Fakultät für Informatik der Universität Karlsruhe
Institut für Logik, Komplexität und Deduktionssysteme

Studienarbeit

Schnelle Vektorquantisierung durch Bucket Voronoi Intersection Suche

Jürgen Fritsch

Karlsruhe, den 19.01.1995

Inhaltsverzeichnis

Abbildungsverzeichnis

Zusammenfassung

In dieser Studienarbeit wird ein Algorithmus zur schnellen Vektorquantisierung vorgestellt, der zur Reduktion der Komplexität eine spezielle Datenstruktur verwendet, einen sogenannten K-dimensionalen (K-d) Baum. Besondere Bedeutung gewinnt dabei die Vorverarbeitungsphase, deren Ziel die Berechnung eines möglichst optimalen K-d Baumes ist, der dann zur effizienten Nächster-Nachbar Suche vom Quantisierer benutzt wird. Um optimale K-d Bäume zu erzeugen, werden Schätzungen der Voronoi-Gebiete der Codebuchvektoren mittels einer sehr großen Menge an Trainingsvektoren bestimmt. Der n-dimensionale Raum der Codebuchvektoren wird dabei durch eine sogenannte Bucket Voronoi Intersection (BVI) möglichst optimal partitioniert, um den verbleibenden Suchaufwand zu minimieren. Es werden dabei Optimierungskriterien für den Fall einer bekannten Verteilung der Testdaten (bei der Vektorquantisierung in Form von Trainingsvektoren), als auch für den Fall, daß die Testdatenverteilung unbekannt ist und nur die Voronoi-Informationen herangezogen werden können, angegeben. Im Kontext der Vektorquantisierung von Fourier-transformierten Sprachdaten wird eine Implementierung des BVI-Algorithmus vorgestellt und analysiert, wobei besonders die Anwendbarkeit des Algorithmus in Spracherkennern untersucht wurde. Dabei konnte durch Integration der BVI-Suche in das Spracherkennungssystem JANUS-2 eine Beschleunigung der Berechnung von HMM-Emissionswahrscheinlichkeiten um den Faktor 2-5 ohne signifikante Verschlechterung der Erkennungsleistung erzielt werden.

1 Einleitung

Ein in vielen Gebieten der Informatik auftretendes Problem ist die Nächster-Nachbar Suche: Bestimme unter N vorgegebenen Punkten des K-dimensionalen Raumes den am nächsten zu einem Anfragepunkt Liegenden. Dieses Problem taucht zum Beispiel in der Mustererkennung auf, wo man einen unbekannten Merkmalsvektor durch Bestimmen des am nächsten gelegenen Referenzmusters klassifiziert. Am häufigsten tritt die Nächster-Nachbar Suche wohl in Verbindung mit der Vektorquantisierung auf, die beispielsweise zur Kompression von Sprach- oder Bilddaten eingesetzt wird.
Die Vektorquantisierung ist ein sehr mächtiges und populäres, allerdings verlustbehaftetes Datenkompressionsverfahren. Der Informationstheoretiker Shannon zeigte, daß man durch Codieren eines Vektors stets bessere Ergebnisse als durch einfaches Kodieren von Skalaren erzielen kann, selbst wenn die Quelle zufällige Daten produziert.
Bei der digitalen Übertragung von Sprache taucht beispielsweise das Problem auf, daß ein mittels PCM digitalisiertes und skalar quantisiertes Sprachsignal eine zu hohe Bandbreite besitzt, um es noch über herkömmliche Telefonleitungen übertragen zu können. Die Digitalisierung verursacht also einen höheren Bandbreitebedarf, den man nur durch geeignete Kompressionsverfahren reduzieren kann.
Ein anderes Beispiel ist die Spracherkennung. Hier extrahiert man aus dem digitalisierten Sprachsignal in einer Vorverarbeitungsphase durch Fourieranalyse, Filterbänke oder verwandte Techniken ein Frequenzspektrum, das diskretisiert eine Folge von Vektoren *(frames)* ergibt, die die Merkmale des Signals jeweils für kurze Zeit repräsentieren. Man arbeitet meist mit statistischen Modellen (HMMs), um die Wahrscheinlichkeit der Zugehörigkeit eines Merkmalsvektors zu einer bestimmten Klasse zu bestimmen. Da man dies aus Effizienzgründen meist nicht für alle Klassen tun will, berechnet man durch eine Vektorquantisierung zuerst einmal den Nächsten-Nachbarn unter allen Klassenprototypen.
In der Praxis hat die Vektorquantisierung jedoch auch ihre Grenzen. Erstens steigt der Aufwand der Quantisierung, die im wesentlichen eine Nächster- Nachbar Suche ist, exponentiell mit der Dimension K an. Zweitens ist die Vektorrate, also die Zahl der zu quantisierenden Vektoren pro Zeiteinheit, bei Realzeitanwendungen so hoch, daß man spezialisierte Signalprozessoren zur Vektorquantisierung einsetzen muß. Man weiß auch, daß die Nächster-Nachbar Suche während des Trainings eines Spracherkenners, teilweise mehr als die Hälfte der gesamten Laufzeit benötigt. Man hat daher ein starkes Interesse an der Reduktion der Komplexität der Nächster-Nachbar Suche.
Ein wichtiger Ansatz zur schnellen Nächster-Nachbar Suche (fast nearest neighbor search) ist der Einsatz von speziellen Datenstrukturen zur strukturierten Suche, sogenannten K-dimensionalen (K-d) Bäumen, wie sie von Bentley [5] eingeführt wurden. Es hat sich gezeigt, daß man geeignete Optimierungsstrategien für den Aufbau solcher multidimensionaler Binärbäume entwickeln muß, um eine signifi-

