichen. Falls $x_i \leq p_i$ ist, wird der linke Sohn von $v$, der alle Tupel aus $B$ enthält, deren $i$-te Koordinaten kleiner oder gleich $p_i$ sind, rekursiv in

derselben Weise durchlaufen. Gilt $x_i > p_i$, so erfolgt dies entsprechend für den rechten Sohnknoten von $v$.

Die erforderliche Laufzeit für das Aufsuchen eines Buckets hängt von der Höhe des Baumes $T$ ab, und ist deshalb im schlechtesten Fall, wenn $T$ „entartet“ ist, nur in linearer Zeit möglich. Im optimalen Fall, wenn $T$ ein idealer $k$-d-Baum ist, liegt der Aufwand hingegen in der Größenordnung $O(\log n)$.

Wurde das Bucket für $X$ lokalisiert, so ist je nach Operation folgendermaßen fortzufahren (sei dabei $S$ die Menge aller Tupel des entsprechenden Blattknotens):

- **einfache Suche:** Falls $X$ in $S$ enthalten ist, gib $X$ aus.
- **Einfügen:** Falls $|S| < b$, füge $X$ in den zu $S$ gehörenden Blattknoten ein, sonst ersetze den Blattknoten durch einen $k$-d-Baum für die Menge $S \cup \{X\}$.
- **Löschen:** Ist $X$ in $S$ enthalten, entferne $X$ aus dem Blattknoten.
  Falls $S \setminus \{X\} = \emptyset$ , ersetze den Vater des Blattknotens durch seinen anderen Sohnknoten.

Beispiel:

Gegeben sei der (ideale) 2-d-Baum $T^{(1)}(B)$ aus Abschnitt 7.1 (Abb. 7.1), der bei einer Bucketgröße $b = 2$ die Menge $B = \{A, B, C, \ldots, I\}$ enthält.

1. *Auffinden* des Punktes $D(30, 20)$ in $T^{(1)}(B)$:
   Um das Bucket im Baum zu lokalisieren, das den Punkt $D$ enthält, sind folgende Aktionen durchzuführen:
   (a) Vergleich der 1. Koordinate von $D$ mit dem Partitionswert 35 der Wurzel von $T^{(1)}(B)$: Da $30 \leq 35$ gilt, wird die Suche im *linken* Teilbaum unterhalb der Wurzel fortgesetzt.
   (b) Vergleich der 2. Koordinate von $D$ mit dem Partitionswert 30 des linken Sohnes der Wurzel: Da $20 \leq 30$ gilt, wird die Suche im *linken* Teilbaum unterhalb dieses Sohnes fortgesetzt.
   (c) Vergleich der 1. Koordinate von $D$ mit dem Partitionswert 15 des linken Sohnes des Wurzelsohnes: Da $30 > 15$ gilt, wird die Suche im *rechten* Teilbaum des Knotens fortgesetzt.
   (d) Der zu durchsuchende Teilbaum ist nun der Blattknoten, dessen Bucket den Punkt $D$ enthält. Somit wird $D$ als Ergebnis der Suche ausgegeben.
2. *Einfügen* zweier neuer Punkte $X(30, 10)$ und $Y(40, 30)$ in $T^{(1)}(B)$ (Abb. 7.2(a)):
   Zunächst sind analog zu oben die beiden Buckets aufzusuchen, in denen $X$ bzw. $Y$ enthalten sein müssen und zu denen die Punkte dann hinzugefügt werden.
   - Einfügen von $X$ in $S = \{D\}$: Da $|S| < b$ ist, wird $X$ in das Bucket des Blattknotens, der den Punkt $D$ enthält, aufgenommen.
   - Einfügen von $Y$ in $S = \{H, I\}$: Da $|S| \geq b$ ist, wird der Blattknoten mit den Punkten $H$ und $I$ ersetzt durch den neuen Teilbaum $T^{(1)}(S \cup \{Y\})$. Der in der Wurzel dieses Teilbaumes abgespeicherte Partitionswert 65 ist der Median aus 40, 65 und 70, den Werten in der 1. Koordinate der drei Punkte $H$, $I$ und $Y$. Die Menge $S \cup \{Y\}$ wird somit auf zwei neue Buckets aufgeteilt, wobei der linke Blattknoten die Punkte $I$ und $Y$ enthält und das rechte Blatt den Punkt $H$.

   Wie man sieht, hat der durch das Einfügen der beiden Punkte neu entstandene 2-d-Baum immer noch eine balancierte Struktur.
3. *Löschen* der Punkte $F$, $G$, $H$ und $I$:
   Durch sukzessives Löschen der vier Punkte entsteht schließlich der nicht mehr ausbalancierte Baum aus Abbildung 7.2(b), dessen rechter Teilbaum nur noch aus dem Blattknoten mit dem Punkt $Y$ besteht.

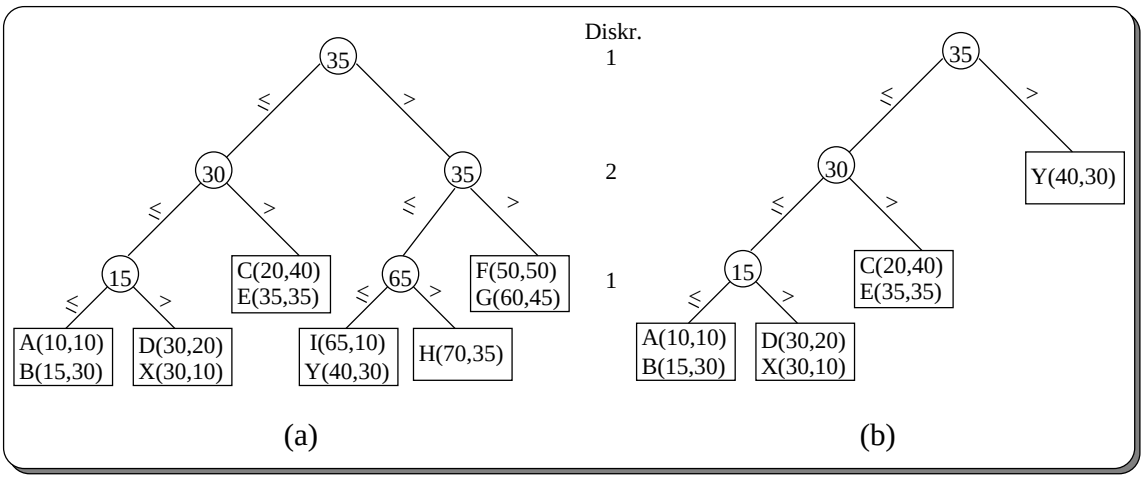


Abbildung 7.2: Suchen, Einfügen und Löschen von Punkten

Es ist klar, daß ein idealer Baum nach einem Einfüge– oder Löschvorgang i.a. nicht mehr ausgeglichen ist. Um das Gleichgewicht wieder herzustellen, ist eine Rebalancierung erforderlich. Dies kann im Gegensatz zu den 1-d-Bäumen jedoch nicht durch Rotationen erreicht werden, da die Partitionierungen im mehrdimensionalen Fall auf verschiedenen Koordinaten definiert sind.

K. Mehlhorn schlägt deshalb eine alternative Vorgehensweise für mehrdimensionale Suchbäume vor. Angenommen, $u = v_0, v_1, \ldots$ sei der Pfad von der Wurzel $v_0$ bis zu dem Blattknoten, dessen Bucketinhalt sich geändert hat, und $l$ sei der kleinste Index eines Knotens auf diesem Pfad, der nicht mehr ausbalanciert ist. Dann kann der $k$-d-Baum rebalanciert werden, indem man den Teilbaum mit der Wurzel $v_l$ in $u$ durch einen idealen $k$-d-Baum für die Tupelmenge $S(v_l)$ ersetzt.

Im obigen Beispiel würde dies bedeuten, daß der 2-d-Baum aus Abb. 7.2(b) für die Menge $\{A, B, C, D, E, X, Y\}$ gänzlich neu aufgebaut werden müßte. Der neue Partitionswert der Wurzel wäre dann 30.

Eine solche Rebalancierung kann nach [Meh84b] in $O(m \log m)$ Schritten durchgeführt werden, wobei $m = |S(v_l)|$. Folglich ist der Aufwand im schlechtesten Fall, wenn der gesamte $k$-d-Baum neu aufgebaut werden muß, von der Größenordnung $O(n \log n)$.

# Kapitel 8

# Starre NN–Suche

Die Datenstruktur des $k$-d-Baumes kann zur Unterstützung der Nearest–Neighbor–Suche herangezogen werden. Friedman et al. [FBF77] haben einen Algorithmus zur Lösung des MNN–Problems entwickelt, der unter Verwendung eines $k$-d-Baumes aus einer Menge $k$-dimensionaler Datensätze die $m$ nächsten Nachbarn zu einem vorgegebenen Anfragepunkt bestimmt. Dabei muß der Parameter $m$ für die MNN–Suche von vorneherein festgelegt werden.

Die Anfrage wird in der $k$-dimensionalen Repräsentation gestellt, in der auch die vorhandenen Datensätze oder Fälle beschrieben sind. Der in der Anfrage spezifizierte Datensatz muß dabei selbst nicht in der vorliegenden Satzmenge enthalten sein.

Sowohl beim Aufbau des $k$-d-Baumes als auch zur Bestimmung der besten Matches wird in der ursprünglichen Version von Friedman et al. ein Distanzmaß verwendet, das über eine Funktion auf der Summe von $k$ Koordinatendistanzfunktionen, die für die jeweiligen Wertebereiche der einzelnen Suchattribute definiert sind, berechnet wird.

Wir werden sehen, daß der Berechnungsaufwand für die Organisation von $n$ Fällen in einem optimierten (d.h. idealen) $k$-d-Baum proportional zu $kn \log n$ ist.

Die $k$-d-Baumstruktur partitioniert eine gegebene Datenmenge so, daß die Anzahl der bei der MNN–Suche im Durchschnitt zu untersuchenden Datensätze möglichst klein ist.

Die zu erwartende Anzahl der Fälle, die in jedem Suchvorgang untersucht werden müssen, ist unabhängig von der Größe der Fallbasis. Friedman et al. zeigen in [FBF77], daß der für die Suche in einem optimierten Baum benötigte Berechnungsaufwand durchschnittlich von der Größenordnung $O(\log n)$ ist. Auf die Frage, wodurch in diesem Zusammenhang ein optimierter $k$-d-Baum charakterisiert ist, werden wir im folgenden Kapitel eingehen.

Der Ansatz von Friedman et al. wurde insofern modifiziert, als wir anstelle der Distanzfunktionen Ähnlichkeitsmaße für den Baumaufbau und die Suche verwenden.

Wie in Kapitel 4 bereits erläutert, kann eine solche Modifikation von Distanz– zu Ähnlichkeitsmaßen durchgeführt werden, wenn zwischen beiden Maßen die Kompatibilitätsrelation besteht. Die für die Distanzfunktionen geltenden Aussagen können dann in analoger Weise auch auf die Ähnlichkeitsmaße übertragen werden. Auf die Voraussetzungen, die dazu im Hinblick auf die Maße erfüllt sein müssen, wird in Abschnitt 8.4 näher eingegangen.

## 8.1 Optimierter Aufbau eines k-d-Baumes

Ein Ziel bei der Suche nach den nächsten Nachbarn eines Falles in einem $k$-d-Baum ist die Minimierung der Anzahl zu untersuchender Fälle. Die Untersuchung eines Falles umfaßt hierbei jeweils folgende Schritte:

1. externer Speicherzugriff auf den Fall
2. Berechnung seiner Ähnlichkeit zum gesuchten Fall

3. Vergleich der Ähnlichkeit mit den Ähnlichkeiten bereits untersuchter Fälle
4. Aktualisierung einer Liste, die zu jedem Zeitpunkt der Suche alle bislang gefundenen nächsten Nachbarn enthält

Um dem angestrebten Ziel möglichst nahe zu kommen, ist es notwendig, den Baum so aufzubauen, daß die zu erwartende Menge der Fälle, die bei der Suche tatsächlich überprüft werden müssen, möglichst gering ist. Durch eine geschickte Anordnung der Fälle innerhalb des Baumes soll erreicht werden, daß in den einzelnen Schritten des Suchvorganges jeweils viele Baumknoten (und somit Fälle) schon von vornherein ausgeschlossen werden können.

Wie wir im folgenden sehen werden, bedeutet das für den Baum nichts anderes, als daß er eine (annähernd) balancierte Struktur besitzt, bei der sich alle Blattknoten im Baum auf benachbarten Levels befinden.

Wie sieht nun ein solcher für die NN–Suche idealer $k$-d-Baum aus? Die Parameter, die hierbei bestimmt werden müssen, sind jeweils das Diskriminatorattribut und der zugehörige Partitionswert an den inneren Knoten sowie für die Blattknoten die Bucketgröße, d.h. die maximale Anzahl von Fällen, die dort gespeichert sein können.

Im allgemeinen hängen diese Parameter von der Verteilung der in den Anfragen auftretenden Fälle im Datenraum ab. Diese Verteilung kennt man jedoch normalerweise nicht, bevor die Queries gestellt worden sind.

Deshalb gehen Friedman et al. so vor, daß sie für den Baumaufbau nur die Information nutzen, die durch die anzuordnende Menge der vorhandenen Fälle gegeben ist. Dadurch wird der Aufbau von der Verteilung der Anfragen unabhängig.

Eine weitere Einschränkung ist noch, daß sowohl die Wahl der Diskriminatoren als auch der Partitionswerte ausschließlich von den zu partitionierenden Fallmengen abhängt. Denn nur dann ist auch ein rekursives Vorgehen beim Baumaufbau möglich, und es kann eine globale Binärbaumoptimierung, die nach [HR76] ein NP–vollständiges Problem ist, vermieden werden.

### 8.1.1 Bestimmung der Partitionswerte

Die Information, die während der Suche an einem inneren Baumknoten zur Verfügung steht, ist durch die Aufteilung der Fallmenge in zwei Partitionen gegeben. Für eine binäre Suche ist die Information aber gerade dann am größten, wenn beide möglichen Alternativen in etwa gleich wahrscheinlich sind. Folglich sollte für jeden der Fälle die Wahrscheinlichkeit, einer der beiden Partitionen zugeordnet zu werden, gleich groß sein.

Um dies zu gewährleisten, wird deshalb unabhängig von der Wahl des Diskriminators als Partitionswert immer der *Median* der Werte herangezogen, die im Diskriminatorattribut der zu partitionierenden Fälle auftreten (vgl. Kapitel 7, Abschnitt 7.2.2).

### 8.1.2 Wahl des Diskriminators

Der Wahl des Diskriminators an einem inneren Knoten liegt folgende Überlegung zugrunde: Es kann diejenige der beiden Partitionen, die den gesuchten Fall *nicht* enthält, von der Suche ausgeschlossen werden, wenn sich in ihr kein Fall mehr befinden kann, der zum gesuchten Fall ähnlicher ist als der aktuelle $m$-te nächste Nachbar.

Dies wird durch einen Test überprüft, der feststellt, ob sich der durch die Partition repräsentierte Bereich des Datenraumes mit der Region um den gesuchten Fall überlappt, die alle bislang gefundenen nächsten Nachbarn enthält. Wie wir noch sehen werden, entspricht diese Region einer $k$-dimensionalen Kugel.

Per Definition ist der Radius dieser Kugel entlang jeder Koordinatenachse gleich groß. Die Wahrscheinlichkeit, daß sich eine Partition mit der Kugel überlappt, ist (gemittelt über alle möglichen Anfragepunkte) dann am geringsten, wenn nach dem Attribut partitioniert wird, bei dem die in den Fällen auftretenden Werte die größte Streuung aufweisen.

Im Ansatz von Friedman et al. in [FBF77] wird die Streuung entlang jeder Koordinatenachse über die Varianz der auftretenden Werte geschätzt. Die Berechnung der Varianz ist so allerdings nur bei quantitativen Attributen möglich.

In dieser Diplomarbeit gehen wir jedoch davon aus, daß Fälle i.a. nicht nur durch quantitative Merkmale beschrieben sind. Was aber vorausgesetzt werden muß, ist eine Ordnung auf den Attribut–Wertebereichen; zum einen, um den Median zu ermitteln und zum anderen, damit die für die Suche erforderlichen Vergleiche an den Knoten durchgeführt werden können.

### Das Maß des Interquartilsabstandes

Zur Schätzung der Streuung verwenden wir in dieser Arbeit das statistische Maß des *Interquartilsabstandes* (vgl. z.B. [Koo87]), das sowohl bei quantitativen als auch bei Rangmerkmalen verwendet werden kann.

Zerteilt der Median eine vorliegende Verteilung von Werten in genau zwei Hälften, so zerlegen die Quartile diese wiederum in Viertel (Abb. 8.1). Das erste Quartil $q_1$ (25%-Quartil) teilt die untere Hälfte der Verteilung in zwei gleich große Bereiche und das dritte Quartil $q_3$ (75%-Quartil) entsprechend die obere Hälfte. Der Median wird als das zweite Quartil bezeichnet. Das Streuungsmaß des Interquartilsabstandes *iqr* ergibt sich aus der Distanz zwischen dem ersten und dem dritten Quartil.

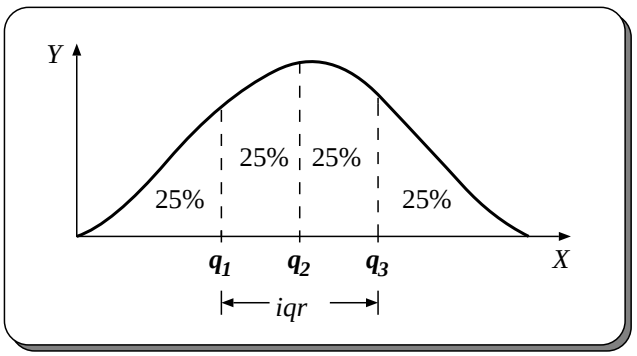


Abbildung 8.1: Interquartilsabstand

Die Position des Medians in einer geordneten Reihe von $n$ Werten ist $mloc = \lfloor \frac{n+1}{2} \rfloor$. Die Position der beiden anderen Quartile läßt sich über den Median (der ja auch für die Bestimmung des Partitionswertes gebraucht wird) bestimmen durch

$$l = \frac{mloc + 1}{2} \tag{8.1}$$

$q_1$ ist dann der $l$-kleinste Wert der geordneten Reihe und $q_3$ analog der $l$-größte. Die größte Streuung weisen somit die Attributwerte auf, deren Quartile am weitesten voneinander entfernt liegen.

### Schätzung der Streuung über Ähnlichkeitsmaße

In unserem Falle haben wir keine Distanzfunktionen, sondern Ähnlichkeitsmaße für die einzelnen Wertebereiche vorgegeben, die in die Schätzung der Streuung und somit in den Aufbau des $k$-d-Baumes miteingehen.

Deshalb modifizieren wir die Bestimmung der Streuung etwas, indem wir sie über die *Ähnlichkeit* zwischen dem ersten und dem dritten Quartil berechnen. Als Diskriminator wählen wir das Attribut mit der größten Streuung, also dasjenige, dessen Quartile *am wenigsten zueinander ähnlich* sind.

Die Streuung läßt sich demnach durch den Wert $iqr = \mu(q_1, q_3)$ abschätzen, wobei $\mu$ das auf den Attributwerten definierte Ähnlichkeitsmaß sei. Ausgewählt wird dann das Attribut mit dem kleinsten Wert für $iqr$.

### 8.1.3 Die Prozedur

Sei $S$ die Menge, für die ein optimierter $k$-d-Baum mit der Bucketgröße $b$ aufgebaut werden soll. Ist $|S| \leq b$, so wird ein Blattknoten erzeugt, der alle Fälle aus $S$ enthält. Ist $|S| > b$, dann wird ein innerer Knoten generiert. Dabei sind folgende Schritte durchzuführen:

Seien $q_1^{(i)}$ und $q_3^{(i)}$ die Quartile für die in $S$ vorhandenen Werte des Attributes $A_i$ ($i \in \{1, .., k\}$) und seien $iqr^{(i)}$ die entsprechenden Interquartilsabstände.

(a) Wahl des *Diskriminators* $d \in \{1, .., k\}$:
   1. berechne für jedes Attribut $A_i$: $iqr^{(i)} := \mu_i(q_1^{(i)}, q_3^{(i)})$
   2. $d$:=$j$, mit $j \in \{1, .., k\}$ und $iqr^{(j)} = min\{iqr^{(i)}\}_{i=1}^{k}$

(b) *Partitionswert* $p$ := Median der Multimenge $\{a_j \mid (a_1, .., a_k) \in S \ \wedge \ j = d\}$

(c) Erzeuge optimierte Bäume für die Partitionen
   - $S_{\leq} = \{(a_1, \ldots, a_k) \in S \mid a_d \leq p\}$
   - $S_{>} = \{(a_1, \ldots, a_k) \in S \mid a_d > p\}$

In einer rekursiven Prozedur läßt sich der Aufbau eines idealen Baumes dann wie folgt beschreiben:

```
procedure BUILD_TREE(setOfData);
local j, disc, minSimilarity, p;
begin
  if Size(setOfData) ≤ b then return MAKE_TERMINAL_NODE(setOfData);
  minSimilarity:=∞;
  for all coordinates A_j (1 ≤ j ≤ k) do
    if SPREAD(A_j,setOfData) < minSimilarity then
    begin
      minSimilarity:=SPREAD(A_j,setOfData);
      disc:=j
    end;
  p:=MEDIAN(disc,setOfData);
  return
    MAKE_NONTERMINAL_NODE
      (disc,p,
      BUILD_TREE(LEFT_SUBFILE(disc,p,setOfData)),
      BUILD_TREE(RIGHT_SUBFILE(disc,p,setOfData))
      );
end BUILD_TREE.
```

Die Prozedur SPREAD($A_j$,*setOfData*) bestimmt für die Datenmenge *setOfData* die Streuung der Werte von Attribut $A_j$ über den Interquartilsabstand. Die Prozedur MEDIAN(*disc*,*setOfData*) berechnet den Median der durch die Datenmenge *setOfData* gegebenen Werte des Diskriminatorattributes *disc*. LEFT_SUBFILE und RIGHT_SUBFILE erzeugen die beiden Partitionen von *setOfData* bzgl. des Diskriminators *disc* und des Partitionswertes $p$. MAKE_TERMINAL_NODE und MAKE_NONTERMINAL_NODE generieren Blatt– bzw. innere Knoten, die sie mit ihren Aufrufparametern belegen.

Jeder Blattknoten enthält in seinem *Bucket* eine Menge von höchstens $b$ Datensätzen. Ein innerer Knoten beinhaltet seinen *Diskriminator*, den zugehörigen *Partitionswert* sowie zwei Zeiger auf seinen linken und seinen rechten Nachfolgeknoten (*leftSon* bzw. *rightSon*).