kante Reduktion der Komplexität der Nächster-Nachbar Suche zu erzielen.
Die meisten bisher entwickelten Optimierungsstrategien, wie zum Beispiel der FBF-Algorithmus [6], erzeugen K-d Bäume, die zwar eine beschleunigte Suche erlauben, meistens aber den Nachteil eines großen Aufwandes im schlechtesten Fall haben.
In dieser Studienarbeit wird nun ein Algorithmus zur Optimierung von K-d Bäumen vorgestellt, der diesen Nachteil nicht aufweist und sogar in der Lage ist, optimale Suchbäume zu generieren. Das sogenannte Bucket Voronoi Intersection (BVI) Verfahren erzeugt diese optimalen Suchbäume unter Verwendung von Näherungen an die Voronoi-Gebiete der Codebuchvektoren. Zur Berechnung dieser Näherungen benötigt der Algorithmus eine große Zahl von Trainingsvektoren, um den im Gegensatz zu anderen Verfahren hier unvermeidbaren Quantisierungsfehler so klein zu halten, daß er in der praktischen Anwendung tolerierbar ist.
Außerdem werden auch Informationen über die Verteilung der zu quantisierenden Vektoren mit in die Optimierung einbezogen, falls man diese zur Verfügung hat. Es existieren dazu zwei Optimierungskriterien, das EOC (exact optimization criterion) für den Fall bekannter Testdatenverteilung, und das GOC (generalized optimization criterion) für den Fall daß kein a-priori Wissen über die zu quantisierenden Vektoren zur Verfügung steht. Beide Kriterien werden detailliert vorgestellt und analysiert.
Es wird weiterhin an einem Sprachererkennungssystem untersucht, inwiefern sich die sehr häufig durchzuführende Nächster-Nachbar Suche durch BVI-optimierte K-d Bäume beschleunigen läßt.

2 Vektorquantisierung

Die Vektorquantisierung ist eine Verallgemeinerung der skalaren Quantisierung auf K-dimensionale Vektoren. Während die skalare Quantisierung meist nur bei der Analog/Digital-Wandlung Verwendung findet, ist das Hauptanwendungsgebiet der Vektorquantisierung die Datenkompression, beispielsweise bei der Kodierung und Kompression von Sprache und Bildern, oder auch bei der Spracherkennung.
Die Vektorquantisierung kann als eine Art von Mustererkennungsverfahren angesehen werden. Dabei wird ein Eingabemuster auf das "am nächsten liegende" Muster einer Referenzmenge abgebildet. Andererseits wird die Vektorquantisierung auch als ein Vorverarbeitungsschritt für komplexere Signalverarbeitungsalgorithmen benutzt, wie zum Beispiel Klassifikation oder linare Transformation.

2.1 Definitionen

Ein Vektorquantisierer Q der Dimension K und der Größe N ist eine Abbildung eines Vektors (Punktes) des K-dimensionalen Euklidischen Raumes, $\mathcal{R}^K$, in eine

endliche Menge C, bestehend aus N Ausgabevektoren. Die Menge C nennt man das *Codebuch*, die darin enthaltenen Vektoren die *Codebuch-Vektoren.*

$$Q : \mathcal{R}^K \rightarrow C \quad \text{wobei} \quad C = (\mathbf{c_1}, \ldots, \mathbf{c_N}) \quad \mathbf{c_i} \in \mathcal{R}^K \quad \forall i \in \{1, \ldots, N\}$$

Ein Vektor $\mathbf{x} = (x_1, \ldots, x_K)$ wird auf denjenigen Vektor aus dem Codebuch C abgebildet, zudem er den geringsten Abstand bezüglich einem vorgegebenen Abstandsmaß $d(\mathbf{x}, \mathbf{y})$ hat. Dies ist gleichbedeutend mit einer Nächster-Nachbar Suche des Vektors $\mathbf{x}$ im Codebuch C. Somit gilt :

$$Q(\mathbf{x}) = \mathbf{c_i} \in C : d(\mathbf{x}, \mathbf{c_i}) \leq d(\mathbf{x}, \mathbf{c_j}) \quad \forall j \in \{1, \ldots, N\}$$

Die Auflösung oder Code Rate (code rate) eines Vektorquantisierers ist definiert als $r = (\log_2 N)/K$. Sie gibt die Anzahl der Bits pro Vektorkomponente an, die zum Kodieren eines Vektors verwendet werden, und gibt somit die Genauigkeit oder Qualität an, die mit dem Vektorquantisierer erreicht werden kann, vorausgesetzt das Codebuch C wurde optimal gewählt.
Wichtig ist, daß die Code Rate bei vorgegebener Dimension K nur von der Größe N des Codebuchs C abhängt und nicht etwa von der Anzahl der Bits, die benötigt werden, um die Vektoren im Codebuch numerisch darzustellen.
Das Codebuch wird typischerweise als eine Tabelle im Speicher realisiert, die Paare $(i, \mathbf{c_i})$ von Indizes und Codebuchvektoren enthält. Dadurch ist das Ergebnis der Quantisierung eines Vektors $\mathbf{x}$ eine natürliche Zahl, die einen Index in die Tabelle darstellt :

$$Q : \mathcal{R}^K \rightarrow \mathcal{I} \quad \text{mit} \quad \mathcal{I} = \{1, \ldots, N\}$$

Bei der Dekodierung wird dann eine natürliche Zahl, die einen gültigen Index darstellen muß, durch einfaches table-look-up auf einen Vektor $\mathbf{c_i} \in C$ abgebildet. Somit präsentiert sich ein Vektorquantisierer (VQ) wie folgt :

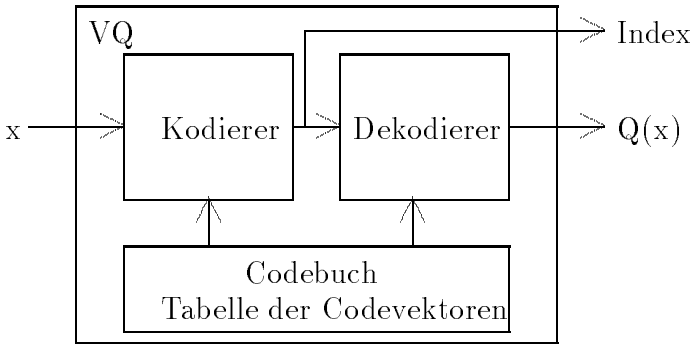

Abbildung 1: Struktur eines Vektorquantisierers

2.2 Komplexität der Vektorquantisierung

Der einfachste Algorithmus zum Quantisieren eines Testvektors $\mathbf{x}$ der Dimension K mit einem Codebuch $C = \{\mathbf{c_i} : i = 1, \ldots, N\}$ gemäß der Nächster-Nachbar Regel verläuft wie folgt:

Schritt 1 Setze $d = d_0, j = 1$ und $i = 1$.