### 8.1.4 Beispiel

Die beschriebene Vorgehensweise soll an einem kleinen 2-dimensionalen Beispiel veranschaulicht werden. Das Beispiel, für das die wesentlichen Schritte beim Aufbau eines optimierten Baumes durchgeführt werden, wurde aus darstellungstechnischen Gründen einfach gehalten. Wir betrachten Fälle, deren Repräsentationen aus zwei Attributen $A_1$ und $A_2$ bestehen:

- $A_1$ ist ein *numerisches* Attribut mit dem Wertebereich $W_1 = \mathcal{N}$ und der üblichen Ordnung auf den natürlichen Zahlen. Das Ähnlichkeitsmaß für $W_1$ sei durch das zur euklidischen Distanzfunktion kompatible Maß $\mu_1(x, y) = 1/(1 + |x - y|)$ $(x, y \in \mathcal{N})$ definiert.
- $A_2$ ist ein *symbolisches* Attribut mit dem Wertebereich $W_2 = \{$*weiß,grau,schwarz*$\}$. Die Ordnung auf den Elementen von $W_2$ sei gegeben durch die Relationen *weiß*<*grau*<*schwarz*, und das Ähnlichkeitsmaß $\mu_2$ für diese Menge durch folgende Matrix:

| | *weiß* | *grau* | *schwarz* |
|---|---|---|---|
| *weiß* | 1 | 0.25 | 0 |
| *grau* | 0.25 | 1 | 0.5 |
| *scwarz* | 0 | 0.5 | 1 |

(Im weiteren werden die Werte *weiß, grau* und *schwarz* durch die Symbole *w, g* bzw. *s* abgekürzt dargestellt.)

Es sei $\mathcal{B} = \{A, B, C, D, E\} \subseteq \{(a_1, a_2) \mid a_1 \in W_1,\ a_2 \in W_2\}$ die Menge der Fallbeispiele, die in einem für die Ähnlichkeitssuche optimierten 2-d-Baum der Bucketgröße $b = 1$ angeordnet werden sollen. Die Ausprägungen der 5 Fallrepräsentationen sind in nachfolgender Tabelle dargestellt:

| | $A$ | $B$ | $C$ | $D$ | $E$ |
|---|---|---|---|---|---|
| $A_1$ | 6 | 1 | 1 | 2 | 4 |
| $A_2$ | $s$ | $w$ | $g$ | $w$ | $s$ |

#### Das Vorgehen beim Aufbau des Baumes

Nach Abschnitt 8.1.3 besteht der Aufbau eines optimierten 2-d-Baumes für die Menge $\mathcal{B}$ aus folgenden Schritten:

- **Schritt I:** Generierung des Wurzelknotens
  $S = \mathcal{B}$, $|S| = 5 \Longrightarrow$ erzeuge *inneren* Knoten:

| Attribut | Werte | Median | $q_1$ | $q_3$ | $iqr$ |
|---|---|---|---|---|---|
| $A_1$ | 1, 1, 2, 4, 6 | 2 | 1 | 4 | 1/4 |
| $A_2$ | $w$, $w$, $g$, $s$, $s$ | $g$ | $w$ | $s$ | 0 |

  Knoteninhalt: $d := 2$, $p := g$, $S_{\leq} := \{B, C, D\}$, $S_{>} := \{A, E\}$

- **Schritt II:** Generierung des *rechten* Wurzelsohnes
  $S = \{A, E\}$, $|S| = 2 \Longrightarrow$ erzeuge *inneren* Knoten:

| Attribut | Werte | Median | $q_1$ | $q_3$ | $iqr$ |
|---|---|---|---|---|---|
| $A_1$ | 4, 6 | 4 | 4 | 6 | 1/3 |
| $A_2$ | $s$, $s$ | $s$ | $s$ | $s$ | 1 |

  Knoteninhalt: $d := 1$, $p := 4$, $S_{\leq} := \{E\}$, $S_{>} := \{A\}$

- **Schritt III:** Generierung eines Blattknotens für $S = \{A\}$ als *rechten* Sohn des Knotens aus Schritt II
- **Schritt IV:** Generierung eines Blattknotens für $S = \{E\}$ als *linken* Sohn des Knotens aus Schritt II
- **Schritt V:** Generierung des *linken* Wurzelsohnes
  $S = \{B, C, D\}$, $|S| = 3 \Longrightarrow$ erzeuge *inneren* Knoten:

| Attribut | Werte | Median | $q_1$ | $q_3$ | *iqr* |
|---|---|---|---|---|---|
| $A_1$ | 1, 1, 2 | 1 | 1 | 2 | 1/2 |
| $A_2$ | *w, w, g* | *w* | *w* | *g* | 0.25 |

  Knoteninhalt: $d := 2, \quad p := w, \quad S_{\leq} := \{B, D\}, \quad S_{>} := \{C\}$

- **Schritt VI:** Generierung eines Blattknotens für $S = \{C\}$ als *rechten* Sohn des Knotens aus Schritt V
- **Schritt VII:** Generierung des *linken* Sohnes für den Knoten aus Schritt V
  $S = \{B, D\}$, $|S| = 2 \Longrightarrow$ erzeuge *inneren* Knoten:

| Attribut | Werte | Median | $q_1$ | $q_3$ | *iqr* |
|---|---|---|---|---|---|
| $A_1$ | 1, 2 | 1 | 1 | 2 | 1/2 |
| $A_2$ | *w, w* | *w* | *w* | *w* | 1 |

  Knoteninhalt: $d := 1, \quad p := 1, \quad S_{\leq} := \{B\}, \quad S_{>} := \{D\}$

- **Schritt VIII:** Generierung eines Blattknotens für $S = \{D\}$ als *rechten* Sohn des Knotens aus Schritt VII
- **Schritt IX:** Generierung eines Blattknotens für $S = \{B\}$ als *linken* Sohn des Knotens aus Schritt VII

Der durch die Schritte I - IX erzeugte 2-d-Baum ist in Abbildung 8.2 dargestellt. Die Kreise stehen für die inneren Knoten. Dabei bezeichnet jeweils der obere Wert den Diskriminator und der untere Wert den Partitionswert. Die Rechtecke stellen die generierten Blattknoten mit ihren jeweiligen Fällen dar. Die (römische) Numerierung der Knoten entspricht der Reihenfolge ihrer oben beschriebenen Generierung.

## 8.2 Der Algorithmus

### 8.2.1 Rekursive Baumsuche

Der $k$-d-Baum, in dem die Menge aller bekannten Fälle organisiert ist, wird beginnend bei seiner Wurzel rekursiv nach den $m$ nächsten Nachbarn durchsucht. Gegeben sind dabei

- die in der Anfrage spezifizierten $k$ Suchschlüsselwerte
- die Anzahl $m$ der zu bestimmenden nächsten Nachbarn
- der $k$-d-Baum, repräsentiert durch seinen Wurzelknoten *root*

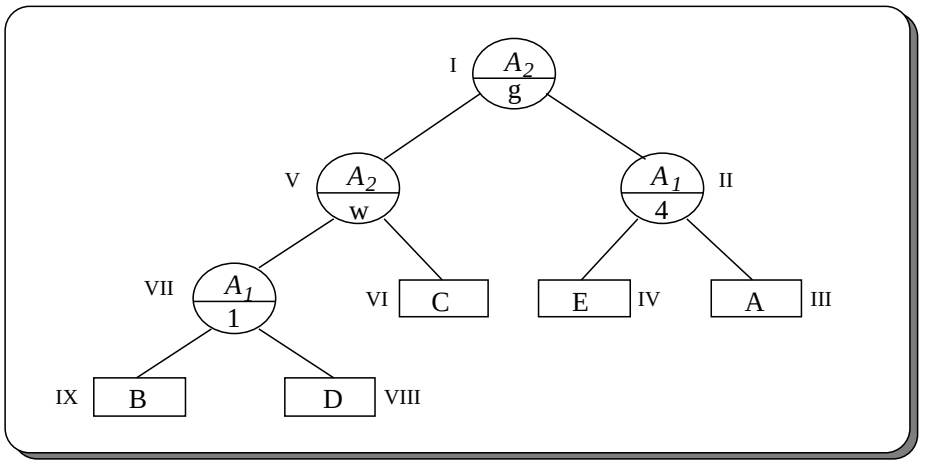


Abbildung 8.2: Optimierter $k$-d-Baum

Während der Suche wird eine ständig aktualisierte Liste in Form einer *Prioritäten–Queue* geführt, die zu jedem Zeitpunkt der Suche die bislang bestimmten $m$ nächsten Nachbarn und deren berechnete Ähnlichkeiten enthält.

Die Fälle sind in der Queue absteigend nach ihren Ähnlichkeiten angeordnet, so daß der erste Fall dieser Liste immer der aktuelle nächste Nachbar ist und der letzte Listeneintrag entsprechend der $m$-te Nachbar. Sobald ein Fall untersucht wird, dessen Ähnlichkeit größer ist als die des $m$-ten nächsten Nachbarn, wird er neu in die Liste aufgenommen und der letzte Eintrag aus der Queue entfernt.

Die rekursive Prozedur, die diesen Algorithmus beschreibt, erhält als Eingabeparameter den zu betrachtenden Baumknoten. Beim ersten Aufruf dieser Prozedur wird ihr als Argument die Wurzel des Baumes übergeben.

Falls der gerade betrachtete Knoten ein Blattknoten ist, werden alle in diesem Bucket enthaltenen Fälle auf ihre Ähnlichkeit zum gesuchten Fall hin untersucht und die Queue wie oben beschrieben aktualisiert. Ist der betrachtete Knoten kein Blatt sondern ein innerer Knoten, dann wird die Prozedur rekursiv für denjenigen der beiden Sohnknoten aufgerufen, in dessen Partition der in der Query spezifizierte Fall enthalten sein müßte.

Nach der Abarbeitung des Aufrufes wird getestet, ob es erforderlich ist, den anderen noch nicht betrachteten Sohnknoten ebenfalls zu durchsuchen.

Dies erfolgt durch den sog. *Bounds–Overlap–Ball*–Test. Schlägt der Test fehl, so kann die Partition des anderen Sohnes keinen der $m$ nächsten Nachbarn enthalten und wird demnach auch nicht mehr untersucht. Ansonsten wird die Prozedur auch noch für diesen Sohn rekursiv aufgerufen.

Am Ende der Prozedur wird schließlich anhand des *Ball–Within–Bounds*–Tests überprüft, ob bereits alle $m$ nächsten Nachbarn gefunden sind oder ob die Suche nach der Rückkehr aus dem Rekursionsaufruf fortgesetzt werden muß.

In einer algorithmischen Notation sieht die Prozedur folgendermaßen aus:

**Globale Variablen**

| | |
|---|---|
| $X_q[1..k]$ | enthält die $k$ Suchattributwerte des in der Anfrage spezifizierten Falles; |
| $PQC[1..m]$ | Queue, die zu jedem Zeitpunkt der Suche die bisher gefundenen $m$ besten Matches enthält ($PQC[1]$ ist der aktuelle nächste Nachbar); |
| $PQS[1..m]$ | Queue der absteigend sortierten Ähnlichkeiten der aktuellen nächsten Nachbarn ($PQS[1]$ ist die Ähnlichkeit von $PQC[1]$ zu $X_q$); |

*Upper*[1..*k*] enthält die oberen Koordinatenschranken des betrachteten Knotens;
*Lower*[1..*k*] enthält die unteren Koordinatenschranken des betrachteten Knotens;

```
procedure SEARCH(node);
local p, d, temp;
begin
  if isTerminal(node) then (* node ist ein Blattknoten *)
  begin
    "untersuche die Fälle in node.bucket und aktualisiere PQC, PQS";
    (* Test, ob Suche beendet werden kann *)
    if BALL–WITHIN–BOUNDS then done else return
  end;
  d:=node.discriminator;
  p:=node.partitionValue;
  (* rekursiver Aufruf für den Sohn, der X_q enthält *)
  if X_q[d] ≤ p then
  begin
    temp:=Upper[d];   Upper[d]:=p;
    SEARCH(node.leftSon);
    Upper[d]:=temp;
  end
  else begin
    temp:=Lower[d];   Lower[d]:=p;
    SEARCH(node.rightSon);
    Lower[d]:=temp;
  end
  (* rekursiver Aufruf für den anderen Sohn *)
  if X_q[d] ≤ p then
  begin
    temp:=Lower[d];   Lower[d]:=p;
    if BOUNDS–OVERLAP–BALL then SEARCH(node.rightSon);
    Lower[d]:=temp;
  end
  else begin
    temp:=Upper[d];   Upper[d]:=p;
    if BOUNDS–OVERLAP–BALL then SEARCH(node.leftSon);
    Upper[d]:=temp;
  end;
  (* Test, ob Suche beendet werden kann *)
  if BALL–WITHIN–BOUNDS then done else return;
end SEARCH.
```

Ein Suchaufruf setzt sich nun aus folgenden Schritten zusammen:

1. Eingabe der Anfragewerte in das globale Array $X_q$;
2. Initialisierung: $PQS[1..m] := -\infty$;
   $Upper[1..k] := \infty$; $Lower[1..k] := -\infty$;
   $root :=$ "Wurzelknoten des $k$-d-Baums";
3. Aufruf: SEARCH(*root*);

### 8.2.2 Obere und untere Schranken der Knoten

Die beiden oben erwähnten Bounds–Tests werden detailliert in Abschnitt 8.3 erläutert. Die Idee, die ihnen zugrunde liegt, ist folgende:

An jedem Knoten erfolgt durch die Partitionierung eine Einschränkung des Datenraumes. Der durch den Knoten repräsentierte Teilraum, und somit die Menge der möglichen Fälle, ist in jeder Dimension durch geometrische Schranken (engl. *bounds*) sowohl nach unten als auch nach oben begrenzt. Diese Schranken ergeben sich aus den Partitionen, die durch die Knoten, die sich als Vorgängerknoten auf dem Pfad von der Wurzel zum betrachteten Knoten befinden, definiert sind.

An jedem Knoten teilt die Partitionierung also nicht nur eine Menge von Fällen in zwei disjunkte Teilmengen auf, sondern definiert durch den Diskriminator und den Partitionswert gleichzeitig auch eine obere bzw. untere Schranke für die Werte, die das Diskriminatorattribut der Fälle in den beiden neu entstandenen Partitionen enthalten kann.

Die durch die Partitionswerte beschriebenen Hyperebenen stehen senkrecht auf den Koordinatenachsen und teilen den unendlichen Datenraum in ein mehrdimensionales Raster ein. Jede einzelne Zelle diese Gitters begrenzt dabei genau einen der Teilräume, die durch die Knoten definiert werden.

Anschaulich kann man sich das so vorstellen, daß jeder Teilraum von einem mehrdimensionalen Würfel (der sog. *bounding box*) umschlossen wird. Dabei wird jeder Würfel durch eine rekursive Aufteilung in den Nachfolgeknoten weiter zerlegt, so daß das Volumen der Teilräume im Baum von Level zu Level kleiner wird.

Seien $Upper(P)$ und $Lower(P)$ zwei $k$-dimensionale Arrays, deren $j$-te Komponenten ($1 \leq j \leq k$) gerade die obere bzw. untere Schranke enthalten, die am Knoten $P$ für das $j$-te Suchschlüsselattribut $A_j$ besteht. Seien weiterhin $A_d$ das Diskriminatorattribut und $partVal(P)$ der Partitionswert von $P$. Dann gilt für $1 \leq j \leq k$

1. am **Wurzelknoten** $R$:

   $$Upper(R)[j] = \infty$$

   und

   $$Lower(R)[j] = -\infty$$

2. am **linken Sohn** $L$ eines Knotens $P$:

   $$Lower(L)[j] = Lower(P)[j]$$

   und

   $$Upper(L)[j] = \begin{cases} Upper(P)[j] & , \quad j \neq d \\ partVal(P) & , \quad j = d \end{cases}$$

3. am **rechten Sohn** $R$ eines Knotens $P$:

   $$Upper(R)[j] = Upper(P)[j]$$

   und

   $$Lower(R)[j] = \begin{cases} Lower(P)[j] & , \quad j \neq d \\ partVal(P) & , \quad j = d \end{cases}$$

## 8.3 Die Bounds–Tests

Das Ziel der beiden logischen Tests *Bounds–Overlap–Ball* und *Ball–Within–Bounds* aus dem obigen Suchalgorithmus ist die Einschränkung der zur Bestimmung der nächsten Nachbarn erforderlichen Untersuchungen von Knoten des $k$-d-Baumes.

Mit Hilfe der globalen Queue, die während der Suche die jeweils aktuellen $m$ NN–Kandidaten enthält, können Aussagen darüber gemacht werden, ob ein Durchsuchen der Fallmenge des gerade zu betrachtenden Knotens überhaupt neue NN–Kandidaten liefern kann oder nicht.

Die räumliche Nähe zwischen zwei Punkten wird über ein Ähnlichkeitsmaß bestimmt. Die einzelnen durch den Knoten repräsentierten Fälle können von vorneherein von der Untersuchung ausgeschlossen werden, wenn es unmöglich ist, daß auch nur einer von ihnen eine größere Ähnlichkeit zum gesuchten Fall hat als der letzte Fall in der Queue, der den aktuellen $m$-ten nächsten Nachbarn bezeichnet.

Die beiden Tests überprüfen dies, ohne dabei die einzelnen Fälle selbst zu untersuchen. Die Auswertungen der Tests können jeweils im Hauptspeicher erfolgen, da sie nur auf die globalen Variablen, wie sie im oben aufgeführten Algorithmus auftreten, zurückgreifen. Somit sind keine Externspeicherzugriffe auf die abgespeicherten Fälle erforderlich, so daß die Durchführung dieser Tests billiger ist als eine jeweilige Überprüfung der tatsächlich abgespeicherten Fälle.

### 8.3.1 Grundlegende Idee

Die zentrale Idee, die beiden Tests zugrunde liegt, läßt sich geometrisch folgendermaßen veranschaulichen:

- Der gesuchte Fall $X_q$ wird als Mittelpunkt einer $k$-dimensionalen Kugel aufgefaßt, deren Radius gerade so groß ist, daß sie alle aktuellen $m$ nächsten Nachbarn von $X_q$ umfaßt und der $m$-te Nachbar ein Punkt auf der Oberfläche dieser Kugel ist (Abbildung 8.3 zeigt dies für $m = 4$ und 2 Dimensionen). Alle weiteren in der Queue $PQC$ enthaltenen nächsten Nachbarn liegen dann entweder innerhalb der Kugel oder genau auf ihrer Oberfläche.

- Ein beliebiger Punkt $X$ des Datenraumes kommt nur dann als ein weiterer Kandidat für die Queue in Frage, wenn er von $X_q$ nicht weiter entfernt liegt als der aktuelle $m$-te nächste Nachbar $PQC[m]$.
  Oder mit anderen Worten: wenn seine Ähnlichkeit zu $X_q$ mindestens so groß ist wie $PQS[m]$, die Ähnlichkeit jenes $m$-ten Nachbarn. Dies wiederum ist nur dann der Fall, wenn $X$ nicht außerhalb der Kugel um $X_q$ liegt.

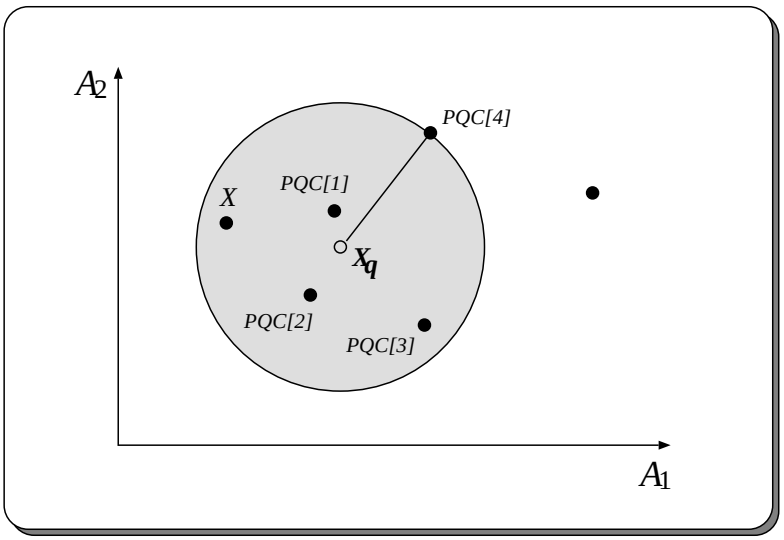


Abbildung 8.3: Idee der Bounds–Tests

Für jeden Knoten sind dessen geometrische Schranken bezüglich der $k$ Suchattributwerte und somit die Menge aller möglichen Fälle bekannt, die in den Blättern unter diesem Knoten im

Baum enthalten sein können. Es wird überprüft, ob sich der durch die geometrischen Ober– und Untergrenzen (den *Bounds*) des zu untersuchenden Knotens definierte $k$-dimensionale Teilraum mit der $k$-dimensionalen Kugel um $X_q$ (in [FBF77] als *Ball* bezeichnet) schneidet.

In Abhängigkeit von den geometrischen Schranken des Knotens werden dazu einige repräsentative (Test–)Punkte aus dem entsprechenden $k$-dimensionalen Teilraum betrachtet. Deren Ähnlichkeiten zum gesuchten Fall $X_q$ werden berechnet und mit der des $m$-ten nächsten Nachbarn verglichen. Ist die Ähnlichkeit eines dieser Testpunkte größer als die des $m$-ten nächsten Nachbarn, so befindet sich der Punkt innerhalb der Kugel, und es liegt eine Überlappung des Teilraumes mit der Kugel vor.

In der ursprünglichen Version von Friedman et al. ([FBF77]) kann der Kugelradius durch die Verwendung von Distanzmaßen direkt bestimmt und zu den Tests herangezogen werden. Da der in dieser Diplomarbeit verfolgte Ansatz jedoch auf Ähnlichkeitsmaßen aufbaut, muß die Realisierung dieser Tests etwas modifiziert werden. Die beiden folgenden Abschnitte erläutern dies genauer.

### 8.3.2 Bounds–Overlap–Ball

Der *Bounds–Overlap–Ball*–Test (*BOB*) liefert den Wert True, wenn die Fälle des aktuell betrachteten Baumknotens noch weiter untersucht werden müssen.

Dies ist dann erforderlich, wenn sich der kleinste Würfel, der den durch die geometrischen Schranken dieses Knotens beschriebenen Teilraum ganz umfaßt, mit der Kugel um $X_q$ schneidet und sich der Teilraum somit mit der Kugel überlappt. In diesem Fall enthält der Knoten möglicherweise einen Punkt, der gerade in diesem Überlappungsbereich liegt und deshalb ein NN–Kandidat sein könnte.

Die Realisierung des *BOB*–Tests ist relativ einfach: Man betrachtet lediglich einen bestimmten Punkt aus dem Datenraum $K$ des Knotens. Dieser Punkt $X_{min}$ ist derjenige, der von allen Punkten des Teilraumes am nächsten zu $X_q$ gelegen ist. Liegt $X_{min}$ nun innerhalb der Kugel, so ist eine Überlappung vorhanden. Liegt er jedoch außerhalb, so kann wegen der Wahl von $X_{min}$ auch keiner der anderen Punkte des Teilraumes innerhalb der Kugel liegen.

In dem 2-dimensionalen Beispiel in Abbildung 8.4 liegt ein Schnitt zwischen dem Teilraum $K_1$ und dem Kreis um $X_q$ vor. Dagegen überlappt sich der Kreis nicht mit den Teilräumen $K_2$ und $K_3$, da $X_{min}^{(2)}$ und $X_{min}^{(3)}$ außerhalb des Kreises liegen. Folglich enthalten diese beiden Teilräume auch keinen Fall, der ähnlicher ist als der aktuelle $m$-te nächste Nachbar, so daß die aufwendige Untersuchung jedes einzelnen Falles in diesen Teilräumen nicht erforderlich ist.