Schritt 2 Berechne $D_j = d(\mathbf{x}, \mathbf{c_j})$.

Schritt 3 Falls $D_j < d$, setze $d = D_j$ und $i = j$.

Schritt 4 Falls $j < N$, setze $j = j + 1$ und gehe zu Schritt 2.

Schritt 5 Stop. Ergebnis ist der Index i.

Diesen Algorithmus kann man als eine vollständige, erschöpfende Suche unter allen N Vektoren des Codebuchs C ansehen. Dieser Algorithmus benötigt N Vektor zu Vektor Distanzberechnungen, die wiederum jeweils K Multiplikationen und $K-1$ Additionen benötigen, wenn man die euklidische Norm als Abstandsmaß benutzt. Das ergibt einen Aufwand von $O(NK)$, der exponentiell mit der Dimension K und der Bitrate r steigt ($N = 2^{Kr}$).

2.3 Voronoi Gebiete

Durch die Nächster-Nachbar Suche in einem Codebuch C wird implizit eine Partitionierung des K-dimensionalen Raumes in die Voronoi-Gebiete der Codebuchvektoren vorgenommen, die für schnelle Suchalgorithmen von essentieller Bedeutung ist. Durch eine explizite Verwendung der Information, die in der Voronoi-Partitionierung des K-dimensionalen Raumes liegt, kann man dem Codebuch eine Struktur aufprägen, die eine schnelle Suche erst möglich macht.

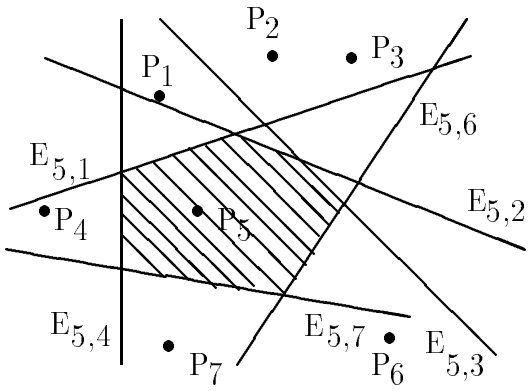

Abbildung 2: Voronoi-Gebiet eines Punktes im $\mathcal{R}^2$

Gegeben sei ein Codebuch $C = \{\mathbf{c_i} : \mathbf{c_i} \in \mathcal{R}^K\}, i = 1, \ldots, N$ und ein Abstandsmaß $d(\mathbf{x}, \mathbf{y})$. Durch das Abstandsmaß wird der K-dimensionale Raum in N

nicht-überlappende Regionen partitioniert, die man die Voronoi-Gebiete der Codebuchvektoren $\mathbf{c_i}$ nennt. Das Voronoi-Gebiet V_i zu einem Vektor $\mathbf{c_i}$ ist definiert als:

$$V_i = \{\mathbf{x} \in \mathcal{R}^K : d(\mathbf{x}, \mathbf{c_i}) \leq d(\mathbf{x}, \mathbf{c_j}) \quad \forall j \in \{1, \ldots, N\}$$

Sei $H(\mathbf{c_i}, \mathbf{c_j})$ mit $i, j = 1, \ldots, N, i \neq j$ der Halbraum definiert durch

$$H(\mathbf{c_i}, \mathbf{c_j}) = \{\mathbf{x} : d(\mathbf{x}, \mathbf{c_i}) < d(\mathbf{x}, \mathbf{c_j})\}$$

Dies ist der Raum aller Punkte, die näher am Vektor $\mathbf{c_i}$ als am Vektor $\mathbf{c_j}$ liegen. Mit dem Euklidischen Abstandsmaß $d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{l=1}^{K}(x_l - y_l)^2}$ definiert $E = \{\mathbf{x} : d(\mathbf{x}, \mathbf{c_i}) = d(\mathbf{x}, \mathbf{c_j})\}$ eine Hyperebene, die den K-dimensionalen Raum $\mathcal{R}^K$ in zwei Halbräume $H(\mathbf{c_i}, \mathbf{c_j})$ und $H(\mathbf{c_j}, \mathbf{c_i})$ trennt. Man kann nun die Voronoi-Gebiete als Schnitt solcher Halbräume definieren:

$$V_i = \cap_{i \neq j} H(\mathbf{c_i}, \mathbf{c_j})$$

Als Schnitt konvexer Gebiete sind die Voronoi-Gebiete ebenfalls konvex. Liegt ein Testvektor $\mathbf{x}$ im Voronoi-Gebiet V_i, so ist der dazugehörige Codebuchvektor $\mathbf{c_i}$ der Nächste- Nachbar von $\mathbf{x}$ im Codebuch C. Zur Bestimmung des Nächsten Nachbarn von $\mathbf{x}$ genügt es also, das Voronoi-Gebiet, in dem $\mathbf{x}$ liegt, zu finden.

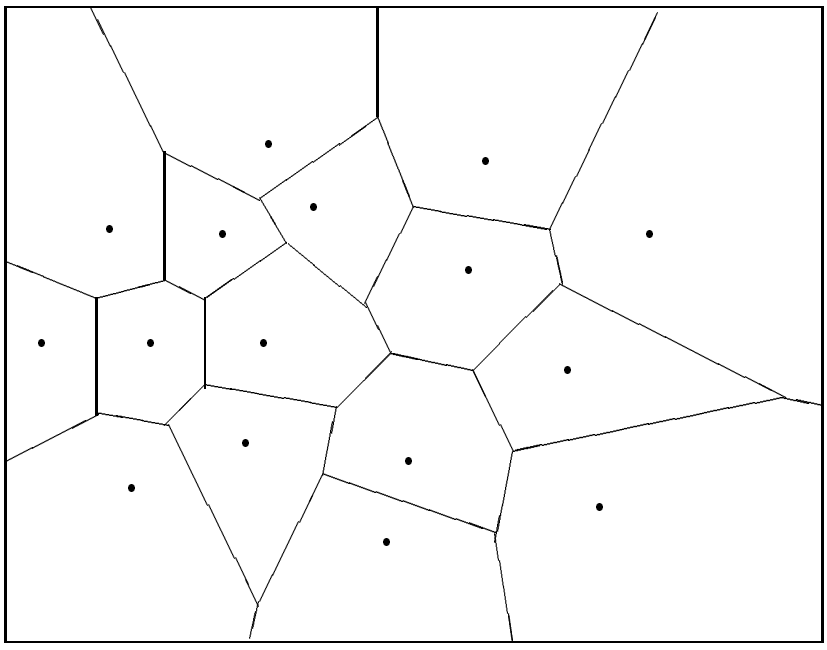

Abbildung 3: Voronoi-Partitionierung im $\mathcal{R}^2$

Die direkte Bestimmung des Voronoi-Gebietes, in dem ein Testvektor $\mathbf{x}$ liegt, ist sehr komplex und bietet im Vergleich mit der Distanz-basierten, vollständigen Suche keine Ersparnis im Berechnungsaufwand, insbesondere wenn man bedenkt,

daß das Ganze sich in Räumen der Dimension $K \geq 10$ abspielt.
Beim Entwurf eines schnellen Algorithmus zur Nächster-Nachbar Berechnung ist man also darauf angewiesen, weniger komplexe Lösungen zur Bestimmung des Voronoi-Gebietes in dem sich ein Testvektor befindet zu finden.

2.4 Algorithmen zur Bestimmung von Codebüchern

Bevor man die eigentliche Vektorquantisierung durchführen kann, muß ein möglichst optimales Codebuch generiert werden. Optimal heißt in diesem Fall, daß der Quantisierungsfehler nach Kodieren und anschließendem Dekodieren von Testvektoren mit diesem Codebuch minimal wird.
Algorithmen zur Bestimmung von Codebüchern sind nicht Thema dieser Studienarbeit, vielmehr werden die fertigen Codebücher vorausgesetzt. Trotzdem soll hier ein kurzer Überblick über einige Verfahren zu deren Berechnung gegeben werden.

2.4.1 Monte Carlo Methode

Die einfachste Methode zur Bestimmung eines Codebuchs besteht darin, die N Vektoren zufällig gemäß der Verteilung der zu quantisierenden Vektoren zu wählen. Die Verteilung wird mit einer sehr großen Trainingsvektorenmenge bestimmt. Man kann aber auch einfach die ersten N Vektoren aus der Trainingsmenge als Codebuchvektoren benutzen. Meistens sind die Vektoren aber stark korreliert und man erzielt bessere Ergebnisse, wenn man nur jeden k-ten Trainingsvektor auswählt.

2.4.2 Pruning

Bei dieser Methode beginnt man mit einem Codebuch, daß alle Trainingsvektoren enthält und sortiert nach und nach ungeeignete Vektoren aus, bis man nur noch die gewünschte Anzahl an Vektoren im Codebuch hat.
In der Praxis verläuft dieses Verfahren wie folgt: Setze den ersten Trainingsvektor in das anfangs leere Codebuch ein. Berechne die Distanz zwischen dem nächsten Trainingsvektor und dem ersten Codebuchvektor. Falls diese Distanz größer als ein vorgegebener Schwellwert ist, füge den Trainingsvektor als zweiten Codebuchvektor in das Codebuch ein, u. s. w. Das Verfahren endet, wenn das Codebuch genug Vektoren enthält. Falls alle Trainingsvektoren benutzt wurden, das Codebuch aber weniger als die gewünschte Anzahl an Vektoren enthält, muß man das Verfahren mit einem kleineren Schwellwert wiederholen.

2.4.3 Paarweise Nächste-Nachbarn

Eine kompliziertere, aber bessere Methode zur Bestimmung eines Codebuchs ist der Paarweise Nächste-Nachbarn (PNN) Clustering Algorithmus, vorgeschlagen

von Equitz [9]. Dieses Verfahren ist vergleichbar mit dem Pruning, da auch hier das Codebuch aus einer anfänglich großen Anzahl von Trainingsvektoren gewonnen wird. Der Algorithmus benötigt mehr Rechenzeit als die Vorangehenden, ist jedoch immer noch schneller als der im nächsten Abschnitt vorgestellte LBG-Algorithmus.
Zu Anfang stellt jeder der Trainingsvektoren einen eigenen Cluster bestehend aus einem einzigen Vektor dar. Ziel ist es nun, Cluster von Vektoren zusammenzufassen, bis man bei der gewünschten Anzahl N angelangt. In jedem Cluster wird der Mittelwert über alle darin befindlichen Vektoren gebildet und dieser dann als Repräsentant in das Codebuch eingetragen.
Der Algorithmus verläuft wie folgt: Zuerst wird die Distanz zwischen allen Paaren von Trainingsvektoren berechnet. Die zwei Vektoren mit dem geringsten Abstand werden zu einem neuen Cluster zusammengefaßt und durch ihren Mittelpunkt repräsentiert. Im weiteren Verlauf des Algorithmus muß man für alle Paare von Clustern jeweils prüfen, wie stark sich die Fehlerrate durch das Zusammenfügen erhöhen würde. Seien $R_i = \{\mathbf{x_i}(l) : l = 1, 2, \ldots, L_i\}$ und $R_j = \{\mathbf{x_j}(l) : l = 1, 2, \ldots, L_j\}$ zwei Cluster bestehend aus L_i bzw. L_j Vektoren. Der Beitrag dieser beiden Cluster zum mittleren Quantisierungsfehler, wenn sie nicht zusammengefügt werden, beträgt