$X_{min}$ läßt sich anhand der oberen und unteren Schranken bestimmen, die in den globalen Arrays *Upper* und *Lower* für jeden gerade betrachteten Knoten jeweils vorliegen:

$$X_{min} := (x_1, x_2, \ldots, x_k)$$

mit

$$x_j = \begin{cases} Upper[j] & , \quad X_q[j] > Upper[j] \\ Lower[j] & , \quad X_q[j] < Lower[j] \\ X_q[j] & , \quad \text{sonst} \end{cases}$$

für $1 \leq j \leq k$.

Für $X_{min}$ gilt dann offensichtlich:

1. $X_{min}$ ist ein Punkt auf einer Kante des Würfels (*Bounding Box*), der die Region des Teilraumes umschließt; dabei liegt $X_{min}$ auf der $X_q$ zugewandten Kante

2. falls sich $X_q$ in keiner der $k$ Dimensionen mit der Bounding Box überlappt (d.h. $X_q[j] < Lower[j]$ oder $X_q[j] > Upper[j]$ für alle $j$), dann ist $X_{min}$ ein Eckpunkt der Box

3. liegt $X_q$ dagegen innerhalb der Box (also $Lower[j] \leq X_q[j] \leq Upper[j]$ für jedes $j$), dann gilt $X_{min} = X_q$

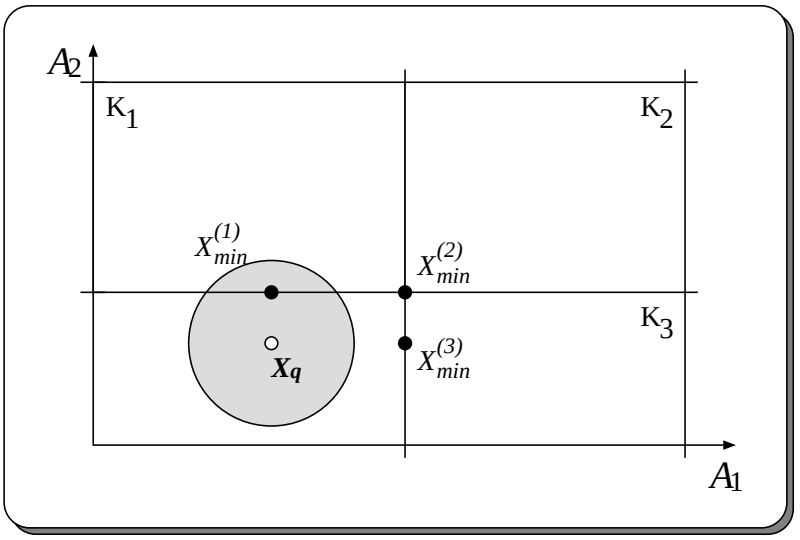


Abbildung 8.4: *BOB*–Test

Ist ein Ähnlichkeitsmaß $Sim$ auf der Menge aller Fälle gegeben, so ist für $X_{min}$ zu testen, ob dessen Ähnlichkeit zu $X_q$ mindestens so groß ist wie die des $m$-ten Nachbarn aus der aktuellen Queue. Folglich gilt

$$\boxed{BOB \iff Sim(X_{min}, X_q) \geq PQS[m] \ ( = Sim(PQC[m], X_q) \ )}$$

### 8.3.3 Ball–Within–Bounds

Der *BOB*–Test überprüft, ob die durch einen bestimmten Knoten des $k$-d-Baumes repräsentierte Fallmenge einen NN–Kandidaten enthalten kann. Die Intention des *Ball–Within–Bounds*–Tests (*BWB*) ist es dagegen, zu entscheiden, ob zum gegenwärtigen Stand der Suche alle $m$ nächsten Nachbarn bereits bestimmt sind.

In der Suchprozedur wird dieser Test nach der Abarbeitung jedes Baumknotens durchgeführt. Die Frage, die der *BWB*–Test hierbei beantworten soll, lautet: Liegt die Kugel um $X_q$ *vollständig* innerhalb der durch die geometrischen Schranken des Knotens beschriebenen Bounding Box?

Liefert der *BWB*–Test den Wert True, so bedeutet das, daß außerhalb der Bounding Box des betrachteten Knotens kein Fall mehr liegen kann, der eine größere Ähnlichkeit zu $X_q$ hat als einer der bisher gefundenen $m$ nächsten Nachbarn in der Queue. Da die Box die Kugel vollständig umschließt und die Bounding Boxes *disjunkte* Teilräume des gesamten Datenraumes beschreiben, kann sich die Kugel auch nicht mit der Box irgendeines anderen Knotens überlappen.

Folglich enthalten die anderen Teilräume keine NN–Kandidaten mehr, so das die Suche beendet werden kann. Die in der Queue enthaltenen Fälle sind somit die $m$ besten Matches zu $X_q$.

Zur Realisierung dieses Tests werden alle Kanten der Bounding Box daraufhin untersucht, ob sie einen Schnittpunkt mit der Kugel um $X_q$ haben (dies ist freilich nur dann nötig, wenn $X_q$ selbst auch innerhalb der Box liegt). Dazu werden für jede Dimension $j$ ($1 \leq j \leq k$) jeweils die zwei Kanten (bzw. Hyperebenen) betrachtet, die in dieser Dimension durch die obere bzw. die untere geometrische Schranke des Knotens beschrieben sind. In jeder der beiden Ebenen wird für den Punkt, der am nächsten zu $X_q$ liegt, getestet, ob er innerhalb der um $X_q$ definierten Kugel liegt. Ist das der Fall, so schneidet die Kugel die entsprechende Ebene. Abbildung 8.5 veranschaulicht dies anhand eines 2-dimensionalen Beispiels.

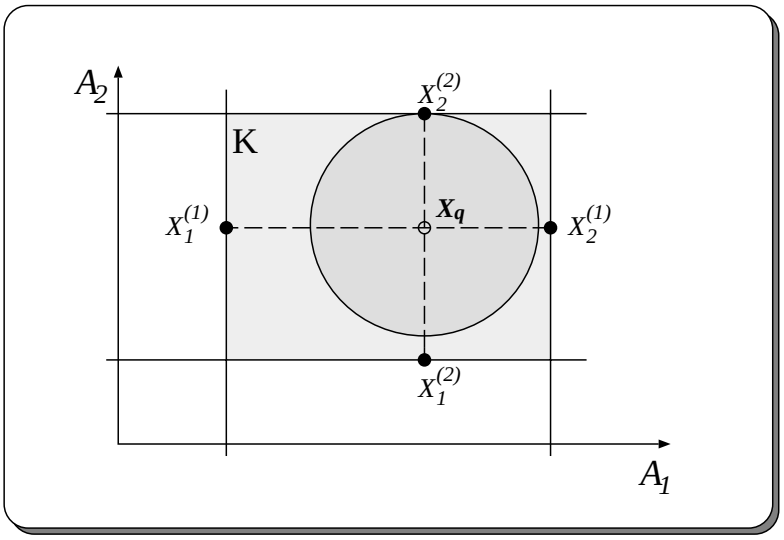


Abbildung 8.5: *BWB*–Test

Bei $k$ Dimensionen werden somit jeweils 2 Punkte betrachtet, die für die $j$-te Dimension wie folgt charakterisiert sind:

$$X_1^{(j)} := (x_{1,1}^{(j)}, x_{1,2}^{(j)}, \ldots, x_{1,k}^{(j)})$$

bzw.

$$X_2^{(j)} := (x_{2,1}^{(j)}, x_{2,2}^{(j)}, \ldots, x_{2,k}^{(j)})$$

wobei für $1 \leq i \leq k$ gilt:

$$x_{1,i}^{(j)} = \begin{cases} Lower[i] & , \quad i = j \\ X_q[i] & , \quad i \neq j \end{cases}$$

bzw.

$$x_{2,i}^{(j)} = \begin{cases} Upper[i] & , \quad i = j \\ X_q[i] & , \quad i \neq j \end{cases}$$

Bei einem gegebenen Ähnlichkeitsmaß $Sim$ gilt dann

$$\boxed{BWB \iff Sim(X_1^{(j)}, X_q) < PQS[m] \quad \wedge \quad Sim(X_2^{(j)}, X_q) < PQS[m] \qquad \forall j = 1, \ldots, k}$$

## 8.4 Die Ähnlichkeitsmaße

Für die Ähnlichkeitsmaße, die sowohl beim Aufbau des $k$-d-Baumes als auch bei der NN-Suche in die Bounds–Tests miteingehen, werden einige Eigenschaften gefordert, die in den vorangegangenen Abschnitten bereits implizit vorausgesetzt wurden.

Das Ähnlichkeitsmaß $Sim(X,Y)$ zwischen zwei Fällen ist dabei über eine $k$-stellige Funktion $F : [0,1]^k \rightarrow [0,1]$ und $k$ binäre Funktionen $\mu_j$ $(1 \leq j \leq k)$ auf den Wertebereichen der $k$ Suchattribute definiert durch

$$Sim(X,Y) = F(\mu_1(x_1,y_1), \mu_2(x_2,y_2), \ldots, \mu_k(x_k,y_k)). \tag{8.2}$$

Die Fälle $X$ und $Y$ seien dabei durch $k$ Attribute $A_1, A_2, \ldots, A_k$ beschrieben. Nach Kapitel 3 (Abschnitt 3.2) entsprechen die beiden Fälle somit $k$-dimensionalen Vektoren $(x_1, x_2, \ldots, x_k)$ bzw. $(y_1, y_2, \ldots, y_k)$, wobei die $j$-ten Komponenten $x_j := a_j^{(X)}$ bzw. $y_j := a_j^{(Y)}$ jeweils einen Wert aus dem Wertebereich $W_j$ des $j$-ten Suchattributes $A_j$ enthalten.

Die Funktion $\mu_j : W_j \times W_j \rightarrow [0,1]$ ist das auf diesem Wertebereich definierte Ähnlichkeitsmaß für zwei solche Werte.

Als Einschränkung ist jedoch zu beachten, daß die $k \leq n$ Suchschlüsselattribute *keine undefinierten* Werte enthalten dürfen.

Die Ähnlichkeit zwischen zwei Fällen setzt sich dann aus den eindimensionalen Ähnlichkeiten, die zwischen den $k$ Koordinaten bestehen, zusammen. Diese gehen auch explizit in den Baumaufbau ein, wo sie zur Bestimmung des Attributes mit der größten Streuung herangezogen werden.

Für die Funktionen $\{\mu_j\}_{j=1}^k$ werden folgende Eigenschaften gefordert:

$$\text{(Symmetrie)} \qquad \mu_j(x,y) = \mu_j(y,x) \tag{8.3}$$

$$\text{(Monotonie)} \qquad \mu_j(x,y) \geq \mu_j(x,z), \text{ falls } x \leq y \leq z \text{ oder } x \geq y \geq z \tag{8.4}$$

Das Ähnlichkeitsmaß $Sim$ fließt über die beiden Bounds–Tests in den Suchalgorithmus mit ein. Damit diese Tests die korrekten Ergebnisse liefern, muß die Funktion $F$ so definiert werden, daß die Ähnlichkeit zwischen zwei Fällen nicht größer wird, wenn die Summe der $k$ Koordinatenähnlichkeiten abnimmt.

## 8.5 Suchbeispiel

Die Vorgehensweise der Suchprozedur *SEARCH* (Abschnitt 8.2) soll an dem Beispiel aus Abschnitt 8.1.4 verdeutlicht werden. Abbildung 8.6 stellt die Partitionierung des 2-dimensionalen Datenraumes, die durch den optimierten Baum für die Menge $\mathcal{B} = \{A, B, C, D, E\}$ entsteht, graphisch dar. Die mit römischen Ziffern bezeichneten Regionen entsprechen dabei den Buckets der Blattknoten.

Für die (2-dimensionalen) Fallrepräsentationen muß ein Ähnlichkeitsmaß *Sim* definiert werden, das den in Abschnitt 8.4 geforderten Bedingungen genügt. Für dieses Beispiel definieren wir

$$Sim(X,Y) \;:=\; F(X,Y) = \frac{1}{2}(\mu_1(x_1,y_1) + \mu_2(x_2,y_2)),$$

mit $X, Y \in \{(a_1, a_2) \mid a_1 \in W_1, a_2 \in W_2\}$ und den in Abschnitt 8.1.4 definierten Ähnlichkeitsfunktionen $\mu_1$ und $\mu_2$.

Mit Hilfe der Prozedur *SEARCH* sollen nun die beiden nächsten Nachbarn (also $m = 2$) des Falles $Q = (4.5, s)$ bestimmt werden. Zu Beginn der Suche werden die Queue $PQC$, die die gefundenen nächsten Nachbarn aufnimmt, und die zugehörige Queue $PQS$ der Ähnlichkeitswerte mit einer leeren Liste initialisiert, also $PQC$:=( ) und $PQS$:=( ). Die Suche läuft dann folgendermaßen ab:

1. Aufruf von *SEARCH* für den (inneren) Knoten I:
   Vergleich der 2. Koordinate von $Q$ mit dem Partitionswert $g$

2. Aufruf von *SEARCH* für den (inneren) Knoten II:
   Vergleich der 1. Koordinate von $Q$ mit dem Partitionswert 4

3. Aufruf von *SEARCH* für Knoten III (Blattknoten, der $A$ enthält): Berechnung der Ähnlichkeit zwischen $Q$ und $A$ durch

$$Sim(Q,A) = \frac{1}{2}(\mu_1(4.5,6) + \mu_2(s,s)) = \frac{1}{2}(\frac{2}{5}+1) = \frac{7}{10}\ ;$$

da $PQC$ noch keinen Fall enthält, werden die Queues zu $PQC$=($A$) und $PQS$=($\frac{7}{10}$) aktualisiert; d.h., $A$ ist der aktuelle beste Match zu $Q$. Es ist jetzt jedoch erst *ein* nächster Nachbar bestimmt, und deshalb ist der Radius des Kreises um $Q$ noch unendlich groß.

4. BWB–Test für Knoten III: *BWB=false*, da der Kreis um $Q$ unendlich groß ist (siehe 3.)

5. Rückkehr zu Knoten II, anschließend Durchführung des BOB–Tests für Knoten IV: *BOB=true*, da der Kreis um $Q$ unendlich groß ist

6. Aufruf von *SEARCH* für Knoten IV (der $E$ enthält) und Berechnung der Ähnlichkeit zwischen $Q$ und $E$:

$$Sim(Q,E) = \frac{1}{2}(\mu_1(4.5,4) + \mu_2(s,s)) = \frac{1}{2}(\frac{2}{3}+1) = \frac{5}{6}\ .$$

Aktualisierung der Queues zu $PQC$=($E$,$A$) und $PQS$=($\frac{5}{6}$, $\frac{7}{10}$); $E$ ist jetzt der beste Match zu $Q$. Der Radius $r$ (Abb. 8.6) ist gerade so groß, daß $E$ ein Punkt des Kreises um $Q$ ist.

7. BWB–Test für Knoten IV: *BWB=false*, da der Kreis um $Q$ mit Bucket III überlappt; so liegt z.B. der Testpunkt (4,$s$), der zufälligerweise mit $E$ identisch ist, innerhalb des Kreises, da seine Ähnlichkeit ($\frac{5}{6}$) größer ist als die des aktuellen $m$-ten Nachbarn $A$ ($\frac{7}{10}$).

8. Rückkehr zu Knoten II (der die Buckets III und IV umfaßt) und Durchführung des BWB–Tests: Getestet wird nur der Punkt $X_1 = (4.5, g)$, da er auf der einzigen nicht unendlich großen Schranke des Buckets liegt (siehe Abb. 8.6). Seine Ähnlichkeit zu $Q$ beträgt

$$\frac{1}{2}(1+0.5) = \frac{3}{4} \quad > \quad \frac{7}{10}\ (= PQS[2]);$$

somit gilt *BWB=false*.

9. Rückkehr zu Knoten I, anschließend BOB–Test für Knoten V, der die Buckets VI, VIII und IX umfaßt: *BOB=true*, da der Kreis um $Q$ mit Bucket VI überlappt (vgl. 8.)

10. Vergleich der 2. Koordinate von $Q$ mit dem Partitionswert von Knoten V, anschließend Aufruf von *SEARCH* für Knoten VI (der $C$ enthält): Berechnung der Ähnlichkeit von $C$ durch $Sim(Q,C) = \frac{1}{2}(\frac{2}{9}+0.5) = \frac{13}{36} < \frac{7}{10}$ ;
die Queues und damit auch der Kreis um $Q$ bleiben unverändert.

11. BWB–Test für Knoten VI liefert *false*, da der (unveränderte) Kreis um $Q$ mit den Buckets III und IV überlappt

12. Rückkehr zu Knoten V, anschließend BOB–Test für Knoten VII (der die Buckets VIII und IX umfaßt): Testpunkt ist $X_2 = (4.5, w)$ mit der Ähnlichkeit

$$Sim(Q,X_2) = \frac{1}{2}(1+0) = \frac{1}{2} < \frac{7}{10}\ ;$$

somit gilt $BOB = false$, und Knoten VII wird *nicht* weiter durchsucht

13. Rückkehr zu Knoten V (mit den Buckets VI, VIII und IX): *BWB=false*

14. Rückkehr zu Knoten I: *BWB=true*, da dieser Knoten die Wurzel ist;
die Suche kann beendet und die Fälle $E$ und $A$ können (in dieser Reihenfolge) als die 2 nächsten Nachbarn von $Q$ ausgegeben werden.

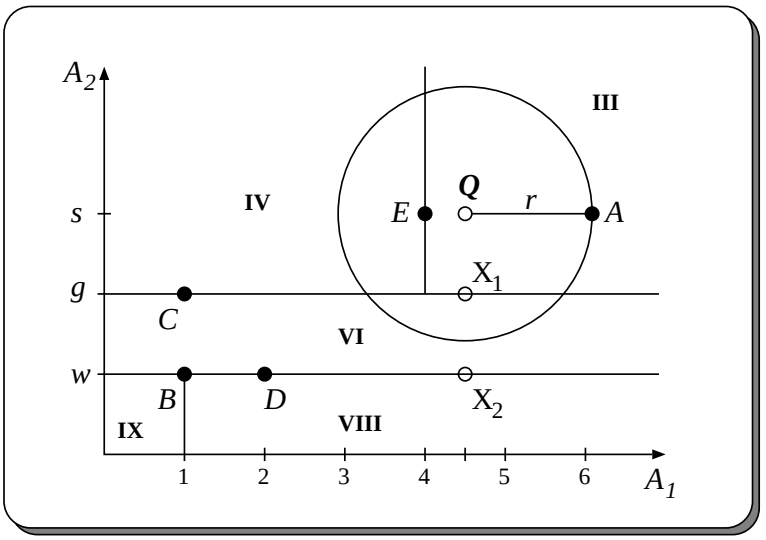


Abbildung 8.6: NN–Suche für $Q = (4.5, s)$

## 8.6 Aufwandsbetrachtungen

### Aufbau eines Baumes

Der *Speicherungsaufwand* bei der Organisation von $n$ Fällen in einem $k$-d-Baum ist proportional zu $n$, da bei einem Baum mit einer Bucketgröße $b$ die Anzahl der inneren Knoten $\lceil n/b \rceil - 1$ beträgt. Für jeden dieser Knoten sind neben den Zeigern auf die beiden Nachfolgeknoten jeweils der Diskriminator und der Partitionswert abzuspeichern.

Der beim Baumaufbau anfallende *Berechnungsaufwand* läßt sich ebenfalls sehr schnell herleiten. Als Kostenmaß nehmen Friedman et al. hier die Anzahl der Vergleiche, die notwendig sind, um den Baum aufzubauen. Auf jedem Level des Baumes werden für die erforderlichen Vergleiche jeweils alle Werte in den $k$ Attributen der zu partitionierenden Fallmenge herangezogen, was einen Aufwand proportional zu $k \cdot n$ bedeutet. Da die Tiefe des Baumes $\log n$ beträgt, ergibt sich die Anzahl der Vergleiche aus der Rekurrenzrelation

$$T_n = 2T_{\frac{n}{2}} + k \cdot n \tag{8.5}$$

Diese Rekurrenzrelation hat die Lösung $T_n = O(k \cdot n \cdot \log n)$. Somit ist der Gesamtaufwand für die Baumkonstruktion proportional zu $kn \log n$.

Diesen Aufwand kann man als Kosten der Preprocessing– oder Lernphase interpretieren. Die Korrektheit des angegebenen Suchalgorithmus ist von der Optimierung der $k$-d-Baum–Struktur jedoch unabhängig. Deshalb ist es in der Praxis nicht unbedingt notwendig, immer sofort nach *jedem* neu in die Fallbasis eingebrachten Fall eine Reorganisation der Zugriffspfadstruktur vorzunehmen.

Im allgemeinen ist eine auch nur periodisch durchgeführte Reorganisation durchaus noch für akzeptable Suchlaufzeiten ausreichend. Das Retrieval liefert auf jeden Fall ein richtiges Ergebnis.

### Suchaufwand

Die Herleitung des zeitlichen *Such*aufwandes ist etwas komplizierter. Da dieser sehr theoretische Beweis ausführlich in [FBF77] beschrieben ist, soll an dieser Stelle auf eine Darstellung in allen Einzelheiten verzichtet werden. Stattdessen sollen nur die wichtigsten Gedankengänge und Überlegungen, die hinter dem Beweis stecken, aufgezeigt werden.

Die Parameter, die in die Aufwandsbetrachtungen eingehen, sind die Anzahl $n$ der Fälle, die Dimensionalität $k$, die Anzahl $m$ der zu bestimmenden nächsten Nachbarn, die Bucketgröße $b$, ein Ähnlichkeitsmaß $Sim(X, Y)$ auf der Fallmenge und die Verteilung $p(X)$ der gegebenen Fälle im Datenraum.

Weiter sei $S_m(X_q)$ die kleinste $k$-dimensionale Kugel um den Anfragepunkt $X_q$, die die $m$ nächsten Nachbarn von $X_q$ genau umschließt, also

$$S_m(X_q) = \{X \mid Sim(X, X_q) \geq Sim(X_m, X_q)\}, \tag{8.6}$$

wobei $X_m$ der $m$-te nächste Nachbar von $X_q$ sei. Das Volumen $v_m$ dieser Kugel ist dann gegeben durch

$$v_m(X_q) = \int_{S_m(X_q)} dX. \tag{8.7}$$

Vorausgesetzt wird ein genügend großes $n$, so daß $S_m(X_q)$ im Verhältnis dazu sehr klein ist und $p(X)$ damit in diesem Breich als annähernd konstant angesehen werden kann.

Über die Wahrscheinlichkeitsverteilung für $u_m(X_q)$ (der Wahrscheinlichkeit, daß ein bestimmter Bereich des Datenraumes von der Kugel um $X_q$ umschlossenen ist) und deren vom Ähnlichkeitsmaß $Sim$ und der Verteilung $p(X)$ unabhängigen Erwartungswert $E(u_m) = m/(n+1)$ läßt sich dann der Erwartungswert für das *Volumen* der Kugel abgeschätzen durch

$$E[v_m(X_q)] \simeq [m/(n+1)][1/\overline{p}(X_q)]. \tag{8.8}$$

$\overline{p}(X_q)$ ist hierbei die mittlere Wahrscheinlichkeitsdichte der Region von $S_m(X_q)$.

Die in Abschnitt 8.1 beschriebene Vorgehensweise beim Aufbau des $k$-d-Baumes bewirkt folgendes:

1. Durch die Verwendung des Medians enthält jeder Blattknoten in etwa $b$ Fälle.
2. Durch die Bestimmung des Diskriminators über das Attribut mit der größten Streuung wird erreicht, daß die Buckets geometrisch gesehen von einem sehr kompakten mehrdimensionalen Würfel begrenzt werden, dessen Kanten parallel zu den Koordinatenachsen verlaufen.