$$\Delta_{i,j} = \sum_{l=1}^{L_i} d(\mathbf{x_i}(l), \mathrm{cent}(R_i)) + \sum_{l=1}^{L_j} d(\mathbf{x_j}(l), \mathrm{cent}(R_j))$$

während der Fehler nach dem Zusammenfügen

$$\Delta'_{i,j} = \sum_{l=1}^{L_i} d(\mathbf{x_i}(l), \mathrm{cent}(R_i \bigcup R_j)) + \sum_{l=1}^{L_j} d(\mathbf{x_j}(l), \mathrm{cent}(R_i \bigcup R_j)) \geq \Delta_{i,j}$$

beträgt. Dabei ist $d(\mathbf{x_i}, \mathbf{y_i})$ ein Abstandsmaß und $\mathrm{cent}(R_i) = 1/L_i \sum_{l=1}^{L_i} \mathbf{x_i}(l)$. Das Paar von Clustern R_i, R_j für welches die Differenz $\Delta'_{i,j} - \Delta_{i,j}$ minimal wird, wird zusammengefügt. Es wird also jeweils genau das Paar zusammengefügt, das den kleinsten Zuwachs des Quantisierungsfehlers verursacht. Man beachte, daß zwar das Zusammenfügen zweier Cluster optimal ist, dies jedoch nicht unbedingt für den gesamten Algorithmus gilt.

2.4.4 Der Linde-Buzo-Gray (LBG) Algorithmus

Der LBG-Algorithmus startet mit einem beliebig gewählten Codebuch der gewünschten Größe N und versucht dieses iterativ zu verbessern. In [11] wird dieser Algorithmus insbesondere auf seine Anwendbarkeit in der Datenkompression untersucht, während er in [2] unter dem Namen "Generalisierter Lloyd Algorithmus" vorgestellt wird.
Ein Iterationsschritt des Algorithmus verläuft wie folgt:

- Sei $C_m = \{\mathbf{c_i} \in \mathcal{R}^k : i = 1, \ldots, N\}$ das Codebuch aus dem m-ten Iterationsschritt und $\mathcal{T}$ die Menge der Trainingsvektoren. Partitioniere die Menge $\mathcal{T}$ in Clustermengen R_i gemäß der Nächster-Nachbar Bedingung:

$$R_i = \{\mathbf{x} \in \mathcal{T} : d(\mathbf{x}, \mathbf{c_i}) \leq d(\mathbf{x}, \mathbf{c_j}); \quad \forall j = 1, \ldots, N\}$$

- Berechne zu jedem Cluster den Mittelpunktsvektor

$$\mathbf{m_i} = \text{cent}(R_i) = \frac{1}{L_i} \sum_{l=1}^{L_i} x_l$$

  Setze $C_{m+1} = \{\mathbf{m_i} : i = 1, \ldots, N\}$ als das neue, verbesserte Codebuch. Sollte nach dem ersten Schritt ein Cluster R_i leer geblieben sein, muß man sich eine Sonderbehandlung überlegen.

Diese Iteration wird nun solange durchgeführt, bis der mittlere Quantisierungsfehler nicht mehr signifikant sinkt. Da dieser Algorithmus in jeder Iteration auf jeden Fall eine Minderung des Quantisierungsfehlers anstrebt, besteht die Möglichkeit, daß er in einem lokalen Optimum steckenbleibt. Der Algorithmus führt also nicht immer zu einem optimalen Codebuch.
Verschiedenste Erweiterungen des LBG-Algorithmus, aber auch andere Optimierungsverfahren wurden vorgeschlagen, um aus lokalen Optima wieder herauszufinden. Eine Familie von solchen Verfahren ist die stochastische Relaxation mit dem populären Vertreter des "Simulierten Ausglühens" (simulated annealing).

3 Schnelle Nächster-Nachbar Algorithmen

Bevor nun im nächsten Kapitel der BVI-Algorithmus vorgestellt wird, sollen hier einige andere Ansätze zur schnellen Nächster-Nachbar Suche vorgestellt werden, da diese zum Teil Vorläufer des BVI-Verfahrens sind. Es wird hier auch schon die Datenstruktur K-d Baum vorgestellt, da diese außer im BVI-Algorithmus auch in anderen NN-Suchverfahren verwendet wird.

3.1 Partielle Distanzberechnung

Eine der einfachsten Methoden zur Reduktion der Komplexität der Nächster-Nachbar Suche bei einem Vektorquantisierer ist die partielle Distanzberechnung (partial distance search), die nur eine geringe Modifikation des Basisalgorithmus (vollständige Suche) verlangt. Vorraussetzung ist jedoch die Verwendung des Euklidischen Abstandsmaßes.
Nach wie vor bleibt dabei das Codebuch unstrukturiert, und bei der Nächster-Nachbar Suche für einen beliebigen Testvektor müssen nach wie vor alle Vektoren des Codebuchs als potentielle Nächste-Nachbarn überprüft werden.