Die Folge davon ist die Aufteilung des Datenraumes in mehrdimensionale kubische Bereiche, die alle nahezu die gleiche Anzahl an Fällen enthalten. Für das zu erwartende Volumen eines solchen Buckets ergibt sich analog zu oben der Wert $E[v_b(X_b)] \simeq [b/(n+1)][1/\overline{p}(X_b)]$, wobei $X_b$ ein Fall innerhalb des Buckets ist.

Man betrachte jetzt den kleinsten $k$-dimensionalen Würfel, der den Bereich $S_m(X_q)$ vollständig umschließt. Der Rauminhalt $V_m(X_q)$ dieses Würfels ist proportional zu $v_m(X_q)$ (dabei sei $g(k)$ der konstante Proportionalitätsfaktor, der von $k$ und dem gewählten Ähnlichkeitsmaß abhängt).

Zur Ermittlung der durchschnittlichen Anzahl der Buckets, die durchsucht werden, muß zunächst die mittlere Anzahl $\overline{l}$ der Buckets bestimmt werden, die mit $S_m(X_q)$ überlappen. Dieser Wert ist nach oben beschränkt durch die Anzahl $\overline{L}$ der Bucketüberlappungen mit dem kleinsten Würfel um $S_m(X_q)$:

$$\overline{l} \leq \overline{L} = ([(m/b) \cdot g(k)]^{1/k} + 1)^k \tag{8.9}$$

Da in jedem Bucket $b$ Fälle liegen, ergibt sich als obere Schranke für die im Durchschnitt untersuchten *Fälle* dann

$$\overline{R} \leq b \cdot \overline{L} = b \cdot ([(m/b) \cdot g(k)]^{1/k} + 1)^k. \tag{8.10}$$

Diese Abschätzung beinhaltet zwei wichtige Ergebnisse:

1. Eine Minimierung der Anzahl untersuchter Fälle bezüglich $b$ ergibt $b = 1$; d.h. jeder Blattknoten enthält nur *einen* Fall und für $\overline{R}$ gilt dann

$$\overline{R} \leq ([m \cdot g(k)]^{1/k} + 1)^k. \tag{8.11}$$

2. Die zu erwartende Anzahl der untersuchten Fälle ist *unabhängig* von $n$ und der Verteilung $p(X)$.

Diese Folgerungen sind leicht einzusehen. Wenn es das Ziel ist, die Überlappungen der Buckets mit anderen Regionen zu minimieren, dann müssen die Buckets so klein wie möglich gemacht werden.

Die Unabhängigkeit von $n$ und der Verteilung $p(X)$ ist eine direkte Folge der Vorgehensweise beim Aufbau eines idealen Baumes. Für die dabei entstehenden Buckets gilt das gleiche wie für die Kugel, in der die $m$ besten Matches liegen. Sie enthalten eine feste Anzahl an Fällen (nämlich $b$), und ihre geometrischen Begrenzungen beschreiben relativ kompakte Teilräume. Folglich hängen sie in derselben Weise von $n$ und $p(X)$ ab wie der Bereich $S_m(X_q)$. Im gleichen Maße, in dem die Anzahl der Fälle oder ihre Verteilung abnimmt, werden auch die Volumina der Buckets und der Kugel um $X_m$ kleiner. Die Anzahl $\overline{l}$ der Überlappungen bleibt jedoch konstant.

Da dadurch die Anzahl der untersuchten Fälle bei abnehmender Größe der Fallmenge konstant bleibt, ist der für die Suche erforderliche Zeitaufwand proportional zu $\log n$. Der $k$-d-Baum ist ein balancierter Binärbaum. Damit kann er abhängig von der Knotenanzahl, die proportional zu $n$ ist, in logarithmischer Zeit von der Wurzel zu den Buckets durchlaufen werden. Der Aufwand für das erforderliche Backtracking im Baum ist proportional zu $\overline{l}$; das wiederum ist, wie gezeigt, von $n$ unabhängig.

Somit ergibt sich für die Suche nach den $m$ besten Matches eines vorgegebenen Falles in Abhängigkeit von der Menge aller bekannten Fälle ein zu erwartender zeitlicher Aufwand der Größenordnung $O(\log n)$.

## 8.7 k.o.–Kriterien

Beim fallbasierten Schließen ist es oft angebracht, wenn an das ähnlichkeitsbasierte Fallretrieval noch zusätzlich sog. „k.o.–Kriterien“ gestellt werden können, die dann jeder Fall aus der Fallbasis unbedingt erfüllen muß, um als Kandidat für die besten Matches in Frage zu kommen. Wenn nicht, so soll sich der Fall auch bei einer noch so großen Ähnlichkeit, die er möglicherweise zum gesuchten Fall hat, nicht als einer der $m$ nächsten Nachbarn qualifizieren können.

Der Algorithmus zur Suche der $m$ besten Matches wurde in der hier vorliegenden Diplomarbeit dementsprechend noch erweitert. Beim Starten des Suchvorganges können die k.o.–Kriterien als eine Menge einfacher Constraints für die Attributwerte der Fallbeschreibungen angegeben werden.

An der eigentlichen Vorgehensweise des Suchens ändert sich dadurch nichts. An den inneren Knoten muß neben dem *Bounds–Overlap–Ball*–Test nur jeweils noch überprüft werden, ob in den Teilräumen Fälle liegen, die diese Kriterien erfüllen.

Da beim Durchsuchen eines Blattknotens nach eventuellen NN–Kandidaten sowieso alle darin enthaltenen Fälle zu untersuchen sind, wird hier jeder dieser Fälle zunächst auf die Erfüllung der Constraints getestet, bevor dann bei einem positiven Resultat des Testes die aufwendigere Ähnlichkeitsberechnung erfolgt.

# Kapitel 9

# Inkrementelle NN–Suche

Der Suchalgorithmus von Friedman et al. hat den nicht unerheblichen Nachteil, daß der Parameter $m$ für die Anzahl der zu bestimmenden besten Matches fest vorgegeben werden muß. Dies bedeutet für viele Anwendungen, in denen man von vornherein eben noch nicht weiß, wie viele nächste Nachbarn zur Problemlösung benötigt werden, eine ziemlich große Einschränkung, die i.a. nicht tragbar ist.

Oft ist es deshalb erforderlich, die NN–Suche auch *inkrementell* durchführen zu können, wobei jeder neue Aufruf der Suchroutine den nächsten besten Match liefern soll.

Der Idealfall wäre, wenn alle $n$ gespeicherten Fälle so behandelt werden könnten, als seien sie in einer Liste sortiert nach ihren Ähnlichkeiten zum gesuchten Fall angeordnet. Dann besteht die Bestimmung der nächsten Nachbarn einfach darin, aus dieser Liste sukzessive einen Fall nach dem anderen auszugeben. Ein expliziter Aufbau einer solchen sortierten Liste kommt jedoch aus offensichtlichen Effizienzgründen nicht in Frage. Die Vorstellung ist vielmehr die, eine Funktion zu definieren, die zusammen mit einem relativ geringen zusätzlichen Speicheraufwand die sortierte Liste in virtueller Form darstellt, ohne sie tatsächlich zu erzeugen. Die zusätzliche Speicherstruktur soll gerade soviel Information enthalten, daß ein inkrementeller Suchaufruf den nächsten Fall aus dieser impliziten Liste als Ergebnis zurückliefern kann.

Eine derartige Funktion bezeichnet man als *Generator*, die zusätzliche Speicherstruktur als *Seed–Struktur* (engl. Keim, Saat). Zusätzlich wird noch eine Initialisierungsfunktion benötigt, deren Aufgabe es ist, den Inhalt der Seed–Struktur vor dem ersten Aufruf des Generators festzulegen.

## 9.1 Einfache Vorgehensweise

Eine sehr einfache Realisierung dieser Vorgehensweise könnte in unserem Falle beispielsweise darin bestehen, Friedmans nicht–inkrementelle Suchprozedur *SEARCH* mehrmals hintereinander auszuführen, wobei durch jeden dieser Aufrufe eine geordnete Liste der von *SEARCH* gefundenen $m$ besten Matches ensteht.

Sollen z.B. die $l$ nächsten Nachbarn inkrementell bestimmt werden, so könnte eine Prozedur *SIMPLE_SEARCH*, die dann entsprechend $l$-mal auszuführen wäre, folgendermaßen aussehen:

In ihrem ersten Aufruf werden die ersten $c$ nächsten Nachbarn mit Hilfe von *SEARCH* bestimmt und in die Liste eingetragen. Der erste dieser gefundenen Matches wird als *der* beste zurückgeliefert. Bei jedem der folgenden $l - 1$ Aufrufe erhält der Benutzer dann aus der Liste den jeweils nächsten noch nicht ausgegebenen Fall als Ergebnis. Sind alle Fälle der Liste bereits ausgegeben, so startet *SIMPLE_SEARCH* die Prozedur *SEARCH* erneut, wobei deren Eingabeparameter gegenüber dem vorangegangenen Aufruf um die Konstante $c$ erhöht wird.

Die Ineffizienz von *SIMPLE_SEARCH* ist offensichtlich. Bei jedem erneuten Aufruf von *SEARCH* werden nämlich alle in den vorangegangenen Aufrufen schon gefundenen Nachbarn jedes Mal wieder mit bestimmt. Ist $l$ z.B. ein ganzzahliges Vielfaches von $c$ (also $l = i \cdot c$), so

beträgt die Anzahl der in den $l$ Aufrufen tatsächlich bestimmten nächsten Nachbarn

$$c + 2c + \ldots + ic = \frac{i(i+1)}{2} \cdot c \quad ,$$

was der Größenordnung $O(l^2)$ entspricht. Ruft der Benutzer *SIMPLE_SEARCH* schließlich auf, um den $(l+1)$–ten nächsten Nachbarn zu erhalten, so werden von *SEARCH* noch einmal *alle* $(i+1)c$ besten Matches bestimmt.

## 9.2 Ein effizienter inkrementeller Algorithmus

Die Version von A. Broder [Bro90] ermöglicht eine bezüglich des erforderlichen Zeit– und Speicheraufwandes effiziente inkrementelle Bestimmung einer beliebigen Anzahl bester Matches.

Broder führt die Suche nach den $m$ besten Matches auf das Problem zurück, in Abhängigkeit von bereits gefundenen $l$ besten Matches schrittweise jeweils den $(l+1)$–ten nächsten Nachbarn zu bestimmen. Seine Suchstrategie basiert auf den grundlegenden Ideen von Friedman et al. aus [FBF77]. Sie geht ebenfalls von einem idealen $k$-d-Baum aus. Zusätzlich verwendet Broders Suchalgorithmus eine Hilfsdatenstruktur, die die in den einzelnen inkrementellen Suchaufrufen gewonnenen Ergebnisse für nachfolgende Aufrufe festhält.

Im folgenden werden sowohl die Initialisierungsfunktion *SEARCH_INIT* für die Seed–Struktur als auch die Generatorfunktion *SEARCH_NEXT* beschrieben. In diesem Zusammenhang wird dann auch die Rolle der Seed–Struktur erläutert werden.

### Die Initialisierungsfunktion

Die Aufgabe von *SEARCH_INIT* ist es, die Seed–Struktur zu Beginn der NN–Suche *einmal* zu initialisieren. Als Eingabe erhält die Funktion einen Zeiger auf die Wurzel des $k$-d-Baumes, der die Fälle enthält, sowie einen Zeiger auf den in der Anfrage spezifizierten Fall $Q$, dessen nächste Nachbarn zu bestimmen sind.

Beide Zeiger werden in der Seed–Struktur abgelegt. $Q$ beschreibt einen Punkt im $k$-dimensionalen Datenraum der Fälle. *SEARCH_INIT* steigt deshalb rekursiv im $k$-d-Baum bis zu dem Blattknoten hinab, dessen Teilraum den durch $Q$ definierten Punkt enthält. Dabei werden die Zeiger auf alle Baumknoten, die auf dem Pfad von der Wurzel bis zu jenem Blattknoten liegen, in Form einer Liste (der *Rekursionstrace*) in der Seed–Struktur abgespeichert.

### Der Generator

Mit Hilfe der Information aus der Seed–Struktur kann *SEARCH_NEXT* dann den ersten der nächsten Nachbarn bestimmen. Hierzu werden die Ähnlichkeiten zwischen $Q$ und jedem der Fälle berechnet, die in dem Bucket des Blattknotens liegen, auf den der erste Eintrag in der Rekursionstrace verweist. Die Zeiger auf diese Fälle werden in eine nach den Ähnlichkeitswerten sortierte Nearest–Neighbor–Queue eingetragen, so daß das erste Element der Queue auf den Fall verweist, der die bisher größte Ähnlichkeit zu $Q$ aufweist.

Dieser Fall ist somit ein möglicher Kandidat für den besten Match, und gleichzeitig bedeutet seine Ähnlichkeit zu $Q$ eine untere Schranke für den aktuellen nächsten Nachbarn. Jeder weitere NN–Kandidat muß deshalb innerhalb der $k$-dimensionalen Kugel um den Mittelpunkt $Q$ liegen, deren Radius gerade so groß ist, daß der erste Fall der NN–Queue ein Punkt auf der Oberfläche dieser Kugel ist (vgl. Kapitel 8).

Wird die Kugel um $Q$ von den geometrischen Grenzen des betrachteten Blattknotens vollständig umschlossen, so bedeutet das, daß mit dem ersten Queue–Eintrag bereits *der* beste Match bestimmt ist (*Ball–Within–Bounds*–Test). Liegt die Kugel jedoch nicht innerhalb dieser Grenzen, so kann der andere Sohnknoten (bezogen auf den Vater des Blattknotens) möglicherweise einen noch ähnlicheren Fall enthalten.

Es erfolgt deshalb ein Backtracking zum Vater des betrachteten Knotens. Gleichzeitig wird der erste Eintrag aus der Rekursionstrace entfernt. Durch rekursives Hinabsteigen im anderen Teilbaum unterhalb des Vaterknotens werden dann neue NN–Kandidaten bestimmt. Es sind hierbei jedoch nur die Blätter zu durchsuchen, die mit der Kugel um $Q$ eine Überlappung bilden, was durch den *Bounds–Overlap–Ball*–Test überprüft wird. Die neuen Kandidaten werden nach ihren Ähnlichkeiten zu $Q$ in die NN–Queue einsortiert. Der erste Queue–Eintrag ist dann der (möglicherweise neue) aktuelle beste Match, durch den eine eventuell kleinere Kugel um $Q$ definiert wird.

Der Backtracking–Prozeß wird solange wiederholt, bis die immer kleiner werdende Kugel um $Q$ irgendwann von einem der betrachteten inneren Knoten vollständig umschlossen wird. Erst dann ist gewährleistet, daß alle in Frage kommenden Fälle untersucht worden sind, und *SEARCH_NEXT* kann den ersten Fall in der Kandidaten–Queue als Ergebnis ausgeben. Die restliche Queue wird schließlich ebenso wie die Ähnlichkeit des ausgegebenen nächsten Nachbarn in der Seed–Struktur abgespeichert.

Im nächsten Aufruf von *SEARCH_NEXT* zur Bestimmung des nächsten besten Matches wird wieder genauso verfahren. Das erste Element der aus dem vorangegangenen Suchaufruf übriggebliebenen NN–Queue ist jetzt der aktuelle nächste Nachbar, der entsprechend eine neue Kugel definiert. Seine Ähnlichkeit ist jedoch höchstens so groß wie die in der Seed–Struktur abgespeicherte Ähnlichkeit. Der durch den ersten Eintrag der Rekursionstrace gegebene Teilbaum wird rekursiv durchlaufen, um neue Kandidaten aus den Blattknoten, die sich mit der nun aktuellen Kugel überlappen, zu bestimmen.

Zur effizienten Unterstützung dieses Vorgehens wird eine weitere Liste in der Seed–Struktur geführt, die alle bereits vollständig durchsuchten Teilbäume enthält. Durch diese Liste soll verhindert werden, daß Teilbäume, die keine weiteren NN–Kandidaten mehr enthalten können, erneut durchlaufen werden. Wie zuvor wird der Backtracking–Prozeß wieder solange ausgeführt, bis die aktuelle Kugel um $Q$ von den geometrischen Grenzen eines der betrachteten inneren Knoten vollständig umschlossen ist.

In algorithmischer Notation sieht dieser inkrementelle Suchvorgang wie folgt aus:

| | | |
|---|---|---|
| **global** | *Seed* | Seed–Struktur, die die erforderlichen Daten, wie sie oben beschrieben sind, enthält; |
| | *Queue* | aktuelle Nearest–Neighbor–Queue; |
| | *ExaminedNodes* | Liste aller bereits vollständig durchsuchten Baumknoten; |

```
procedure SEARCH_NEXT;
local query                (* gesuchter Fall *);
      maxSimilarity        (* Ähnlichkeit des aktuellen nächsten Nachbarn *);
      trace                (* Rekursionstrace *);
      currentNode          (* aktuell betrachteter Knoten *);
      bounds               (* Schranken des betrachteten Knotens *);
      nextNearestNeighbor  (* auszugebender nächster Nachbar *);

begin
  query:=Seed.Query;
  maxSimilarity:=Seed.Max_Similarity;
  trace:=Seed.Recursion_Trace;
  Queue:=Seed.NN_Queue;
  currentNode:=trace[1];
  bounds:=boundsOf(currentNode);
  EXAMINE_NODE(currentNode,query,bounds);
  (* solange die Rekursionstrace noch nicht abgearbeitet ist und entweder die NN–Queue
  leer ist oder die Kugel nicht innerhalb der aktuellen Bounding Box liegt, tue folgendes *)
```

```
  while ( not empty(trace) and
          (empty(Queue) or
          (not BALL–WITHIN–BOUNDS(bounds, query, maxSimilarity)) ) do
  begin (* Backtracking in der Rekursionsstrace *)
    removeFirstEntry(trace);
    currentNode:=otherChild(fatherOf(trace[1]));
    EXAMINE_NODE(currentNode,query,boundsOf(currentNode));
    maxSimilarity:=similarityOf(Queue[1]);
  end;
  nextNearestNeighbor:=Queue[1];
  removeFirstEntry(Queue);
  (* Abspeichern der aktuellen NN–Queue und der Trace in der Seed–Struktur *)
  Seed.Queue:=Queue;
  Seed.Recursion_Trace:=trace;
  maxSimilarity:=similarityOf(nextNearestNeighbor);
  return nextNearestNeighbor;
end SEARCH_NEXT;
```

Die Prozedur EXAMINE_NODE(*currentNode,query,bounds*) durchsucht den Baumknoten *currentNode*, falls dieser nicht schon bereits in früheren Suchaufrufen durchsucht worden ist und falls sich seine Schranken mit der aktuellen Kugel überlappen. Ist *currentNode* ein Blattknoten, werden seine Fälle nach ihren Ähnlichkeiten in die NN–Queue einsortiert. Bei einem inneren Knoten werden dessen Söhne rekursiv durchsucht. Sind alle Knoten im Teilbaum unterhalb von *currentNode* untersucht worden, so wird *currentNode* anstelle seiner Söhne in die Liste der untersuchten Knoten eingetragen, und EXAMINE_NODE liefert den Wert true zurück.

```
procedure EXAMINE_NODE(currentNode,query,bounds);
local temp1, temp2 : Boolean;
begin
  if is_in_ExaminedNodes(currentNode) then return true;
  if not BOUNDS–OVERLAP–BALL(bounds,query,similarityOf(Queue[1])) then
    return false;
  if isTerminal(currentNode) then
  begin
    merge(Queue,buildUp_Queue(currentNode.bucket));
    addTo_ExaminedNodes(currentNode);
    return true;
  end
  else begin (* rekursiver Aufruf für die Söhne von currentNode *)
    temp1:=EXAMINE_NODE(currentNode.leftSon,query,
                        boundsOf(currentNode.leftSon));
    temp2:=EXAMINE_NODE(currentNode.rightSon,query,
                        boundsOf(currentNode.rightSon));
    if temp1 and temp2 then
    begin
      removeFrom_ExaminedNodes(currentNode.leftSon);
      removeFrom_ExaminedNodes(currentNode.rightSon);
      addTo_ExaminedNodes(currentNode);
      return true;
    end
    else return false;
  end;
end EXAMINE_NODE;
```

## 9.3 Beispiel

Ein einfaches 2-dimensionales Beispiel soll die Vorgehensweise des inkrementellen Suchalgorithmus noch einmal verdeutlichen (Abb. 9.1 (a)–(c)).

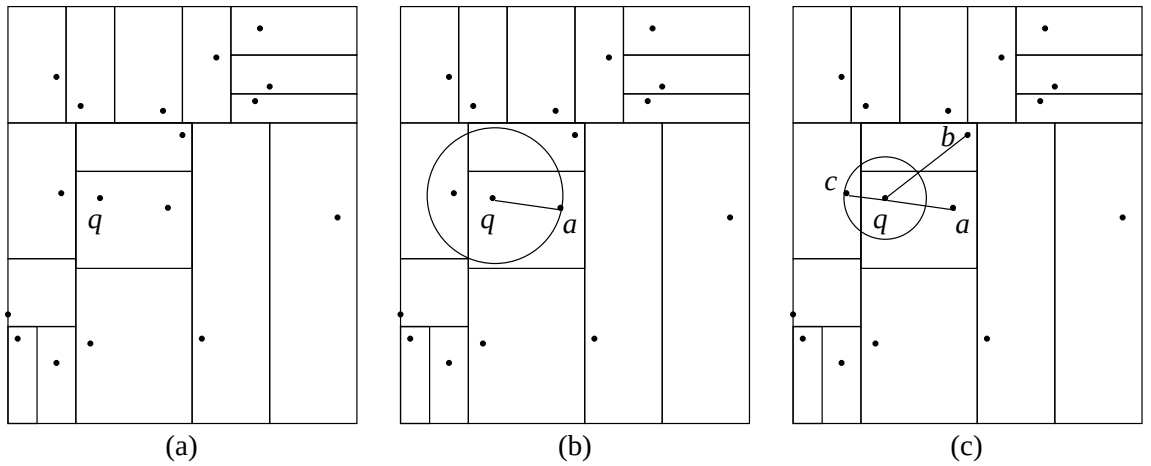


Abbildung 9.1: Inkrementelle Suche

Abbildung 9.1(a) zeigt eine $k$-d-Baumstruktur mit einer Bucketgröße $b = 1$ für 16 gegebene Datensätze, die hier als Punkte dargestellt sind. $q$ bezeichnet den in der Anfrage spezifizierten Punkt, dessen nächste Nachbarn inkrementell bestimmt werden sollen.

Durch den Algorithmus wird zunächst der Baum von der Wurzel bis zu dem Blattknoten hin durchlaufen, in dessen Teilraum der Punkt $q$ liegt. In Abbildung 9.1(b) ist die Ähnlichkeit zwischen dem Anfragepunkt $q$ und dem Punkt $a$, der im Bucket dieses Blattknotens abgespeichert ist, berechnet. Damit ist $a$ unser erster Kandidat für den nächsten Nachbarn von $q$.

Der durch $a$ definierte Kreis um den Mittelpunkt $q$ liegt jedoch nicht vollständig innerhalb der Grenzen des betrachteten Blattknotens, so daß es weitere Buckets gibt, die einen eventuell besseren NN–Kandidaten enthalten können. Folglich muß im $k$-d-Baum wieder nach oben gegangen werden, um weitere Äste des Baumes zu durchlaufen. Es werden schließlich die beiden Blattknoten, die sich mit dem Kreis noch überlappen, aufgesucht und die beiden Punkte $b$ und $c$ untersucht.

In Abbildung 9.1(c) sind die Ähnlichkeiten zwischen $q$ und den beiden Punkten $b$ und $c$ berechnet. Da sowohl die Ähnlichkeit von $a$ als auch die von $b$ kleiner ist als die Ähnlichkeit von $c$, ist $c$ jetzt der neue Kandidat für den nächsten Nachbarn.

Darüber hinaus überlappt sich der durch $c$ neu entstandene Kreis um $q$ nicht mehr mit einem der anderen noch nicht untersuchten Buckets. Das bedeutet, daß der Suchvorgang hier beendet und $c$ als der aktuelle nächste Nachbar von $q$ ausgegeben werden kann. Die Punkte $a$ und $b$ werden in dieser Reihenfolge in der Kandidaten–Queue der Seed–Struktur abgespeichert.

Der nächste Aufruf von *SEARCH_NEXT* würde dann $a$ ausgeben, da der durch $a$ definierte Kreis um $q$ keine Überschneidungen mit noch nicht untersuchten Blattknoten aufweist. Im dritten Aufruf jedoch überlappt der durch $b$ und $q$ beschriebene Kreis mit mehreren noch nicht betrachteten Blättern. Um also den dritten nächsten Nachbarn zu bestimmen, müssen hier deshalb wieder alle Ähnlichkeiten zwischen $q$ und den in diesen Blättern enthaltenen Punkten berechnet werden.

## 9.4 Performance–Verbesserung

In jedem *SEARCH_NEXT*–Aufruf müssen jeweils alle Blattknoten, die eine Überlappung mit der neuen Kugel aufweisen, im $k$-d-Baum aufgesucht werden. Dabei liegen einige von ihnen in bereits teilweise durchsuchten Teilbäumen. Folglich gibt es innere Baumknoten, für die der *Bounds–Overlap–Ball*–Test mehrmals durchgeführt wird (im Extremfall gerade so oft, wie der Teilbaum unterhalb eines solchen Knotens Blätter enthält).

Innerhalb eines einzelnen *SEARCH_NEXT*–Aufrufes jedoch wird keiner der inneren Knoten mehr als einmal getestet. Demnach verbessert sich die Performance des Algorithmus, wenn die inneren Knoten weniger häufig besucht werden müssen.

Dies kann man mit einer „look ahead"–Vorgehensweise erreichen, die bei einem *SEARCH_NEXT*–Aufruf nicht nur einen einzigen, nämlich *den* nächsten Nachbarn bestimmt, sondern jeweils gleich *die i nächsten*.

Im ersten Aufruf wird deshalb für den Aufbau der NN–Kandidaten–Queue die Kugel herangezogen, deren Radius durch den $i$-ten Eintrag der Queue bestimmt ist. Der Baum wird dann — wie bisher auch — solange durchsucht, bis einer der betrachteten inneren Knoten diese Kugel ganz umschließt. Somit ist gewährleistet, daß die ersten $i$ Einträge der NN–Queue auch die $i$ besten Matches sind. *SEARCH_NEXT* gibt dann den ersten davon aus.

Über einen Zähler, dessen aktueller Wert jeweils in der Seed–Struktur abgespeichert wird, ist bei jedem *SEARCH_NEXT*–Aufruf genau bekannt, wie viele der ersten Queue–Einträge tatsächlich noch als *die* nächsten besten Matches gültig sind. Wird in einem Aufruf irgendwann festgestellt, daß alle $i$ gültigen nächsten Nachbarn bereits ausgegeben sind, müssen die nächsten $i$ besten Matches nach demselben Verfahren bestimmt werden. In jedem Falle liefert *SEARCH_NEXT* immer nur den ersten Queue–Eintrag zurück und verringert gleichzeitig den Zähler für die noch verbliebenen gültigen nächsten Nachbarn jeweils um 1.

Die Folge dieser einfachen Modifikation ist eine Reduzierung der an den inneren Knoten durchgeführten *BOB*–Tests um den Faktor $1/i$. In einem viel stärkeren Maße noch als die Bounds–Tests beeinflussen jedoch die Ähnlichkeitsberechnungen den zeitlichen Suchaufwand bei der inkrementellen Vorgehensweise. Die Anzahl dieser Berechnungen läßt sich jedoch leider nicht mehr verringern.

# Kapitel 10

# Typkonzept zur Realisierung der Ähnlichkeitssuche in k-d-Bäumen

Um nun Anwendungsbereiche des fallbasierten Schließens mit der Nearest–Neighbor–Suche in $k$-d-Bäumen verknüpfen zu können, sind, wie im Verlaufe dieser Diplomarbeit schon mehrfach angeklungen ist, einige Modifikationen bzw. Erweiterungen durchzuführen.

Im Gegensatz zu den in der Literatur behandelten klassischen Problemen, die vorwiegend geometrischer Natur sind, kann man in realen Anwendungen im allgemeinen nicht davon ausgehen, daß ausschließlich numerische Daten vorliegen. Vielmehr treten Daten verschiedenster Wertebereiche auf, die sowohl numerisch als auch nicht–numerisch sein können, wobei natürlich auch die nicht–numerischen Daten für das ähnlichkeitsbasierte Retrieval relevant sein können.

Um den Erfordernissen realer Anwendungsbereiche des fallbasierten Schließens gerecht zu werden, müssen folglich auch diese Daten Berücksichtigung bei der Verwendung der $k$-d-Baumstruktur finden.

Dabei muß für die Wertebereiche gewährleistet sein, daß auf ihren Elementen eine Ordnung und ein Ähnlichkeitsmaß definiert sind (vgl. Kapitel 3, Abschnitt 3.2).

Die Forderung nach geordneten Wertebereichen ergibt sich aus der Organisation und der Suche in $k$-d-Bäumen, die beide auf Vergleichen von Attributwerten an den inneren Knoten basieren. Die Berechnung der Ähnlichkeiten zwischen Fällen erfolgt in dieser Diplomarbeit über Ähnlichkeitsfunktionen auf Attributebene (siehe Kapitel 8).

Der Ähnlichkeitsbegriff ist jedoch in hohem Maße anwendungsabhängig. Deshalb muß der Benutzer die erforderlichen Ähnlichkeitsmaße je nach Anwendung selbst definieren können.

Gefordert ist somit ein Konzept zur Spezifikation *anwendungsabhängiger Wertebereiche*, die die geforderten Eigenschaften besitzen. Die Wertebereiche entsprechen dann in gewisser Weise *benutzerdefinierten Datentypen* für die einzelnen Suchschlüsselattribute der Fallbeschreibungen. Mit Hilfe dieser Typen modelliert der Benutzer dann auch den anwendungsspezifischen Ähnlichkeitsbegriff.

In diesem Kapitel wird ein solches Typkonzept zunächst formal beschrieben. Der Mechanismus zur Realisierung dieses Konzepts wird in Kapitel 12 im Rahmen der Implementierungsbeschreibung erläutert.

## 10.1 Benutzerdefinierte Typen

### 10.1.1 Spezifikation

Die Wertebereiche, die ein Benutzer für die Repräsentationsattribute festlegen kann, umfassen als Elemente entweder einfache oder komplexe Werte (vgl. Kapitel 3, Abschnitt 3.2). Durch ihre Zuordnung zu Attributen übernehmen die Wertebereiche die Rolle von Datentypen, die die Menge der Werte für ein Attribut einschränken. Darüber hinaus muß der Benutzer für jeden dieser Typen

ein eigenes Ähnlichkeitsmaß vorgeben, das in die Modellierung der Ähnlichkeit zwischen den Fällen eingeht.

Somit können Werte als Elemente verschiedener Wertebereiche mit unterschiedlichen Ähnlichkeitsmaßen auftreten und je nach Wertebereich entsprechend anders interpretiert werden. Dadurch können sie z.B. auch zur Darstellung unterschiedlicher anwendungsspezifischer Daten herangezogen werden.

**Definition (benutzerdefinierter Typ):** Ein *benutzerdefinierter Typ* $T$ ist ein 5-Tupel $T = (l, t, r, s, f)$, wobei für die einzelnen Komponenten gilt:

- $l \in \mathcal{L} = \{l_0, l_1, l_2, \ldots\}$ ist ein Label, durch das der Typ $T$ eindeutig identifizierbar ist
- $t$ spezifiziert die *Grundmenge* der (einfachen oder komplexen) Werte, die ein Attribut dieses Typs $T$ annehmen kann
- die geordnete Menge $r$ schränkt die Grundmenge $t$ auf eine Teilmenge ein — den *Wertebereich* von $T$; ein Attribut vom Typ $T$ kann dann nur Werte aus $r$ annehmen;
  wird $r$ nicht näher spezifiziert, so kann ein Attribut dieses Typs mit *allen* Werten aus der Grundmenge $t$ belegt werden
- $s$ ist ein benutzerdefinierter Typ, der bezüglich $T$ als dessen übergeordneter *Supertyp* zu interpretieren ist;
  dadurch entsteht eine Typhierarchie, die die *is-a*-Relation zwischen benutzerdefinierten Typen modelliert
- $f$ schließlich ist eine 2-stellige reellwertige Funktion, durch die das Ähnlichkeitsmaß auf den Werten, die $T$ umfaßt, definiert ist; (wie $f$ genau auszusehen hat, wurde bereits in Kapitel 8 beschrieben)

Damit die Typhierarchie, die durch die Definition solcher Typen entsteht, die Struktur eines Baumes hat, wird noch ein sog. *Nulltyp* als Wurzel dieses Baumes definiert.

Der Nulltyp selbst kann jedoch keinem Attribut zugeordnet werden. Seine einzige Funktion ist die eines Pseudo–Supertyps für all jene Typen, denen sonst kein ausgezeichneter Benutzertyp übergeordnet werden kann. In seiner Eigenschaft als oberster Typ definiert der Nulltyp ein Default–Ähnlichkeitsmaß auf der universellen Menge aller möglichen Werte, das auf alle seine untergeordneten Typen weitervererbt werden kann. Dieses Maß überprüft zwei beliebige Werte auf ihre Gleichheit und liefert den Wert 1, wenn beide gleich sind, und den Wert 0, wenn beide ungleich sind.

**Definition (Nulltyp):** Der *Nulltyp* ist das Tupel $NULL = (l_0, \perp, \perp, \perp, f_d)$, dessen Werte $t$, $r$ und $s$ nicht definiert sind. Das für diesen Typ angegebene universelle Ähnlichkeitsmaß $f_d$ ist für zwei beliebige Werte $x$ und $y$ definiert durch

$$f_d(x, y) = \begin{cases} 1 & , \text{ falls } x = y \\ 0 & , \text{ sonst} \end{cases}$$

Sei $\mathcal{T}$ eine Menge von benutzerdefinierten Typen einschließlich des Nulltyps. Dann müssen die einzelnen Komponenten eines benutzerdefinierten Typs $T \in \mathcal{T} \setminus \{NULL\}$ folgende Bedingungen erfüllen (die Komponenten von $T$ seien mit $l^{(T)}$, $t^{(T)}$, $r^{(T)}$, $s^{(T)}$ und $f^{(T)}$ bezeichnet):

1. mit Ausnahme von $r^{(T)}$ müssen alle Komponenten definiert sein
2. $\forall\, U, V \in \mathcal{T} : l^{(U)} \neq l^{(V)}$ (eindeutige Identifizierbarkeit)
3. $t^{(T)} \in \{$*Integer, Float, Boolean, String, Symbol, OrderedSet, List, Interval*$\}$ (siehe Kapitel 3, Abschnitt 3.2); das heißt, die dem Typ zugrundeliegende Menge $t^{(T)}$ muß eine *geordnete* Menge sein

4. $s^{(T)} \in \mathcal{T}$
5. $s^{(T)} \neq T$
6. $t^{(T)} \subseteq t^{(S)}$, falls $s^{(T)} = S$
7. (a) $r^{(T)} \subseteq t^{(T)}$
   (b) $r^{(T)} \subseteq r^{(S)}$, falls $s^{(T)} = S$
8. $f^{(T)} : t^{(T)} \times t^{(T)} \rightarrow [0,1] \subseteq \mathcal{R}$

### 10.1.2 Verwendung der Typen in Attributen

Wird ein solcher benutzerdefinierter Typ $T \in \mathcal{T}$ einem Fallrepräsentationsattribut $A_j$ zugeordnet, so muß entsprechend für den $j$-ten Attributwert einer Fallbeschreibung $X$ aus der Fallbasis $\mathcal{FB}$ gelten: $A_j(X) \in t^{(T)}$ bzw. $A_j(X) \in r^{(T)}$.
Außerdem ergibt sich die Ähnlichkeit dieses Attributwertes von $X$ zu dem eines weiteren Falles $Y \in \mathcal{FB}$ durch

$$\mu_j(x_j, y_j) = f^{(T)}(x_j, y_j), \qquad \text{wobei } x_j := A_j(X) \text{ und } y_j := A_j(Y).$$

## 10.2 Typhierarchien

Durch die Definition solcher Typen entstehen baumartige Typhierarchien, in denen jeder Typ (mit Ausnahme des Nulltyps) genau einen übergeordneten Supertyp besitzt.

Die Knoten der Bäume entsprechen den benutzerdefinierten Typen, wobei die Baumwurzel den Nulltyp enthält. Jeder andere innere Knoten ist der Supertyp aller durch seine Söhne repräsentierten Typen. Die Kante zwischen einem Vater– und seinem Sohnknoten stellt jeweils eine Spezialisierung des im Vater repräsentierten Typs dar (*is-a*-Relation), so daß ein Pfad von der Wurzel bis zu einem Baumknoten eine ganze Spezialisierungshierarchie beschreibt.

Innerhalb dieser Spezialisierungshierarchien können die Werte der Komponenten $t$, $r$ und $f$ jeweils vom Vaterknoten auf seine Söhne weitervererbt werden, sofern sie dort durch neue Werte nicht überschrieben werden. Ein Überschreiben der Werte von $t$ und $r$ ist nur möglich, wenn diese beiden Mengen lediglich weiter eingeschränkt werden. Die Ähnlichkeitsfunktion $f$ kann hingegen neu definiert werden. Nach dem *Vererbungsprinzip* gilt für einen Typ $T$ und dessen Supertyp $S$ somit

1. $T = (l^{(T)}, \perp, r^{(T)}, s^{(T)}, f^{(T)}) \Rightarrow t^{(T)} = t^{(S)}$
2. $T = (l^{(T)}, t^{(T)}, \perp, s^{(T)}, f^{(T)}) \Rightarrow r^{(T)} = r^{(S)}$
3. $T = (l^{(T)}, t^{(T)}, r^{(T)}, s^{(T)}, \perp) \Rightarrow f^{(T)} = f^{(S)}$

Im folgenden sind noch einige beispielhafte Typdefinitionen aufgeführt:

- die Grundtypen *NUMBER, INTEGER* und *FLOAT*:
  - *NUMBER* = ($l_1$, Number,$\perp$, *NULL*, $f_e$)
  - *FLOAT* = ($l_2$, Float,$\perp$, *NUMBER*,$\perp$)
  - *INTEGER* = ($l_3$, Integer,$\perp$, *NUMBER*,$\perp$)

  Die Grundmenge des Wertebereiches von *NUMBER* ist die Menge aller (reellen) Zahlen. Durch die Definition der Typen *INTEGER* und *FLOAT* als Subtypen von *NUMBER* wird der Wertebereich in diesen beiden Typen auf die ganzen bzw. die rationalen Zahlen eingeschränkt.
  Die Ähnlichkeitsfunktion $f_e$ vererbt sich von *NUMBER* auf beide Subtypen. $f_e$ ist hierbei die aus dem euklidischen Abstandsmaß für numerische Werte abgeleitete Ähnlichkeit $f_e(x, y) = 1 \ / \ (1 + |x - y|)$.

- boolesche Grundtypen:
  - $BOOLEAN = (l_4, \text{Boolean}, \perp, NULL, \perp)$
  - $SYMMETRIC_BOOLEAN = (l_5, \perp, \perp, NULL, f_s)$
  - $ASYMMETRIC_BOOLEAN = (l_6, \perp, \perp, NULL, f_{as})$

  Die Grundmenge der drei Typen ist die Menge der beiden booleschen Werte. Die beiden Subtypen von *BOOLEAN* unterscheiden sich in der Definition ihres Ähnlichkeitsmaßes für zwei boolesche Werte $a$ und $b$:

$$f_s = \begin{cases} 1 & , \text{ falls } a = b \\ 0 & , \text{ sonst} \end{cases}$$

$$f_{as} = \begin{cases} 1 & , \text{ falls } a = b = \text{true} \\ c & , \text{ falls } a = b = \text{false} \\ 0 & , \text{ sonst} \end{cases} \qquad (c \in [0,1] \subseteq \mathcal{R})$$

- der Grundtyp *SYMBOL*:
  - $SYMBOL = (l_7, \text{Symbol}, \perp, NULL, \perp)$

  Die Grundmenge von *SYMBOL* umfaßt alle als Symbole interpretierbaren Werte; sie ist nicht weiter eingeschränkt. Da hier keine Ähnlichkeit explizit definiert ist, erbt dieser Typ das Default–Ähnlichkeitsmaß aus seinem Supertyp *NULL*. Somit beträgt die Ähnlichkeit zwischen zwei Symbolen 1, wenn sie beide gleich sind, ansonsten ist sie 0.

- Definition eines Typs, dessen Wertebereich alle ganzen Zahlen aus dem Intervall $[5, 20] \subseteq \mathcal{Z}$ enthält, wobei die Ähnlichkeit zwischen zwei Zahlen aus diesem Intervall durch die Funktion $f_e$ berechnet werden soll, die bereits für *NUMBER* definiert wurde:
  - $MY_INTERVAL = (l_8, \perp, [5, 20], INTEGER, \perp)$

  Die für *INTEGER* spezifizierte Grundmenge ist die Menge $\mathcal{Z}$. Für den Typ *MY_INTERVAL* wird sie auf die Werte aus dem Intervall $[5, 20]$ eingeschränkt.

Abbildung 10.1 zeigt schließlich die Typhierarchie, die durch die obigen Definitionen entsteht.

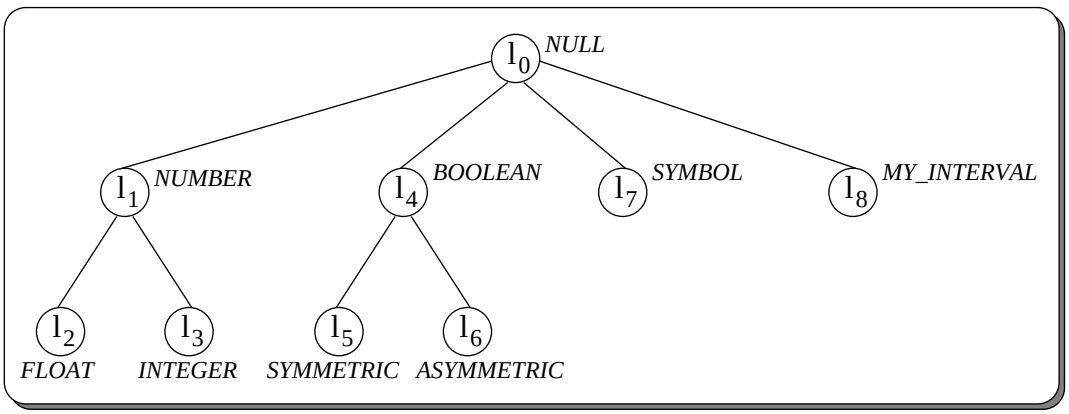


Abbildung 10.1: Typhierarchie

# Kapitel 11

# Zusammenfassung und Ausblick

In dieser Diplomarbeit wurde ein Mechanismus für das *ähnlichkeitsbasierte Retrieval* von Fällen aus einer Fallbasis bereitgestellt. Der Mechanismus unterstützt den initialen Prozeßschritt des fallbasierten Schließens in Form eines einfachen Matchers, der Ähnlichkeiten auf der niedrigen Abstraktionsstufe der syntaktischen Fallbeschreibungen berücksichtigt (*Surface Matches*). Die Fallbasis sowie die für das Retrieval erforderlichen Datenstrukturen und Algorithmen wurden auf einem objektorientierten Datenbanksystem implementiert (siehe Kapitel 12).

Der dieser Arbeit zugrunde liegende Problemlösungsansatz faßt das ähnlichkeitsbasierte Retrieval als Best–Match– bzw. *Nearest–Neighbor*–Suche auf, wobei ein neuer Problemfall in Form einer Anfrage (*Query*) an die Fallbasis spezifiziert wird. Zu suchen sind die Fälle aus der Fallbasis, die zu dieser Anfrage die besten Matches liefern. Da die Fälle durch Attribut–Wert–Paare beschrieben sind, entspricht dies einem bekannten Problem aus dem Bereich der Information–Retrieval–Systeme: dem assoziativen Retrieval.

Diese Diplomarbeit basiert deshalb auch auf Ansätzen aus diesem Bereich. Insbesondere wurden Verfahren der Best–Match–Suche in *mehrdimensionalen Zugriffspfadstrukturen* verwendet. Hierbei untersuchten wir die Datenstruktur der *k-d-Bäume*. (Eine übersichtsartige Zusammenfassung gibt hierzu auch das Positionspapier [ÖW92]).

### Modellierung der Ähnlichkeit

Der Begriff der Ähnlichkeit geht sowohl in die Organisation der Fallbasis als auch in den Retrievalmechanismus mit ein. Die Modellierung erfolgt über ein Ähnlichkeitsmaß, das sich aus den einzelnen Maßen für die verschiedenen Wertebereiche der Repräsentationsattribute zusammensetzt. Zur Unterstützung der NN–Suche werden mit Hilfe statistischer Methoden und unter Verwendung der Ähnlichkeiten auf Attributebene optimierte (ideale) $k$-d-Bäume aufgebaut. Die besten Matches werden durch eine ähnlichkeitsgesteuerte Suche in diesen Bäumen bestimmt.

Entscheidend für die Verwendung von $k$-d-Bäumen in Anwendungen fallbasierter Systeme ist deshalb das entwickelte *Typkonzept* zur Spezifikation *anwendungsabhängiger Ähnlichkeitsmaße* für beliebige Wertebereiche, die ein Benutzer selbst definieren kann.

Eine der zentralen Fragen in der weiteren Entwicklung des angestrebten fallbasierten Fertigungsplanungssystems wird nun die Definition entsprechend geeigneter Ähnlichkeitsmaße sein, die den Begriff der „fertigungstechnischen Ähnlichkeit“ so modellieren, daß der Aufwand für die erforderlichen Modifikationen bekannter Fertigungspläne möglichst gering ist.

### Aufbau optimierter k-d-Bäume

In unserem Ansatz werden die Bäume top–down aufgebaut, d.h., die Buckets ergeben sich aus rekursiv durchgeführten Partitionierungen der gegebenen Fallmenge. Ein entscheidender Punkt für die Effizienz der NN–Suche ist der Aufbau optimierter Bäume. Dies beinhaltet vor allem die Bestimmung der Suchschlüssel und der entsprechenden Werte, nach denen die Fallmengen im

Baum jeweils partitioniert werden. In unserem Ansatz wird hierfür das statistische Streuungsmaß des Interquartilsabstandes in Verbindung mit Ähnlichkeitsmaßen herangezogen.