Die Distanz des Testvektors $\mathbf{x}$ zu einem Codebuchvektor $\mathbf{c_i}$, beide mit der Dimension K, muß sequentiell in K Schritten berechnet werden. Dabei steigt die anfänglich auf Null gesetzte Distanz in jedem Schritt monoton. Daher kann man bei der Suche nach einer minimalen Distanz unter allen N Vektoren des Codebuchs die Berechnung der Distanzen im j-ten Iterationsschritt ($j < K$) abbrechen, falls die bis dahin berechnete Teildistanz $D_j = \sum_{l=1}^{j}(x_l - c_i l)^2$ das bisherige Minimum überschreitet. Die restlichen $(K - j)$ Iterationsschritte sind überflüssig, da die Distanz D nur noch größer als D_j sein kann.
Dieser Algorithmus findet zu gegebenem Testvektor $\mathbf{x}$ immer den korrekten Nächsten-Nachbarn im Codebuch C. Im Falle dieser Studienarbeit, bei der mit Vektoren der Dimension $K = 16$ und Codebüchern der Größe $N = 50$ gearbeitet wurde, war der Algorithmus der nur partielle Distanzen berechnet etwa um den Faktor 2 schneller als der Basisalgorithmus.
Alle weiteren Verfahren, die das Codebuch auf irgend eine Weise strukturieren, um bei einer Anfrage nur einen Teil aller Codebuchvektoren betrachten zu müssen, arbeiten in ihrer letzten Phase mit dem Algorithmus der partiellen Distanzberechnung, so auch das Bucket Voronoi Intersection Verfahren.

3.2 Projektionsmethode

Um die Komplexität des Quantisierungsvorganges zu reduzieren, versucht man die Grenzen der Voronoi-Gebiete der Codebuchvektoren in geeigneter Weise vorzuberechnen und in einer Datenstruktur zu speichern, die schnelle Nächster-Nachbar Berechnungen ermöglicht. Die Strategie ist dabei, den Großteil der Komplexität des Verfahrens in die Vorberechnungsphase zu stecken, da diese ja nur einmal ausgeführt werden muß und nicht zeitkritisch ist.
Ein Beispiel für ein solches Verfahren ist die Projektionsmethode, vorgestellt in [10]. Der Algorithmus bestimmt zu jedem Voronoi-Gebiet den kleinsten umschließenden Hyperquader im K-dimensionalen Raum. Dazu projeziert man ein Voronoi-Gebiet V_i auf die j-te Koordinatenachse und erhält dadurch Minimum und Maximum des Voronoi-Gebietes in der j-ten Koordinate. Der umschließende Hyperquader ist durch $2K$ Hyperebenen bestimmt, die alle orthogonal zu einer Koordinatenachse liegen.
Berechnet man zu allen N Voronoi-Gebieten die umschließenden Hyperquader, so wird jede Koordinatenachse in $2N$ disjunkte Intervalle aufgeteilt. In einer Tabelle speichert man nun für jedes dieser Intervalle diejenigen Voronoi-Gebiete, deren Projektion im betreffenden Intervall liegen. Für jede der K Koordinatenachsen wird eine eigene Tabelle generiert. Damit ist die Vorberechnungsphase abgeschlossen.

Um den Nächsten-Nachbarn zu einem Testvektor $\mathbf{x}$ zu finden, wird jede Komponente x_j skalar quantisiert, indem das Intervall bestimmt wird, in dem x_j liegt. Durch einfaches table-look-up kann man daraus dann die Menge der Codebuchvektoren bestimmen, die als Kandidaten für den Nächsten-Nachbarn in Frage

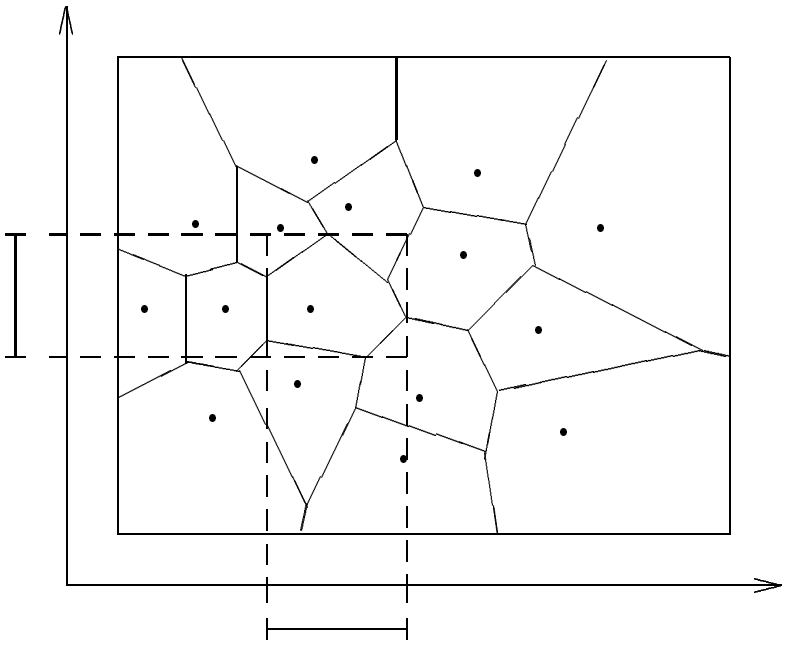

Abbildung 4: Projektion von Voronoi-Gebieten im $\mathcal{R}^2$

kommen. So erhält man für jede Koordinate j eine Kandidatenmenge M_j. Da der gesuchte Nächste-Nachbar in allen K Kandidatenmengen M_j enthalten sein muß, bildet man die Schnittmenge über die Kandidatenmengen, $M = \cap_{j=1}^{K} M_j$. Aus der Menge M der übrig gebliebenen Kandidatenvektoren wird nun der Nächste-Nachbar durch eine vollständige Suche ermittelt.
In der Menge M befinden sich meist viel weniger Vektoren als die ursprünglichen N Vektoren des Codebuchs. Die Berechnung und Auswertung der Kandidatenmengen erfordert nur Vergleiche, kann daher in der Aufwandsabschätzung gegenüber den Abstandsberechnungen bei der vollständigen Suche vernachlässigt werden.
Obwohl man theoretisch mit der Projektionsmethode stets den Nächsten-Nachbarn bestimmen kann, gilt das in der Praxis nur noch dann, wenn man die exakten Projektionen der Voronoi-Gebiete bestimmen kann. Diese Aufgabe analytisch zu lösen, ist für höherdimensionale Räume zu komplex, so daß man in der Praxis Näherungen an die exakten Projektionen durch das Kodieren einer sehr großen Trainingsvektorenmenge bestimmt. Cluster von Trainingsvektoren, die dabei zum gleichen Codebuchvektor quantisiert werden, dienen als Näherung für die echten Voronoi-Gebiete.
Die Fehlerrate des Verfahrens hängt also von der Güte der Näherungen ab und somit indirekt von der Größe der Trainingsvektorenmenge.