- Vom Standpunkt des *Concept Clustering* aus entspricht die dadurch gewonnene Partitionierung der Fälle lediglich einer Annäherung an die Einteilung in konzeptuelle Klassen. Könnte jedoch das Concept–Clustering–Problem gelöst werden, so bestände eine mögliche Alternative darin, den Baumaufbau bottom–up zu gestalten. Ausgehend von den Klassen wären die Buckets dann so zu wählen, daß die zueinander ähnlichsten Fälle im Baum nebeneinander liegen. Unter dieser Prämisse könnten dann auch entsprechend die Partitionswerte an den inneren Knoten bestimmt werden.

- Der *Lernvorgang* besteht zunächst nur darin, daß ein neu gelöster Fall in die Fallbasis aufgenommen und die Zugriffspfadstruktur entsprechend angepaßt wird. In diesem Zusammenhang wäre es noch interessant zu überlegen, wie weitere Lerneffekte aus den auftretenden Anfragen abgeleitet werden können. Ein in dieser Richtung eventuell zu untersuchender Ansatz könnte in Anlehnung an die einfachen binären Suchbäume beispielsweise darin bestehen, die Wahrscheinlichkeiten, die sich aus den Zugriffshäufigkeiten für die vorhandenen Fälle ergeben, in den Aufbau der Bäume mit einzubeziehen (vgl. hierzu z.B. [Meh77], Kap. III.3). Ziel ist es dabei, den Baum so zu organisieren, daß Zugriffe auf Fälle mit höheren Anfragehäufigkeiten schneller erfolgen können als auf Fälle mit niedrigeren Wahrscheinlichkeiten.

- Die *Korrektheit* der vorgestellten Suchalgorithmen ist von einer balancierten Baumstruktur jedoch unabhängig. Die Verwendung idealer Bäume begünstigt ausschließlich das *Effizienzverhalten* dieser Verfahren.

Zusammenfassend läßt sich festhalten, daß der $k$-d-Baum in Verbindung mit dem bereitgestellten Typkonzept eine durchaus geeignete Datenstruktur zur Unterstützung des ähnlichkeitsbasierten Fallretrievals ist. Empirische Untersuchungen von Friedman et al. ([FBF77]) haben zudem gezeigt, daß die Vorgehensweise der beschriebenen Suchverfahren (außer bei sehr kleinen Fallmengen) in der Regel schneller ist als andere bekannte Methoden.

### Verwendung einer objektorientierten Datenbank

Wir implementierten die Fallbasis auf dem objektorientierten DBS GemStone. Mit Hilfe der DB–Sprache OPAL konnten wir die $k$-d-Bäume als *abstrakten Datentyp* realisieren. Die NN–Suche entspricht dabei einer für diesen Typ definierten Operation. Aufgrund der Objektorientierung von OPAL ist es zudem möglich, die komplexen Objekte, die üblicherweise in Anwendungen fallbasierter Systeme auftreten, geeignet zu modellieren.

In Kapitel 2 wurde bereits erläutert, daß ein solcher Ansatz mit konventionellen DBS nicht oder nur sehr schwer zu realisieren ist.

Allerdings kann man den objektorientierten Ansatz in gewisser Weise auch als Rückschritt gegenüber den relationalen DBS ansehen, da OPAL z.B. eine *prozedurale* DB–Sprache ist. In relationalen Datenbanken stehen jedoch bereits *deklarative*, wertorientierte Sprachen zur Verfügung, in denen der Anwender nur noch spezifiziert, "was" er haben möchte, während das "wie" dem DBS überlassen bleibt.

So führt z.B. auch J.D. Ullman in [Ull88] an, daß in zukünftigen Systemen eine Verknüpfung von wissensbasierten Systemen (KBS) und Datenbanksystemen (DBS) zu sog. *Wissensverwaltungssystemen* zu erwarten sei, in denen die Integration der deklarativen Sprachen wissensbasierter Systeme mit den Retrievalsprachen der DBS eine effizientere Wissensbereitstellung als in bisherigen KBS erlaubt.

# Teil IV

# Implementierung

# Kapitel 12

# Implementierungsbeschreibung

## 12.1 Übersicht

Im Rahmen dieser Diplomarbeit wurde ein anwendungsunabhängiges generisches Tool erstellt, das folgende Funktionen umfaßt:

- Erstellung und Generierung einer Fallbasis für Anwendungen des fallbasierten Schließens
- Erzeugung von $k$-d-Baum–Strukturen für Anwendungsobjekte
- Best–Match–Suche auf der Datenstruktur des $k$-d-Baumes:
  - starre Suche nach den $m$ nächsten Nachbarn für ein festes $m$
  - inkrementelle NN–Suche
- Definition anwendungsspezifischer Typen mit wertebereichsabhängigen Ähnlichkeitsmaßen

Die Implementierung dieses Tools erfolgte in dem objektorientierten Datenbanksystem GemStone in der dort zur Verfügung stehenden Datendefinitions– und Manipulationssprache OPAL [Serv86]. Über die von GemStone bereitgestellte Benutzerschnittstelle *GSI* können die Funktionen und Mechanismen auch in Smalltalk-80–Anwendungen aufgerufen werden.

Die benötigten Datenstrukturen wurden in Form von OPAL–Klassen implementiert und die auf diesen Datenstrukturen definierten Operationen entprechend als Methoden für diese Klassen und ihre Instanzen.

Die Implementierungbeschreibung beschränkt sich hier auf das wesentlichste. Sie erfolgt hauptsächlich aus der Sicht eines Anwenders, so daß wir hier möglichst nicht auf spezielle Implementierungsdetails eingehen werden. Es soll vor allem beschrieben werden, wie ein Anwendungsprogramm die bereitgestellten Funktionen verwenden kann.

## 12.2 Die Fallbasis

### 12.2.1 Konzeption

Es können Fallbasen für verschiedene Anwendungsbereiche des fallbasierten Schließens aufgebaut werden. Eine Fallbasis enthält für ein bestimmtes Anwendungsgebiet neben den gespeicherten Fällen noch die Suchstrukturen, die zur Unterstützung des ähnlichkeitsbasierten Fallretrievals als Zugriffspfade für die Fälle angelegt werden können.

Eine vom Benutzer generierte Fallbasis wird als Instanz der OPAL–Klasse `Database` unter einem eindeutigen Namen in der GemStone–Datenbank abspeichert. Jede Instanz wird dabei in das GemStone–Dictionary `UserGlobals` eingetragen, wobei der angegebene Name der Fallbasis gleichzeitig als Dictionary–Schlüssel dient, unter dem sie in `UserGlobals` referenziert werden kann.

Eine `Database`–Instanz ist ein SymbolDictionary, das folgende Komponenten enthält:

- die Menge der in der Fallbasis gespeicherten Fälle
- die Menge der auf diesen Fällen erzeugten Zugriffspfadstrukturen
- eine Liste mit allen Fallattributen, die als Suchschlüssel in einer der bestehenden Zugriffspfadstrukturen verwendet werden

Im `UserGlobals`–Dictionary können mehrere Fallbasen angelegt werden, von denen immer eine als aktuelle Fallbasis gekennzeichnet ist, an die dann auch alle Instanzen–Nachrichten geschickt werden. Zu diesem Zweck wurde für die Klasse `Database` die Klassenvariable `nameOfCurrent` definiert, die den Namen der gerade aktuellen Fallbasis enthält. Der Zugriff auf die aktuelle Fallbasis kann somit über den Aufruf

```
Database current
```

erfolgen, der die entsprechende `Database`–Instanz zurückliefert. Durch die Nachricht

```
Database current: aName
```

kann die `Database`–Instanz, die in `UserGlobals` unter dem Schlüssel `aName` abgelegt ist, zur aktuellen Fallbasis erklärt werden.

Die Fälle (im folgenden auch mit „ApplicationObjects“ bezeichnet), die in einer Fallbasis abgelegt werden sollen, müssen Instanzen der Klasse 'ApplicationObject' oder einer Unterklasse davon sein. Für die Klasse `ApplicationObject` sind die Instanzenvariablen `attributes`, `relations` und `searchKeys` definiert. Diese drei Variablen sind Dictionaries, die die fallbeschreibenden Attribute, Relationen sowie eventuell definierte Suchschlüsselattribute enthalten. (Die grundlegenden Datenstrukturen hierfür sind in [Kie92] näher beschrieben.)

Alle in die Fallbasis eingebrachten ApplicationObjects werden als Elemente einer gemeinsamen Menge abgespeichert. Diese Menge wird bei der Generierung der Fallbasis als eine Instanz der Klasse 'SetOfApplicationObjects' oder einer Unterklasse davon angelegt. `SetOfApplicationObjects` ist eine Unterklasse der OPAL–Klasse 'Set'. Eine `SetOfApplicationObjects`–Instanz kann nur ApplicationObjects enthalten.

Auf einer solchen Fallmenge kann der Benutzer dann $k$-d-Bäume als Zugriffspfadstrukturen für die Anwendungsobjekte erzeugen. Die Datenstruktur des $k$-d-Baumes ist momentan die einzige implementierte Suchstruktur. Sie wurde als Unterklasse von `SearchStructure` implementiert (siehe Abschnitt 12.3). Weitere Zugriffspfade, die im Laufe der Zeit eventuell noch zusätzlich bereitgestellt werden, sollten dann ebenfalls als Unterklassen von `SearchStructure` definiert werden.

Alle von einem Benutzer erzeugten Zugriffspfade werden in der Fallbasis in einer gemeinsamen Menge abgelegt. Diese Menge ist eine Instanz der Klasse `SetOfSearchStructures`, die ebenfalls eine Unterklasse der OPAL–Klasse 'Set' ist. Die Menge ist in jeder Fallbasis (einer `Database`–Instanz) unter dem Namen `#SearchStructures` abgespeichert.

### 12.2.2 Generierung einer Fallbasis

Mit dem Aufruf

```
Database generate: aName withTables: anArrayOfArrays
```

wird eine neue Fallbasis generiert bzw. eine bereits vorhandene neu initialisiert. `aName` ist der Name, unter dem die Fallbasis als `Database`–Instanz im `UserGlobals`–Dictionary abgelegt wird. `anArrayOfArrays` ist ein Array, dessen Einträge zweielementige Arrays der Form `#[#setName, aSetClass]` sind. Durch diese Einträge spezifiziert der Benutzer die Sets in der Fallbasis, in denen die ApplicationObjects gespeichert werden sollen.

Hierzu muß für jeden Set in dem Parameter `#setName` ein Name (als ein Symbol) angegeben werden, unter dem der Set in der Fallbasis angesprochen werden kann. Die Set–Namen

müssen innerhalb einer `Database`–Instanz eindeutig sein. `aSetClass` spezifiziert den Set, der die ApplicationObjects aufnehmen soll. Der Wert dieses Parameters ist entweder die Klasse `SetOfApplicationObjects` selbst oder eine ihrer Unterklassen, die dann jedoch vorher in der DB erzeugt worden sein muß. Durch die Nachricht

```
SetOfApplicationObjects subclass: aString
                        constrainedTo: anApplicationClass
```

kann eine solche Unterklasse von `SetOfApplicationObjects` angelegt werden. Sie steht dann als OPAL–Klasse mit dem Namen `aString` in der DB zur Verfügung. `anApplicationClass` ist dabei entweder die Klasse `ApplicationObject` oder eine ihrer (vom Benutzer zu definierenden) Unterklassen. Die in `anApplicationClass` spezifizierte Klasse besagt, daß Set–Instanzen dieser neuen Unterklasse ausschließlich Instanzen dieser Klasse oder einer ihrer Unterklassen enthalten können (vgl. dazu auch [Serv86] bzw. OPAL–Handbücher).

**Empfehlung:**

> *Prinzipiell sollten Anwendungsfälle immer durch Instanzen einer* Unterklasse von 'ApplicationObject'*, die der Benutzer zu diesem Zweck in der DB entsprechend definieren muß, dargestellt werden!*

Beispiel:

Es soll eine Fallbasis mit dem Namen 'CaseBase' erzeugt werden, in der die Fälle als Instanzen einer benutzerdefinierten Klasse mit dem Namen 'Workpiece' abgespeichert werden können. (Wir nehmen an, daß diese Klasse als Unterklasse von 'ApplicationObject' bereits in der DB definiert sei.) Durch die Sequenz

```
| aSetClass |
aSetClass := SetOfApplicationObjects subclass: 'SetOfWorkpieces'
                                     constrainedTo: Workpiece.
Database generate: #CaseBase withTables: #[ #[#MyWorkpieces, aSetClass] ].
```

wird dann im `UserGlobals`–Dictionary von GemStone eine neue `Database`–Instanz unter dem Schlüssel `#CaseBase` angelegt, die gleichzeitig auch zur aktuellen Fallbasis wird. Für die neue Fallbasis erfolgt eine Initialisierung ihrer Suchschlüsselliste mit dem Namen `#SearchKeyList` und einer (leeren) `SetOfSearchStructures`–Instanz mit dem Namen `#SearchStructures`, in der alle zukünftig erzeugten Zugriffsstrukturen abgespeichert werden.

Weiterhin wird der ebenfalls noch leere Set `#MyWorkpieces` (eine Instanz der neu definierten Klasse `SetOfWorkpieces`) angelegt, der die `Workpiece`–Instanzen aufnehmen kann.

### 12.2.3 Aktualisieren einer Fallbasis

Jede Fallbasis ist als Instanz der Klasse `Database` ein SymbolDictionary. Somit kann der Zugriff auf ihre einzelnen Komponenten durch die für Dictionaries definierte Methode `at:` erfolgen. Die Menge aller in der aktuellen Fallbasis definierten Zugriffspfade erhält man beispielsweise durch

```
Database current at: #SearchStructures
```

Ist 'CaseBase' gerade die aktuelle Fallbasis, so liefert

```
Database current at: #MyWorkpieces
```

alle in 'CaseBase' abgespeicherten `Workpiece`–Instanzen.

Um neue Fälle in die Fallbasis zu übernehmen, müssen sie in den Set, der die ApplicationObjects enthält, eingefügt werden. Dies kann mit Hilfe der für 'SetOfApplicationObjects' definierten Instanzenmethode `store:` erfolgen.

Beispiel:
Mit

```
Database current: #CaseBase.
(Database current at: #MyWorkpieces) store: aWorkpiece
```

wird die `Workpiece`–Instanz `aWorkpiece` in die aktuelle Fallbasis aufgenommen und in dem Set `#MyWorkpieces` abgespeichert.

Falls auf der Fallmenge Suchstrukturen definiert sind, erfolgt automatisch auch eine Aktualisierung dieser Strukturen mit dem neuen Fall.

Analog verhält es sich, wenn ein bekannter Fall aus der Fallbasis zu entfernen ist. Dies kann durch die Nachricht `erase: anApplicationObject` an die entsprechende Fallmenge veranlaßt werden. Der gelöschte Fall wird dann auch aus den vorhandenen Suchstrukturen entfernt.

### 12.2.4 Einfaches Retrieval von Fällen

Für das einfache Retrieval von Fällen aus einer Fallbasis stellt GemStone für Sets die Methoden `select: aBlock` und `reject: aBlock` zur Verfügung. Der angegebene Block erwartet als Eingabe jeweils ein Element aus dem betrachteten Set. In dem Block können Bedingungen spezifiziert werden, die die Variablenwerte der Set–Elemente erfüllen müssen. Es erfolgt somit in beiden Fällen eine Untermengenbildung des ursprünglichen Sets.

Schickt man z.B. die Nachricht `select: aBlock` an eine `SetOfApplicationObjects`–Instanz einer Fallbasis, so wird als Ergebnis eine ebensolche Set–Instanz zurückgeliefert, die alle Fälle enthält, für die die Auswertung des Blockes `aBlock` den Wert `true` ergibt.

Dagegen enthält das Ergebnis der Nachricht `reject: aBlock` alle die Elemente des ursprünglichen Sets, für die der Block `aBlock` zu `false` ausgewertet wird.

## 12.3 k-d-Bäume

### 12.3.1 Die Klasse 'KDTree'

Die Datenstruktur des $k$-d-Baumes wurde durch die Klassen `KDTree`, `KDTreeNode`, `InternalNode` und `TerminalNode` implementiert. Ein $k$-d-Baum ist eine Instanz der Klasse `KDTree`, die wie folgt definiert ist:

```
SearchStructure subclass: 'KDTree'
instVarNames: #( 'root' 'searchKeys' 'maxBucketSize'
                 'isOptimized' 'globalDict')
classVars: #()
constraints: #[  #[ #root, KDTreeNode],
                 #[ #searchKeys, SymbolSet],
                 #[ #maxBucketSize, Integer],
                 #[ #isOptimized, Boolean],
                 #[ #globalDict, SymbolDictionary] ]
```

Repräsentiert wird ein $k$-d-Baum durch seine Wurzel (`root`), die ein KDTreeNode sein muß. Ein Baumknoten ist entweder ein innerer Knoten oder ein Blattknoten. Innere Knoten sind Instanzen der Klasse `InternalNode` und Blattknoten Instanzen von `TerminalNode`. Die Klasse `KDTreeNode` ist die abstrakte Oberklasse zu `InternalNode` und `TerminalNode`, so daß diese Klassen in folgender Hierarchie zueinander stehen:

```
Object
  KDTreeNode
    InternalNode
    TerminalNode
```

Die Klassendefinitionen lauten

```
KDTreeNode subclass: 'InternalNode'
instVarNames: #( 'loSon' 'hiSon'
                 'discriminator' 'partitionValue')
classVars: #()
constraints: #[  #[ #loSon, KDTreeNode],
                 #[ #hiSon, KDTreeNode],
                 #[ #discriminator, Symbol] ]
```

bzw.

```
KDTreeNode subclass: 'TerminalNode'
instVarNames: #( 'bucket')
classVars: #()
constraints: #[  #[ #bucket, SetOfApplicationObjects] ]
```

`InternalNode`–Instanzen enthalten demnach zwei weitere Knoten `loSon` und `hiSon`, die die beiden Söhne eines inneren Knotens repräsentieren. Zusätzlich wird in jedem InternalNode der Name des Diskriminatorattributes (`discriminator`) und der zugehörige Partitionswert (`partitionValue`) abgespeichert. Die Variable `bucket` der Klasse `TerminalNode` enthält für einen TerminalNode alle ApplicationObjects, die im Bucket dieses Blattknotens liegen.

Die `KDTree`–Instanzenvariable `root` enthält den eigentlichen $k$-d-Baum. Die weiteren Variablen einer `KDTree`–Instanz beinhalten:

- `searchKeys` : die Namen aller im $k$-d-Baum verwendeten Suchschlüsselattribute
- `maxBucketSize` : Bucketgröße des Baumes
- `isOptimized` : `true`, falls der Baum ausbalanciert ist, und `false` sonst
- `globalDict` : Dictionary, das bei der Suche dazu verwendet wird, benötigte Hilfsvariablen temporär zu speichern

Die aus der Oberklasse 'SearchStructure' ererbte Instanzenvariable `dbTableName` enthält für jede `KDTree`–Instanz den Namen der Menge von ApplicationObjects aus der Fallbasis, für die der Baum erzeugt wird.

### 12.3.2 Erzeugung eines k-d-Baumes

Ein $k$-d-Baum kann für eine in der aktuellen Fallbasis vorhandene Menge von ApplicationObjects durch die Nachricht

```
(Database current at: aSetName)
    createKDTreeOn: aSymbolArray bucketSize: anInteger
```

erzeugt werden. Das Symbol `aSetName` bezeichnet dabei den Namen, unter dem die Menge der ApplicationObjects, für die eine Suchstruktur aufgebaut werden soll, in der aktuellen Fallbasis (`Database current`) abgelegt ist. Dieser Name wird der Variablen `dbTableName` der neu erzeugten `KDTree`–Instanz zugewiesen.

In dem Array `aSymbolArray` sind die Namen der ApplicationObject–Attribute anzugeben, die in dem $k$-d-Baum als Suchschlüssel verwendet werden. Alle Attribute, die solche Suchschlüssel darstellen, müssen unter ihrem Namen entweder im Dictionary `attributes` oder im Dictionary

`searchKeys` der ApplicationObjects vorhanden sein. Über den Parameter `anInteger` wird die Bucketgröße für den Baum festgelegt. Die Angabe der Bucketgröße ist optional. Wird nur die Methode `createKDTreeOn:` verwendet, so wird automatisch ein Baum mit der Default–Bucketgröße 1 generiert.

Der erzeugte $k$-d-Baum ist ein idealer Baum. Er wird wie in Kapitel 8 beschrieben aufgebaut. Nachdem er generiert worden ist, wird er in der aktuellen Fallbasis in dem Set `(Database current at: #SearchStructures)` abgespeichert. Zusätzlich werden alle im $k$-d-Baum angegebenen Suchattributnamen noch in die Liste `(Database current at: #SearchKeyList)` aufgenommen.

Beispiel:

Es sei 'CaseBase' wieder unsere aktuelle Fallbasis. Für die dort abgelegten Workpieces soll ein $k$-d-Baum mit einer Bucketgröße 2 für die Attribute `length` und `height` erzeugt und in der Fallbasis abgespeichert werden. Wenn wir davon ausgehen, daß jedes in 'CaseBase' abgespeicherte Workpiece diese beiden Attribute enthält, dann kann dies folgendermaßen erreicht werden:

```
(Database current at: #MyWorkpieces)
    createKDTreeOn: #[#length, #height] bucketSize: 2
```

In der eigentlichen Definition eines $k$-d-Baumes wird davon ausgegangen, daß der mehrdimensionale Suchschlüssel auch wirklich Schlüsseleigenschaft besitzt und somit jede Kombination von Suchattributwerten in der Menge der ApplicationObjects höchstens einmal auftritt. In realen Anwendungen des fallbasierten Schließens bedeutet diese Annahme jedoch eine nicht unerhebliche Einschränkung.

Aus diesem Grund können in unserer Implementierung auch Suchschlüsselkombinationen verwendet werden, die nicht eindeutig sind. Hierbei kann jedoch das Problem auftreten, daß irgendwann im Laufe des Baumaufbaus eine Menge von ApplicationObjects nicht mehr partitioniert werden kann, da alle Objekte die gleichen Suchattributwerte besitzen. Dies ist allerdings nur ein Terminierungsproblem, das wir dadurch gelöst haben, daß beim Auftreten einer solchen Situation einfach alle verbliebenen Objekte zusammen in einem Bucket abgespeichert werden. In diesem (und nur in diesem) Falle kann ein Bucket dann auch mehr Objekte enthalten als durch die Bucketgröße spezifiziert worden ist.

Das ist insofern unproblematisch, als Objekte mit gleichen Suchschlüsselwerten bei der Best–Match–Suche analytisch sowieso als ein einziges Objekt betrachtet werden können. Die Bucketgröße gibt somit nur an, wieviel Objekte mit *verschiedenen* Suchschlüsseln maximal in einem Blattknoten gespeichert sind.

### 12.3.3 Rebalancierung von k-d-Bäumen

Im allgemeinen bleibt die balancierte Struktur idealer Bäume beim Einfügen oder Löschen neuer ApplicationObjects nicht erhalten, da eine Rebalancierung aus Effizienzgründen nicht nach jeder Aktualisierung der Fallbasis vorgenommen werden kann. Es bleibt deshalb dem Benutzer überlassen, die Zugriffsstruktur(en) für die Fallbasis von Zeit zu Zeit selbst zu optimieren. Hierzu kann die für `KDTree`–Instanzen implementierte Methode `optimize:` verwendet werden. Man schickt an einen $k$-d-Baum `aKDTree`, der optimiert werden soll, einfach die Nachricht

```
aKDTree optimize.
```

Neu erzeugte KDTrees müssen nicht optimiert werden, da durch jede Baumgenerierung jeweils ein idealer $k$-d-Baum aufgebaut wird. Die Methode `optimize:` ist auch so implementiert, daß der zu optimierende Baum einfach durch einen neu erzeugten Baum für alle zum Zeitpunkt des Optimierungsaufrufes in der Fallbasis gespeicherten Fälle ersetzt wird.

### 12.3.4 Best–Match–Suche

Die Best–Match–Suche wurde im Rahmen dieser Diplomarbeit in zwei Versionen implementiert:

- die **starre** Version, in der die $m$ nächsten Nachbarn eines Anfrageobjektes gesucht werden, wobei der Parameter $m$ fest vorgegeben wird
- die **flexible** Version, die beliebig viele nächste Nachbarn inkrementell bestimmt

In beiden Fällen wird eine Nachricht an den zu durchsuchenden KDTree gesendet.

#### Starre Suche

Für die Suche nach den $m \geq 1$ nächsten Nachbarn eines Objektes wurde der in Kapitel 8 beschriebene Algorithmus von Friedman et al. implementiert. Um den $k$-d-Baum `aKDTree` nach den $m$ nächsten Nachbarn von `aQueryObject` zu durchsuchen, muß an den $k$-d-Baum `aKDTree` die Nachricht

```
aKDTree searchFor: aQueryObject m: m
```

geschickt werden. `aQueryObject` ist dabei ein ApplicationObject, das alle in `aKDTree` angegebenen Suchattribute enthalten muß. Der Parameter $m$ ist eine natürliche Zahl.

Für die Suche können auch k.o.–Kriterien in Form eines OPAL–Blockes, der ein ApplicationObject als Eingabeparameter erwartet, spezifiziert werden. Durch den Aufruf

```
aKDTree searchFor: aQueryObject m: m constraints: aBlock
```

kann ein solcher Block (hier mit `aBlock` bezeichnet) der Suche als Parameter übergeben werden. Der Block wird während des Suchvorganges für alle zu untersuchenden ApplicationObjects evaluiert, die sich dann je nach Auswertung als NN–Kandidaten qualifizieren oder nicht.

In der Instanzenvariable `globalDict` des zu durchsuchenden KDTrees stehen während der Suche folgende globale Variablen zur Verfügung:

- die aktuelle Nearest–Neighbor–Queue (unter dem Namen `#Queue`); jedes Element der Queue enthält einen der aktuellen $m$ NN–Kandidaten mit dem berechneten Ähnlichkeitswert
- die oberen und unteren Schranken des gerade betrachteten Baumknotens (`#Bounds`)
- eine Liste, in der über alle während der Suche untersuchten ApplicationObjects Protokoll geführt wird (`#Examined`)
- der Parameter $m$ (`#m`), die Bucketgröße (`#b`) und der Block, der die k.o.–Kriterien spezifiziert (`#selectionBlock`)
- eine Statistik über die Anzahl der während der Suche durchgeführten BOB– bzw. BWB–Tests sowie über die durchgeführten Vergleichsoperationen an inneren Knoten des $k$-d-Baumes (`#Statistics`)

Als Ergebnis der Suche wird ein Dictionary zurückgeliefert, das neben der Queue mit den $m$ nächsten Nachbarn noch die Liste aller untersuchten Baumknoten und die während der Suche erstellte Statistik enthält.

#### Inkrementelle Suche

Für die inkrementelle Suche wurde die Version von A. Broder implementiert (vgl. Kapitel 9). Sie gliedert sich in drei Phasen:

1. Aufruf der *Initialisierungsfunktion* zum Aufbau der Seed–Struktur
2. mehrmaliger Aufruf der *Generatorfunktion* zur inkrementellen Bestimmung der nächsten Nachbarn
3. *Beenden* des Suchvorganges

Der Suchvorgang wird mit dem Aufruf

```
aKDTree prepareForIncrSearch: aQueryObject
```

bzw.

```
aKDTree prepareForIncrSearch: aQueryObject constraints: aBlock
```

gestartet. Hierbei erfolgt die Initialisierung der Seed–Struktur wie in Kapitel 9 beschrieben. Nach dieser Initialisierungsphase können die $m$ nächsten Nachbarn zu `aQueryObject` durch $m$-maliges Wiederholen des Aufrufes

```
aKDTree incrSearchNext:aQueryObject
```

inkrementell bestimmt werden. Jeder dieser $m$ Aufrufe gibt als Ergebnis den jeweils nächsten Nachbarn mit dem berechneten Ähnlichkeitswert aus.
Ein Suchvorgang wird schließlich beendet mit

```
aKDTree finishIncrSearch.
```

Hierbei werden die temporären globalen Variablen aus der Instanzenvariable `globalDict` des KDTrees entfernt und als Ergebnis des gesamten Suchvorganges ein Dictionary ausgegeben, das die Ausprägung der Seed–Struktur nach dem letzten Aufruf von `incrSearchNext` enthält.

In der Seed–Struktur stehen während der Suche jeweils die aktuellen geometrischen Schranken der inneren Knoten (`#Bounds`), die Rekursionstrace (`#Trace`) und ebenfalls eine Statistik wie oben (`#Statistics`) zur Verfügung.

## 12.4 Das Typkonzept

### 12.4.1 Wertebereiche der Repräsentationsattribute

Fälle werden als `ApplicationObject`–Instanzen dargestellt (vgl. [Kie92]). Ihre Repräsentationsattribute sind Instanzen der Klasse `Attribute`, die in den beiden `ApplicationObject`–Instanzenvariablen `attributes` und `searchKeys` unter ihrem Attributnamen eingetragen werden. Eine Instanz der Klasse `Attribute` besitzt folgende Instanzenvariablen:

- `name`: enthält den Attributnamen
- `type`: gibt die Bezeichnung des (benutzerdefinierten) Typs an, der diesem Attribut zugeordnet ist
- `default`: Wert aus dem Wertebereich des in `type` angegebenen Typs; dieser Wert wird als Default–Wert für das Attribut interpretiert
- `value`: Attributwert aus dem Wertebereich des angegebenen Typs

Die beiden Variablen `value` und `default` enthalten Instanzen von Smalltalk-80– bzw. OPAL–Klassen, die die einfachen und komplexen Attributwerte darstellen (siehe Kapitel 3, Abschnitt 3.2). *Einfache* Attributwerte sind Instanzen folgender (System–)Klassen:

- `Boolean`
- `Number`
- `Integer`
- `Float`
- `String`
- `Symbol`

Zur Darstellung *komplexer* Attributwerte wie Mengen und Listen wurden zusätzlich die Klassen `UnorderedSet`, `OrderedSetType`, `OrderedSet`, `List` und `IntervalType` implementiert. Diese Klassen sind Spezialisierungen der Klasse `Array` (in OPAL) bzw. `OrderedCollection` (in Smalltalk-80) und stehen in folgender hierarchischer Beziehung zueinander:

UnorderedSet( 'constraint')
  OrderedSetType
    OrderedSet( 'order')
    List
      IntervalType

### Die Klasse 'UnorderedSet'

Ungeordnete Mengen werden als Instanzen der Klasse `UnorderedSet` dargestellt. Ihre Definition lautet (in OPAL)

```
Array subclass: 'UnorderedSet'
  instVarNames: #( 'constraint')
  classVars: #()
  constraints: #[ #[ #constraint, Class] ].
```

Eine neue Instanz dieser Klasse kann mit der Nachricht

```
UnorderedSet withElements: aCollection constrainedTo: aClass
```

generiert werden. Der Parameter `aCollection` ist eine Collection, die die Elemente der neu erzeugten Menge enthält. Dabei ist zu beachten, daß jede `UnorderedSet`–Instanz die Mengeneigenschaft erfüllen muß, d.h., jedes Element aus `aCollection` darf nur einmal vorkommen. Weiterhin müssen alle Elemente Instanzen der im Parameter `aClass` angegebenen Klasse oder einer ihrer Unterklassen sein. Auf den Elementen eines UnorderedSets ist keine Ordnung vorgegeben. Die Wertegleichheit zwischen zwei UnorderedSets ist über die Gleichheit ihrer Elementmengen definiert.

### Die Klasse 'OrderedSetType'

Die Klasse `OrderedSetType` repräsentiert Mengen, auf denen eine Ordnung definiert ist. Diese Klasse ist eine abstrakte Oberklasse für die Klassen `OrderedSet` und `List`. Sie wurde aus modellierungstechnischen Gründen in dieser Form implementiert. Deshalb sollten keine Instanzen von dieser Klasse erzeugt werden. Mengen, auf deren Elementen eine Ordnung definiert ist, sollten ausschließlich Instanzen der Klassen `OrderedSet`, `List` oder `IntervalType` sein.

### Die Klasse 'OrderedSet'

Die Definition der Klasse `OrderedSet` lautet

```
Array subclass: 'OrderedSet'
  instVarNames: #( 'order')
  classVars: #().
```

Instanzen dieser Klasse stellen geordnete Mengen dar, deren Ordnungen *explizit* in der Instanzenvariable `order` angegeben werden müssen. Für die Elemente solcher Mengen gelten darüber hinaus dieselben Bedingungen wie für die ungeordneter Mengen. Die Nachricht

```
OrderedSet withElements: aCollection
           constrainedTo: aClass
           order: aBlockOrList
```

erzeugt analog zu oben eine `OrderedSet`–Instanz mit den in `aCollection` angegebenen Elementen. Zusätzlich muß im Parameter `aBlockOrList` explizit noch eine Ordnungsrelation zwischen den Elementen spezifiziert werden. `aBlockOrList` ist hierbei entweder ein Smalltalk– bzw. OPAL–Block mit zwei Eingabeparametern, oder eine Liste, in der die <–Relationen, die zwischen den Elementen bestehen, aufgeführt sind.

Ist `aBlockOrList` eine Block–Instanz, so werden in den beiden Argumenten zwei Elemente der geordneten Menge erwartet. Die Auswertung des Blockes muß genau dann den Wert `true` liefern, wenn das erste Argument kleiner ist als das zweite Argument. Ein Wert $x$ ist somit genau dann kleiner als ein Wert $y$, falls der Aufruf

```
aBlockOrList value: x value: y
```

den Wert `true` liefert.

Sollen hingegen alle zwischen den Mengenelementen bestehenden Relationen explizit in einer Liste angegeben werden, so muß in dem Parameter `aBlockOrList` eine Instanz der zu diesem Zweck implementierten Klasse 'OrderingList' angegeben werden. Diese Klasse ist eine Array–Spezialisierung. Ihre Instanzen enthalten als Elemente ausschließlich 2-elementige Arrays der Form `#[value1, value2]`, durch die die Relation *value1* < *value2* dargestellt wird. Ein Wert $x$ ist genau dann kleiner als ein Wert $y$, falls die in `aBlockOrList` angegebene `OrderingList`–Instanz das Array–Element `#[x, y]` enthält.

### Die Klasse 'List'

Die Klasse `List` modelliert geordnete Mengen in Form von Listen. Die Klassendefinition lautet

```
OrderedSetType subclass: 'List'
  instVarNames: #()
  classVars: #().
```

Eine Liste ist dadurch gekennzeichnet, daß ihre Ordnung implizit durch die Anordnung ihrer Elemente vorgegeben ist. Das heißt, ein Element $x$ einer `List`–Instanz ist genau dann kleiner als ein weiteres Element $y$, falls der Index, unter dem $x$ in dieser Liste abgespeichert ist, kleiner ist als der von $y$. Eine solche Liste kann durch

```
List withElements: anArray constrainedTo: aClass
```

erzeugt werden. Die Reihenfolge der in `anArray` angegebenen Listenelemente wird beibehalten. Auch hier gilt, daß jedes Element aus `anArray` eine Instanz der Klasse `aClass` (oder einer Unterklasse davon) sein muß und nur einmal in der neu erzeugten `List`–Instanz vorkommt.

### Die Klasse 'IntervalType'

`IntervalType` ist eine Unterklasse von `List`. Instanzen dieser Klasse können mit

```
IntervalType withElements: anArrayOfTwoElements
             constrainedTo: aClass
```

erzeugt werden, wobei der Parameter `anArrayOfTwoElements` ein Array ist, das genau zwei Elemente enthält. Das erste Element gibt dabei die Intervalluntergrenze an und das zweite Element die Intervallobergrenze. Durch eine `IntervalType`–Instanz wird implizit die Liste repräsentiert, die alle Werte zwischen diesen beiden Grenzen einschließlich der Grenzen selbst enthält. Als Einschränkung ist zu beachten, daß solche Intervalle nur für Werte der System–Klassen `Magnitude`, `String` oder deren Unterklassen definiert werden können. Weiterhin muß der Wert, der die Intervalluntergrenze bezeichnet, kleiner oder gleich der Obergrenze sein.

Die Ordnungsrelation zwischen zwei Werten $x$ und $y$ aus einem solchen Intervall ist dann durch die im System implementierte Ordnung für Instanzen der erwähnten Systemklassen vorgegeben.

Mit Hilfe der beschriebenen Klassen ist es möglich, beliebig komplexe Wertebereiche zu konstruieren. Diese können dann zur Spezifikation benutzerdefinierter Typen herangezogen werden. Dies wird im nächsten Abschnitt beschrieben. Im Anhang A.2 sind einige Beispiele zur Definition solcher Wertebereiche aufgeführt.

### 12.4.2 Benutzerdefinierte Typen

Benutzerdefinierte Typen, wie sie in Kapitel 10 spezifiziert wurden, werden durch Instanzen der Klasse `ApplicationType` dargestellt. Diese ist definiert durch

```
Object subclass: 'ApplicationType'
  instVarNames: #('name' 'supertype' 'similarity'
                  'range' 'valueType')
  classVars: #('list')
  constraints: #[ #[ #name, Symbol], #[ #supertype, Symbol] ] .
```

Die Funktion der Klassenvariable `list`, die eine globale Liste aller benutzerdefinierten Typen enthält, wird im nachfolgenden Abschnitt näher erläutert.

Die Instanzenvariable `name` enthält ein Symbol, das den benutzerdefinierten Typ eindeutig identifiziert. Sie entspricht dem Label des Typs. Unter diesem Namen kann der Typ z.B. in der Variablen `type` einer `Attribut`–Instanz referenziert werden. Die Variable `supertype`, ebenfalls ein Symbol, bezeichnet den Namen der `ApplicationType`–Instanz, die als Supertyp übergeordnet ist. Gemäß der Spezifikation aus Kapitel 10 dürfen die Werte von `name` und `supertype` nicht identisch sein.

Die Variable `similarity` enthält das Ähnlichkeitsmaß für die Werte des definierten Typs. Wie ein solches Ähnlichkeitsmaß spezifiziert werden kann, wird weiter unten erläutert.

Der Wert von `valueType` ist eine Klasse, die angibt, welcher Art die Werte des definierten Typs sein können. Die Werte, die dieser Variablen zugewiesen werden können, sind die in Abschnitt 12.4.1 beschriebenen Klassen für einfache und komplexe Attributwerte. `valueType` beschreibt somit den Wertebereich, der alle möglichen Instanzen der angegebenen Klasse umfaßt.

Dieser Bereich kann durch die Variable `range` auf eine bestimmte Menge von Instanzen der in `valueType` angegebenen Klasse eingeschränkt werden. Dies kann durch die Konstruktion komplexer Wertebereiche, wie sie in Abschnitt 12.4.1 beschrieben ist, erfolgen. Hierbei ist zu beachten, daß diese Wertebereiche ausschließlich geordnete Mengen sein dürfen, in deren Instanzenvariable `constraint` zudem dieselbe Klasse enthalten sein muß wie in `valueType`. Außerdem muß der Wertebereich eine Teilmenge der in der Variablen `range` des Supertyps spezifizierten Menge sein. Ebenso darf `valueType` höchstens Spezialisierungen des `valueType`–Wertes des Supertyps enthalten.

Sind in den Variablen `similarity`, `valueType` oder `range` keine Werte spezifiziert, so erbt der definierte ApplicationType über einen Vererbungsmechanismus die entsprechenden Werte von seinem Supertyp.

Eine als ein benutzerdefinierter Typ interpretierbare `ApplicationType`–Instanz kann durch zwei Methoden generiert werden. Entweder durch die Nachricht

```
ApplicationType new: aName
                supertype: aSymbol
                similarity: aBlockOrList
                range: anOrderedSet
```

oder durch

```
ApplicationType new: aName
                supertype: aSymbol
                similarity: aBlockOrList
                valueType: aValueClass .
```

Beide Methoden belegen die Instanzenvariablen `name`, `supertype` und `similarity` mit den in den Parametern `aName` bzw. `aSymbol` bzw. `aBlockOrList` übergebenen Werten. Zu beachten ist hierbei, daß in der globalen Typenliste noch kein Typ mit dem Namen `aName` existieren darf. Es muß jedoch bereits ein ApplicationType mit dem Namen `aSymbol` dort vorhanden sein.

Im ersten Fall wird durch die Belegung der Variablen `range` mit einer geordneten Menge eine Wertebereichseinschränkung vorgenommen. Die Variable `valueType` erhält automatisch den `constraint`–Wert von `anOrderedSet`.

Im zweiten Fall erfolgt keine Einschränkung des Wertebereiches. Hier muß die Variable `valueType` explizit mit einer Klasse belegt werden.

Werden für die Variablen `similarity`, `range` oder `valueType` Nil–Werte übergeben, so werden diese als undefinierte Werte interpretiert. Das bedeutet ferner, daß hier — nach dem Vererbungsprinzip aus Kapitel 10 — die entsprechenden Werte des Supertyps übernommen werden sollen.

### 12.4.3 Die globale Typen–Liste

In der Klassenvariable `list` werden alle benutzerdefinierten Typen abgespeichert. Diese stehen dann global für alle in der Datenbank erzeugten Anwendungs–Fallbasen zur Verfügung. Die Liste ist eine Instanz der Klasse `ListOfApplicationTypes`, einer Spezialisierung der Systemklasse Dictionary. In ihr werden die `ApplicationType`–Instanzen unter ihrem jeweiligen Label, dem angegebenen Namen, abgelegt. Diese Liste kann durch die Nachricht

```
ListOfApplicationTypes initialize
```

initialisiert werden. Dabei werden bereits einige vordefinierte ApplicationTypes in der Liste abgespeichert. Im Anhang A.2 sind die vordefinierten Typen wie auch eine beispielhafte Definition eines ApplicationTypes aufgeführt.

Eine `ApplicationType`–Instanz `anApplicationType` wird durch die Nachricht

```
anApplicationType addToListOfTypes
```

unter ihrem Namen in der globalen Typenliste abgespeichert.

Durch den Aufruf

```
anApplicationType removeFromListOfTypes
```

wird die Instanz aus der Liste entfernt und mit ihr rekursiv alle Typen, deren Supertyp diese Instanz ist.

Auf eine in der Liste vorhandene `ApplicationType`–Instanz kann über ihren Namen zugegriffen werden. Dies kann durch

```
ApplicationType named: aSymbol
```

erfolgen, wobei der Parameter `aSymbol` den Typnamen angibt. Zurückgeliefert wird die entsprechende `ApplicationType`–Instanz. (Im Protokoll der Klassenmethoden von `ApplicationType` sind unter der Kategorie 'Accessing BasicTypes' noch zusätzliche Methoden implementiert, die den Zugriff speziell auf die vordefinierten Typen ermöglichen.)

### 12.4.4 Implementierung von Koordinatenähnlichkeiten

Die Koordinatenähnlichkeiten der Fälle müssen über Ähnlichkeitsmaße für die einzelnen Suchattribute der ApplicationObjects modelliert werden. Diese Maße sind in den `ApplicationType`–Instanzen zu definieren, die die benutzerdefinierten Datentypen für die Attribute repräsentieren. Zur Implementierung von Ähnlichkeitsmaßen für die ApplicationTypes stehen zwei Mechanismen zur Verfügung:

- Angabe einer Ähnlichkeitsmatrix für die Werte des spezifizierten Wertebereiches
- Implementierung als zweistellige Funktion in Form eines Smalltalk– bzw. OPAL–Blockes

In jedem Falle müssen die implementierten Maße den Bedingungen aus Kapitel 8 (Abschnitt 8.4) genügen.

### Die Klasse 'SimilarityList'

Eine Ähnlichkeitsmatrix, die für je zwei Werte des Wertebereiches die entsprechende Ähnlichkeit enthält, muß als eine Instanz der OPAL–Klasse `SimilarityList` angelegt werden. Diese Klasse ist eine Array–Unterklasse, deren Instanzen nur 3-elementige Arrays der Form `#[value1, value2, similarityValue]` enthalten, wobei `value1` und `value2` zwei Werte aus dem Wertebereich des Typs sind. `similarityValue` ist ein Wert zwischen 0 und 1, der die Ähnlichkeit zwischen den beiden Werten `value1` und `value2` angibt, so daß gilt: $\mu(value1,value2) = similarityValue$. Mit der Nachricht

```
SimilarityList withAll: aCollectionOfArrays
```

kann eine solche `SimilarityList`–Instanz generiert werden. `aCollectionOfArrays` ist eine Collection 3-elementiger Arrays, die jeweils eine Ähnlichkeitsrelation zwischen zwei Werten repräsentieren.

Wurde z.B. eine Ähnlichkeitsmatrix `aSimilarityList` erzeugt, so erhält man die Ähnlichkeit zwischen zwei Werten $x$ und $y$ durch den Aufruf

```
aSimilarityList value: x value: y.
```

Falls $x = y$ gilt, wird der Wert 1 ausgegeben. Sind $x$ und $y$ verschieden und ist in `aSimilarityList` ein Ähnlichkeitswert für $x$ und $y$ spezifiziert, so wird dieser zurückgeliefert, ansonsten der Wert 0.

Eine in dieser Form dargestellte Ähnlichkeitsmatrix kann der Instanzenvariable `similarity` eines ApplicationTypes zugewiesen werden. Alle Ähnlichkeitsbestimmungen zwischen Werten des ApplicationTypes erfolgen dann über diese Matrix.

### Implementierung als 'Block'–Instanz

Neben Ähnlichkeitsmatrizen gibt es noch die Möglichkeit, 2-stellige Funktionen zu spezifizieren, die die Ähnlichkeit zwischen zwei eingegebenen Werten berechnen. Die Ähnlichkeitswerte werden hier nicht explizit für jedes Wertepaar angegeben sondern über eine funktionale Vorschrift bestimmt.

Solche Ähnlichkeitsfunktionen müssen in Form von Block–Instanzen, die zwei Werte als Eingabeparameter erwarten, implementiert werden. Um mit Hilfe solcher Blocks die Ähnlichkeit zweier Werte $x$ und $y$ zu bestimmen, muß der Block mit diesen beiden Werten evaluiert werden. Dies geschieht dadurch, daß an den Block die Nachricht `value: x value: y` geschickt wird. Der Wert des evaluierten Blockes ist dann ein Wert aus dem Intervall $[0, 1] \subseteq \mathcal{R}$, der als Ähnlichkeit zwischen den beiden Eingabewerten interpretiert werden kann.

Beispiel:

Das aus dem euklidischen Abstand abgeleitete Ähnlichkeitsmaß $f_e(x, y)$ für numerische Werte kann als ein 2-Argumenten–Block z.B. folgendermaßen implementiert werden (in dieser Diplomarbeit wurden noch weitere Implementierungen gebräuchlicher Ähnlichkeitsmaße in Form solcher Blocks vorgenommen; sie sind im Anhang A.1 aufgeführt):

```
[ :val1 :val2 | 1/ ( 1 + ((val1 - val2) abs) )].
```

Die Sequenz (in OPAL–Code)

```
| aBlock |
aBlock:=[ :val1 :val2 | 1/ ( 1 + ((val1 - val2) abs) )].
^ aBlock value: 2 value: 5
```

liefert als Ergebnis den Ähnlichkeitswert 1/4 für die Zahlen 2 und 5.

Ein Block, durch den das Ähnlichkeitsmaß einer `ApplicationType`–Instanz implementiert ist, muß der Instanzenvariablen `similarity` zugewiesen werden. Die Ähnlichkeiten zwischen Werten dieses Typs werden dann über die Auswertung des Blockes mit diesen Werten bestimmt.

## 12.5 Ähnlichkeitsmaße für Fälle

Nach Kapitel 8 wird die Ähnlichkeit zweier Fälle $X$ und $Y$ über das Ähnlichkeitsmaß $Sim(X,Y) = F(\mu_1,...,\mu_k)$ bestimmt, wobei das Argument der $k$-stelligen Funktion $F$ ein $k$-dimensionaler Ähnlichkeitsvektor $(\mu_1, ..., \mu_k)$ ist, dessen Komponenten die einzelnen Ähnlichkeiten der beiden Fälle in den $k$ Suchattributen enthalten. Diese Koordinatenähnlichkeiten werden durch Maße für die Attribute berechnet, die in den ApplicationTypes zu definieren sind (siehe Abschnitt 12.4).

Die Berechnung der Ähnlichkeit zwischen Fällen wurde zweistufig implementiert:

1. Berechnung der $k$ Koordinatenähnlichkeiten und Erstellung des entsprechenden Ähnlichkeitsvektors
2. Bestimmung der (Fall–)Ähnlichkeit durch Anwenden der Funktion $F$ auf den im 1. Schritt erzeugten $k$-dimensionalen Ähnlichkeitsvektor

Beispiel:

Es seien 'Case' eine benutzerdefinierte Unterklasse von 'ApplicationObject', durch die Fälle modelliert werden, und `theFirstCase` und `theSecondCase` zwei Instanzen dieser Klasse. Innerhalb der implementierten Suchalgorithmen erfolgt dann die Ähnlichkeitsberechnung für die Fälle `theFirstCase` und `theSecondCase` durch den Aufruf

```
theFirstCase similarityTo: theSecondCase keys: aSymbolSet.
```

Die für ApplicationObjects implementierte Instanzenmethode `similarityTo:keys:` berechnet zunächst für die in `aSymbolSet` spezifizierten $k$ Suchattribute die Koordinatenähnlichkeiten zwischen `theFirstCase` und `theSecondCase` und erzeugt daraus einen $k$-dimensionalen Ähnlichkeitsvektor `aSimVector`. Danach schickt sie die Nachricht

```
theFirstCase class similarity: aSimVector
```

an die Klasse von `theFirstCase` und gibt den dort in Abhängigkeit vom Ähnlichkeitsvektor `aSimVector` berechneten Wert aus.

Die Ähnlichkeitsfunktion $F$ muß somit durch die Klassenmethode `similarity:` (im Beispiel für die Klasse 'Case') implementiert werden. Als *Default–Maß* ist für die Klasse `ApplicationObject` folgende Ähnlichkeitsfunktion implementiert:

$$F(\mu_1, \ldots, \mu_k) = \frac{1}{k} \sum_{i=1}^{k} \mu_i$$

Das heißt:

> *Der Benutzer muß diese Methode, falls das Default–Maß nicht übernommen werden soll, je nach Anwendung in der Klasse* reimplementieren, *durch die er seine Fälle modelliert. Diese Klasse sollte deshalb auch immer eine* Unterklasse von 'ApplicationObject' *sein!*

Bei der Reimplementierung ist auf die Gewährleistung der in Kapitel 8 (Abschnitt 8.4) für $F$ geforderten Eigenschaften zu achten.

# Anhang

# Anhang A

# Implementierungsbeispiele

## A.1 Ähnlichkeitsmaße

Im Zusammenhang mit der Implementierung des Typkonzepts wurden einige gebräuchliche Ähnlichkeitsmaße in Form von OPAL–Blöcken bereits implementiert (vgl. Kapitel 12). Sie stehen dadurch auch dem Anwender zur Verfügung (so kann er sie z.B. Wertebereichen zuordnen).

Hierzu enthält die Datenbank eine Unterklasse von `Object` mit dem Namen `SimilarityBlock`. Sie wurde ausschließlich zur Organisation der Methoden eingerichtet, in denen die Ähnlichkeitsmaße implementiert sind. Werden diese Methoden als Nachrichten an die Klasse `SimilarityBlock` geschickt, so erzeugen sie die entsprechenden OPAL–Blockinstanzen, die zu ihrer Evaluation jeweils zwei Eingabewerte erwarten und als Ergebnis einen Wert für die Ähnlichkeit der beiden Werte zurückliefern.

Durch die Nachricht

```
SimilarityBlock euclid
```

wird z.B. ein OPAL–Block generiert, durch den das aus der euklidischen Abstandsfunktion abgeleitete Ähnlichkeitsmaß für numerische Werte $f_e(x,y) = 1/(1+|x-y|)$ implementiert ist.

Im folgenden sind weitere in dieser Form implementierte Ähnlichkeitsmaße mit den entsprechenden Aufrufen aufgeführt.

- das Default–Ähnlichkeitsmaß $f_d$ für zwei *beliebige* Werte:

  $$f_d(x,y) = \begin{cases} 1 & , \text{ falls } x = y \\ 0 & , \text{ sonst} \end{cases}$$

  Aufruf: `SimilarityBlock default`

- Ähnlichkeitsmaße für *boolesche* Werte:

  1. *symmetrisch*:

     $$f_s = \begin{cases} 1 & , \text{ falls } a = b \\ 0 & , \text{ sonst} \end{cases}$$

     Aufruf: `SimilarityBlock symmetricBool`

  2. *asymmetrisch*:

     $$f_{as} = \begin{cases} 1 & , \text{ falls } a = b = \text{true} \\ c & , \text{ falls } a = b = \text{false} \\ 0 & , \text{ sonst} \end{cases} \qquad (c \in [0,1] \subseteq \mathcal{R})$$

     Aufruf: `SimilarityBlock asymmetricBool: c`

- ein Ähnlichkeitsmaß $f_{st}$ für *Strings*:

  Aufruf: `SimilarityBlock stringSpelling`

  Dieses Maß, das die Schreibweise zweier Strings miteinander vergleicht und daraus einen Ähnlichkeitswert berechnet, wurde unter Verwendung der Methode `spellAgainst: aString`, die in Smalltalk-80 für `CharacterArray`–Instanzen definiert ist, implementiert.

## A.2 ApplicationTypes

Durch den Aufruf

```
ListOfApplicationTypes initialize
```

wird die globale Typenliste in der Datenbank mit folgenden vordefinierten Typen (als `ApplicationType`–Instanzen) initialisiert:

- *Nulltyp* (Aufruf: `ApplicationType createNilType`)
- *Number* (Aufruf: `ApplicationType createNumberType`)
- *Float* (Aufruf: `ApplicationType createFloatType`)
- *Integer* (Aufruf: `ApplicationType createIntegerType`)
- *String* (Aufruf: `ApplicationType createStringType`)
- *Symbol* (Aufruf: `ApplicationType createSymbolType`)
- *SymmetricBoolean* (Aufruf: `ApplicationType createSymmetricBooleanType`)
- *AsymmetricBoolean* (Aufruf: `ApplicationType createAsymmetricBooleanType: 0.2`)

Im folgenden ist ein kleines Beispiel zur Definition eines benutzerdefinierten Typs angegeben. Es soll ein Typ mit Namen 'Ampel' erzeugt werden, dessen (geordneter) Wertebereich ungeordnete Mengen von Symbolen enthält. Die für diesen Typ gültigen Werte seien dabei auf die Mengen {rot}, {gelb}, {rot, gelb} und {grün} beschränkt. Eine Ordnung sei für diese vier Mengen gegeben durch die Relationen

$$\{\text{rot}\} < \{\text{gelb}\} < \{\text{rot, gelb}\} < \{\text{grün}\}.$$

Ein solcher Wertebereich kann z.B. durch folgende Aufrufsequenz (in OPAL–Code) erzeugt werden:

```
| set1 set2 set3 set4 anArray |
\Definition der vier Symbolmengen.\
set1 := UnorderedSet withElements: #[#rot] constrainedTo: Symbol.
set2 := UnorderedSet withElements: #[#gelb] constrainedTo: Symbol.
set3 := UnorderedSet withElements: #[#rot, #gelb] constrainedTo: Symbol.
set4 := UnorderedSet withElements: #[#grün] constrainedTo: Symbol.
\Aufbau einer Liste, die auch die Ordnungsrelationen repräsentiert.\
anArray := Array new.
anArray addLast: set1; addLast: set2; addLast: set3; addLast: set4.
^ List withElements: anArray constrainedTo: UnorderedSet.
```

Der Wertebereich wird dadurch als Liste ungeordneter Mengen angegeben, die somit gleichzeitig auch die Ordnungsrelationen zwischen diesen Elementen definiert.

Das Ähnlichkeitsmaß sei für die vier Mengen durch folgende Matrix vorgegeben:

| | {rot} | {gelb} | {rot, gelb} | {grün} |
|---|---|---|---|---|
| {rot} | 1 | 0 | 0 | 0 |
| {gelb} | 0 | 1 | 0.5 | 0.2 |
| {rot, gelb} | 0 | 0.5 | 1 | 0.8 |
| {grün} | 0 | 0.2 | 0.8 | 1 |

Diese Matrix kann durch

```
| entry1 entry2 entry3 anArray |
\Spezifikation der Ähnlichkeitswerte.\
entry1 := Array new.
entry1 addLast: #[#gelb]; addLast: #[#rot gelb]; addLast: 0.5.
entry2 := Array new.
entry2 addLast: #[#gelb]; addLast: #[#grün]; addLast: 0.2.
entry3 := Array new.
entry3 addLast: #[#rot #gelb]; addLast: #[#grün]; addLast: 0.8.

\Aufbau einer SimilarityList, die die Matrix repräsentiert.\
anArray := Array new.
anArray addLast: entry1; addLast: entry2; addLast: entry3.
^ SimilarityList withAll: anArray.
```

als `SimilarityList`–Instanz spezifiziert werden (beachte, daß die Werte 1 und 0 hierbei nicht explizit angegeben werden müssen).

Angenommen, der Parameter `theRange` enthalte die oben erzeugte Liste mit den vier möglichen Symbolmengen und der Parameter `theSimilarityList` die Ähnlichkeitsmatrix. Dann kann der benutzerdefinierte Typ 'Ampel' mit

```
(ApplicationType new: #Ampel
                 supertype: #NilType
                 similarity: theSimilarityList
                 range: theRange)
addToListOfTypes.
```

als eine `ApplicationType`–Instanz generiert und in der globalen Typenliste unter dem Schlüssel `#Ampel` abgespeichert werden. Die Nachricht

```
ApplicationType named: #Ampel.
```

liefert als Ergebnis dann genau diese Typinstanz aus der globalen Liste.

# Anhang B

# Hinweise zur Benutzung des Tools

## B.1 Laden der OPAL–Codefiles

Um die im Rahmen dieser Diplomarbeit implementierten OPAL–Klassen mit den zugehörigen Methoden in eine GemStone–Datenbank einfilen zu können, wurde der für die Klassen– und Methodendefinitionen erforderliche OPAL–Code in Files abgelegt. Wegen der Abhängigkeiten, die zwischen den Klassen bzw. ihren Methoden bestehen, mußte der Implementierungscode auf mehrere Files verteilt werden. Hierbei gibt es zwei Arten von Files:

- Files, die die *Klassen*definitionen und –methoden enthalten
- Files, die ausschließlich die *Instanzen*methoden enthalten

Um einen korrekten Ablauf des File–in–Vorganges zu gewährleisten, muß beim Einfilen unbedingt die im folgenden aufgeführte Reihenfolge beachtet werden. Dabei sind im ersten Schritt die Klassendefinitionen und –methoden aus den Files

```
/GemStone_Fileout/*_Classes.opl
```

einzufilen, wobei die Reihenfolge der Files relevant ist:

1. /GemStone_Fileout/BasicApplicationObjects_Classes.opl
2. /GemStone_Fileout/BasicTypeObjects_Classes.op
3. /GemStone_Fileout/TypeObject_Classes.opl
4. /GemStone_Fileout/DatabaseObjects_Classes.opl
5. /GemStone_Fileout/DatabaseSets_Classes.opl
6. /GemStone_Fileout/DatabaseKernel_Classes.opl
7. /GemStone_Fileout/ApplicationObjects_Classes.opl
8. /GemStone_Fileout/KDTreeNodes_Classes.opl
9. /GemStone_Fileout/BasicSearchingObjects_Classes.opl
10. /GemStone_Fileout/NNSearchingObjects_Classes.opl
11. /GemStone_Fileout/SeedStructure_Classes.opl

Danach können dann die Files, die die Instanzenmethoden enthalten, in *beliebiger* Reihenfolge geladen werden:

- /GemStone_Fileout/ApplicationObjects_Methods.opl
- /GemStone_Fileout/BasicApplicationObjects_Methods.opl
- /GemStone_Fileout/BasicSearchingObjects_Methods.opl
- /GemStone_Fileout/BasicTypeObjects_Methods.opl
- /GemStone_Fileout/DatabaseKernel_Methods.opl
- /GemStone_Fileout/DatabaseObjects_Methods.opl
- /GemStone_Fileout/DatabaseSets_Methods.opl
- /GemStone_Fileout/KDTreeNodes_Methods.opl
- /GemStone_Fileout/NNSearchingObjects_Methods.opl
- /GemStone_Fileout/SeedStructure_Methods.opl
- /GemStone_Fileout/TypeObject_Methods.opl

## B.2 Benutzung der GSI–Schnittstelle zu Smalltalk-80

Soll das in GemStone implementierte Tool über die *GSI*–Schnittstelle an eine Smalltalk-80–Anwendung angebunden werden, so müssen im entsprechenden Smalltalk–Image zuvor folgende Methoden abgeändert werden:

**Klassenkategorie:** *'Collections–Sequenceable'*
**Klasse:** *'OrderedCollection'*
**Instanzenmethode:** (Kategorie: 'gemstone functions')

```
toGemStoneWithEntry: reportEntry cycleDictionary: cycleDictionary
    | instSize value valueStream |
    reportEntry objImpl: 0.
    instSize := self class instSize.
"!!! 1. Änderung !!!"
    value := Array new: instSize + self size "-2".
    reportEntry value: value.
"!!! 2. Änderung !!!"
    reportEntry namedSize: instSize "-2".
    valueStream := WriteStream on: value.
"!!! 3. Änderung !!!"
    "3" 1 to: instSize
    do: [:i | valueStream nextPut:
    (GemStoneObject
    newFrom: (self instVarAt: i) cycleDictionary: cycleDictionary) ].
    self do: [:each | valueStream nextPut:
    (GemStoneObject newFrom: each cycleDictionary: cycleDictionary) ].
    ^reportEntry
```

**Klassenkategorie:** *'GemStone–Objects'*
**Klasse:** *'GSSession'*
**Klassenmethode:** (Kategorie 'default constants')

*defaultSmalltalkToGemStone*

1. Auskommentieren der Zeile
```
stToGs at: OrderedCollection
      put: (SpecialGemStoneObjects at: #OArray);
```
2. Einfügen der neuen Zeile
```
stToGs at: Array
      put: (SpecialGemStoneObjects at: #OArray);
```

Diese Änderungen sind beide notwendig, um beim Transfer von Objekten aus Smalltalk nach GemStone die Abbildung von Smalltalk–'Arrays' auf GemStone–'Arrays' und von Smalltalk–'OrderedCollections' auf GemStone–'OrderedCollections' (die dort im Rahmen dieser Arbeit neu implementiert worden sind) zu gewährleisten.

# Literaturverzeichnis


[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading Mass. (1974).

[AKM+89] K.-D. Althoff, S. Kockskämper, F. Maurer, M. Stadler und S. Wess, Ein System zur fallbasierten Wissensverarbeitung in technischen Diagnosesituationen, in ÖGAI (Hrsg.), *Proc. 5. Österreichische Artificial–Intelligence Tagung*, Springer (1989).

[AMTW91] K.-D. Althoff, F. Maurer, Ralph Traphöner und S. Wess, Die Lernkomponente der MOLTKE–Werkbank zur Diagnose technischer Systeme, *KI – Künstliche Intelligenz*, 91(1) (1991).

[AW91] K.-D. Althoff and S. Wess, Case–Based Knowledge Acquisition, Learning and Problem Solving for Diagnostic Real World Tasks, SEKI–Report SR-91-07 (SFB), Department of Computer Science, University of Kaiserslautern, Germany (1991).

[AW92] K.-D. Althoff und S. Wess, Case–Based Reasoning and Expert System Development, in: F. Schmalhofer, G. Strube, and T. Wetter (eds), *Contemporary Knowledge Engineering and Cognition*, Springer–Verlag (1992).

[AWB+92] K.-D. Althoff, S. Wess, B. Bartsch–Spörl, D. Janetzko und A. Voss, Fallbasiertes Schließen in Expertensystemen: Welche Rolle spielen Fälle für wissensbasierte Systeme?, *KI – Künstliche Intelligenz*, 92(4) (1992).

[Ben75] J.L. Bentley, Multidimensional binary search trees used for associative searching, *Commun ACM* 18, pp. 509-517 (1975).

[BM86] M.L. Brodie and J. Mylopoulos (eds), *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies*, Springer–Verlag, New York (1986).

[BM88] R. Barletta and W. Mark, Explanation–Based Indexing of Cases, *Proceedings of the DARPA Workshop on Case–Based Reasoning*, Clearwater, FL (1988).

[Bro90] A.J. Broder, Strategies for efficient incremental nearest neighbor search, *Pattern Recognition*, Vol. 23, No. 1/2, pp. 171-178 (1990).

[Das91] B.V. Dasarathy, *Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques*, IEEE Computer Society Press, Los Alamitos, CA (1991).

[FBF77] J.H. Friedman, J.L. Bentley, and R.A. Finkel, An algorithm for finding best matches in logarithmic expected time, *ACM Trans. math. Software* 3, pp. 209-226 (1977).

[Fei63] E.A. Feigenbaum, The Simulation of Verbal Learning Behavior, in: Feigenbaum and Feldman (eds), *Computers and Thought*, McGraw Hill, pp. 297-309 (1963).

[Fis87] D.H. Fisher, *Knowledge acquisition via incremental conceptual clustering*, Doctoral dissertation, Department of Information and Computer Science, University of California, Irvine (1987).

[GF91] D. Gentner and K.D. Forbus, MAC/FAC: A Model of Similarity-based Retrieval, in: *Proceedings of the 13th Annual Conference of the Cognitive Science Society*, pp. 504-509, Cognitive Science Society, Chicago (1991).

[HR76] L. Hyafil and R.L. Rivest, Constructing optimal binary decision trees is NP–complete, *Information Processing Letters 5*, pp. 15-17 (1976).

[HS81] E. Horowitz and S. Sahni, *Algorithmen: Entwurf und Analyse*, Springer Verlag (1981).

[Jac88] K.J. Jacquemain, *Effiziente Datenstrukturen und Algorithmen für mehrdimensionale Suchprobleme*, Hochschultexte Informatik (Bd. 5), Hüthig Verlag, Heidelberg (1988).

[Jan92] D. Janetzko, The Assessment of Usability of Previous Cases in Case–Based Reasoning, Unpublished paper, University of Freiburg, Germany (1992).

[Kie92] A. Kiefer, *Ein objektorientiertes featurebasiertes Entwurfssystem für rotationssymmetrische Drehteile*, Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern, Deutschland (1992).

[Koo87] L.H. Koopmans, *Introduction to Contemporary Statistical Methods*, Second Edition, Duxbury Press, Boston (1987).

[Leb86] M. Lebowitz, UNIMEM, a general learning system: An overview, in: *Proceedings of ECAI-86*, Brighton, England (1986).

[Leb87] M. Lebowitz, Experiments with Incremental Concept Formation: UNIMEM, *Machine Learning*, Vol. 2 No. 2 pp. 103-138 (1987).

[Lev90] R.A. Levinson, Pattern Associativity and the Retrieval of Semantic Networks, Technical Report 90-30, Baskin Center of Computer Science, University of California, Santa Cruz (1990).

[LS87] P.C. Lockemann und J.W. Schmidt, *Datenbank–Handbuch*, Springer–Verlag (1987).

[Meh77] K. Mehlhorn, *Effiziente Algorithmen*, Teubner, Stuttgart (1977).

[Meh84a] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer, Berlin Heidelberg (1984).

[Meh84b] K. Mehlhorn, *Data Structures and Algorithms 3: Multi–dimensional Searching and Computational Geometry*, Springer, Berlin Heidelberg (1984).

[NHS84] J. Nievergelt, H. Hinterberger, and K.C. Sevcik, The grid file: an adaptable, symmetric multikey file structure, *ACM Trans. Database System*, 9:1, pp. 38-71 (1984).

[ÖW92] H. Öchsner und S. Wess, Ähnlichkeitsbasiertes Retrieval von Fällen durch assoziative Suche in einem mehrdimensionalen Datenraum, in: K.-D. Althoff, S. Wess, B. Bartsch–Spörl und D. Janetzko (Hrsg.), Proc. *Ähnlichkeit von Fällen beim fallbasierten Schließen*, SEKI–Report, Universität Kaiserslautern, Deutschland (1992).

[PPW92] J. Paulokat, R. Präger und S. Wess, CAbPlan – fallbasierte Arbeitsplanung, in: T. Messner und A. Winklhofer (Hrsg.), *Beiträge zum 6. Workshop Planen und Konfigurieren*, Nr. 166 in FR-1992-001, S. 169, Deutschland (1992).

[Ric92] M.M. Richter, Classification and learning of similarity measures, in: *Proc. der 16. Jahrestagung der Gesellschaft für Klassifikation e.V.*, Springer Verlag (1992).

[RKW89] E.L. Rissland, J.L. Kolodner, and D. Waltz, Case–based reasoning from DARPA: Machine learning program plan, in: Hammond (ed), *Proceedings: Case–Based Reasoning Workshop*, San Mateo, California, pp. 1-13 (1989).

[RW91] M.M. Richter and S. Wess, Similarity, Uncertainty and Case-Based Reasoning in PATDEX, in: R.S. Boyer (ed), *Automated Reasoning*, Kluwer Academic Publisher, pp. 249-265 (1991).

[Serv86] Servio Logic, *Programming in OPAL*, Servio Logic Development Corp., Beaverton, Oregon (1986).

[Tve77] A. Tversky, Features of Similarity, *Psychological Review*, 84, pp. 327-352 (1977).

[Ull88] J.D. Ullman, *Principles of Database and Knowledge Base Systems*, Volume I, Computer Science Press, Rockville, Md. (1988).

[Wes91] S. Wess, PATDEX/2: Ein System zum adaptiven, fallfokusierenden Lernen in technischen Diagnosesituationen, SEKI-Working Paper SWP-91/01, Fachbereich Informatik, Universität Kaiserslautern, Deutschland (1991).

[WPA92] S. Wess, J. Paulokat und K.-D. Althoff, Fallbasiertes Schließen – ein Überblick, Technical report, Fachbereich Informatik, Universität Kaiserslautern, Deutschland (1992).