3.3 K-dimensionale (K-d) Bäume

Ein K-d Baum ist eine Verallgemeinerung des einfachen Binärbaumes, durch den der K-dimensionale Raum $\mathcal{R}^K$ in disjunkte Hyperquader partitioniert wird. Jeder

innere Knoten des K-d Baumes repräsentiert eine Hyperebene E, die senkrecht zu einer der K Koordinatenachsen liegt:

$$E = \{\mathbf{x} \in \mathcal{R}^K : x_j = h\}$$

Durch die Koordinatenachse j, zu der die Hyperebene senkrecht steht, und den Parameter h, der die Position auf der Achse angibt, ist die Hyperebene vollständig bestimmt, sodaß jeder innere Knoten aus den zwei Skalaren (j, h) besteht. Durch die Hyperebene E werden die zwei Halbräume

$$H_L = \{\mathbf{x} \in \mathcal{R}^K : x_j \leq h\} \quad \text{und} \quad H_R = \{\mathbf{x} \in \mathcal{R}^K : x_j > h\}$$

definiert. Es kann nun durch einen einfachen, skalaren Vergleich der Form $x_j \leq h$ bestimmt werden, auf welcher Seite der Hyperebene E ein vorgegebener Testvektor $\mathbf{x} = (x_1, \ldots, x_j, \ldots, x_K)^T$ liegt.
Das initiale Gebiet, das meist aus dem Einheitshyperwürfel besteht, korrespondiert mit dem Wurzelknoten eines K-d Baumes. Die beiden Unterräume H_L und H_R, die durch die Hyperebene in diesem Knoten erzeugt werden, entsprechen dem linken und rechten Sohn der Wurzel. Jeder dieser beiden Räume wird nun sukzessive durch Hyperebenen, die orthogonal zu den Koordinatenachsen liegen, weiter unterteilt. Nach d solchen Unterteilungen erhält man einen vollständigen, balancierten Binärbaum der Höhe d mit 2^d disjunkten Hyperquadern als Blättern, die *buckets* genannt werden.

Zu einem vorgegebenen Testvektor $\mathbf{x}$ kann mit nur d skalaren Vergleichen festgestellt werden, in welchem der 2^d buckets der Testvektor liegt. Man beginnt dazu an der Wurzel und verzweigt, je nach Ausgang des Hyperebenen-Tests $x_j \leq h$ am Knoten (j, h) nach links oder rechts, bis man schließlich in dem Blatt ankommt, in dessen bucket der Testvektor liegt.
Ein K-d Baum stellt also eine Partitionierung eines initialen Gebietes des $\mathcal{R}^K$ in 2^d disjunkte Hyperquader dar, die eine sehr schnelle Lokalisierung eines Testvektors erlaubt. Ein K-d Baum hat zudem nur geringen Speicherplatzbedarf, da man in jedem inneren Knoten des K-d Baumes nur 2 skalare Werte (j, h) speichern muß. Da jedoch die Anzahl der Knoten im K-d Baum exponentiell mit der Höhe ansteigt, kann man in der Praxis sicher nur K-d Bäume mit einer Höhe $d < 20$ verwenden.
In Abbildung 5 ist ein K-d Baum der Höhe $d = 3$ zusammen mit der durch ihn erzeugten Partitionierung für den planaren Fall dargestellt. Das initiale Gebiet ABCD wird durch die Hyperebene EF in zwei Hälften geteilt, die dem linken und rechten Sohn des Wurzelknotens entsprechen. Diese beiden Hälften ABEF und FECD werden dann wiederum durch die Hyperebenen GH und IJ geteilt, u. s. w. Mit den folgenden 3 skalaren Vergleichen wird der Testvektor $\mathbf{x} = (x_1, x_2)$ im bucket IEQR lokalisiert:

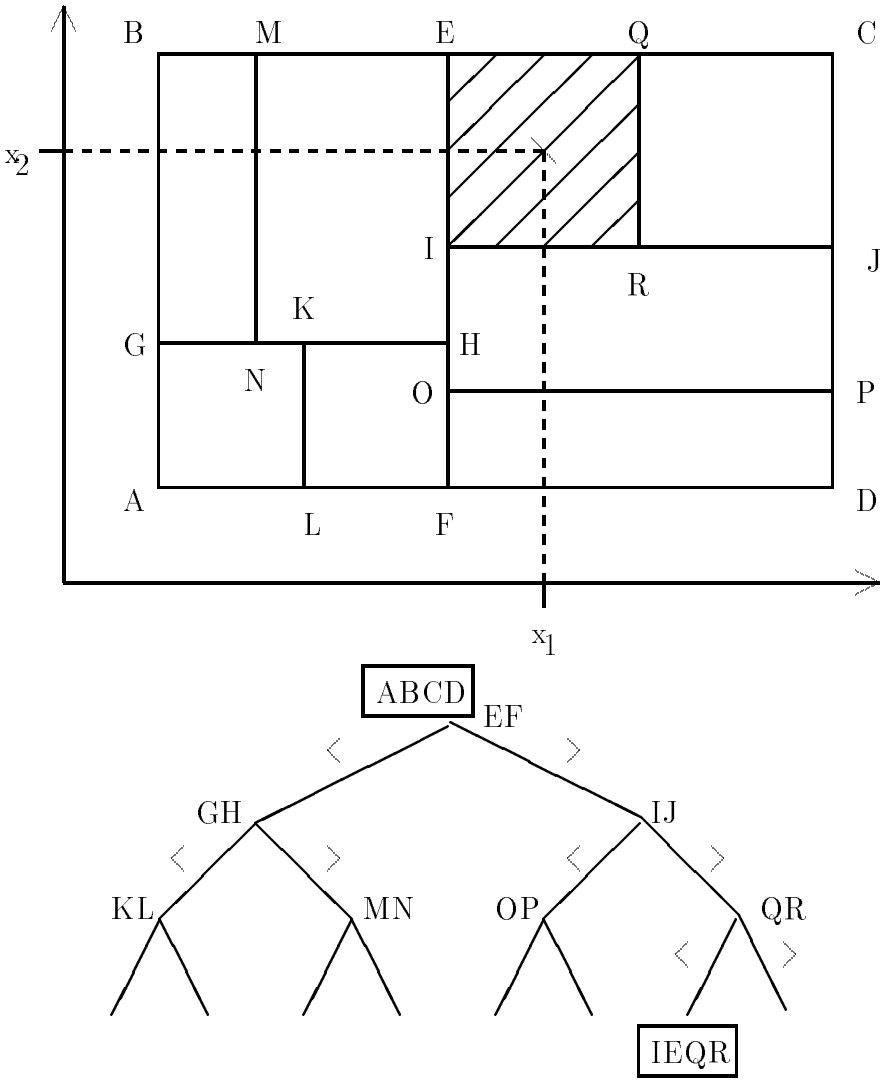

Abbildung 5: Partitionierung des $\mathcal{R}^2$ durch einen K-d Baum

1.Vergleich x_1 mit der Hyperebene EF: rechter Sohn

2.Vergleich x_2 mit der Hyperebene IJ: rechter Sohn

3.Vergleich x_1 mit der Hyperebene QR: linker Sohn

Um einen K-d Baum zur schnellen Nächster-Nachbar Quantisierung zu benutzen, erweitert man den Baum an den Blättern um eine Listenstruktur, in der all diejenigen Codebuchvektoren gespeichert werden, die noch als Kandidaten in Frage kommen, wenn ein Testvektor in dem jeweiligen Hyperquader lokalisiert wurde. Ein derart modifizierter K-d Baum kann bei der Suche in einem Codebuch sehr effizient sein, allerdings muß man sich dazu geeignete Verfahren überlegen, um einen für diesen Zweck optimalen K-d Baum aufzubauen. Optimal ist der K-d Baum dann, wenn die mittlere Anzahl der Codebuchvektoren, die sich in jedem Blatt des Baumes befinden und die noch sequentiell durchsucht werden müssen, minimal im Vergleich zu allen anderen möglichen K-d Bäumen ist.

3.4 Der Friedman-Bentley-Finkel Algorithmus

Der FBF-Algorithmus stellt einen ersten Ansatz dar, wie man optimierte K-d Bäume zu einem vorgegebenen Codebuch C generieren kann (siehe [6]). Das dem Algorithmus zugrundeliegende Verfahren optimiert den Baum lokal in jedem inneren Knoten, d. h. bei der Wahl einer Hyperebene zu einem Knoten fließt keinerlei Information aus anderen Knoten des Baumes in die Entscheidung ein. Man hat sich für ein lokales Optimierungsverfahren entschieden, weil die globale Optimierung des Baumes viel zu viele Freiheitsgrade hat und deswegen viel zu komplex wäre.
Geht man beispielsweise von einem Baum der Höhe $d = 12$ mit Vektoren der Dimension $K = 16$ aus, so hätte man für $2^{12} - 1 = 4095$ innere Knoten jeweils die Entscheidung für eine von 16 Koordinatenachsen zu treffen, inklusive einer optimal trennenden Hyperebene, was der gleichzeitigen Optimierung von über 100000 Parametern entspräche.
Der FBF-Algorithmus bestimmt deshalb lokal, für die zu einem inneren Knoten i gehörenden Hyperquader H_i, eine Trennhyperebene E_i. Er startet mit dem Wurzelknoten und berechnet die Hyperebenen der inneren Knoten in der Reihenfolge einer Breitensuche. Er berücksichtigt dazu nur diejenigen Codebuchvektoren, die innerhalb des zu einem Knoten gehörenden Hyperquaders H_i liegen, was im Falle des Wurzelknotens noch alle N Vektoren des Codebuchs sind.
Die beiden Parameter (j, h) einer Hyperebene werden dabei folgendermaßen bestimmt:

- Für alle K Koordinatenachsen bestimmt man die Verteilung der zur Achse gehörenden Komponente der Codebuchvektoren und wählt für den Parameter j diejenige Achse, entlang derer die Verteilung maximale Varianz aufweist.

- Für den Parameter h der Hyperebene wählt man dann den Median der Verteilung entlang der Achse, für die man sich entschieden hat.

Zur Bestimmung der Varianz und des Medians benutzt man nur die Vektoren des Codebuchs, die in dem Hyperquader liegen, für den die Trennhyperebene berechnet werden soll.
Als Ergebnis liefert dieser Algorithmus einen vollständigen, balancierten Binärbaum, der in jedem bucket gleich viele Codebuchvektoren enthält, mit der Vorraussetzung, daß die Größe N des Codebuchs eine 2-er Potenz ist.
Bei der Nächster-Nachbar Suche mit dem entstandenen K-d Baum der Höhe d wird zu vorgegebenem Testvektor $\mathbf{v}$ in d Schritten von der Wurzel zu einem Blatt der Hyperquader bestimmt, in dem der Testvektor liegt. Es genügt nun aber nicht, linear unter allen Codebuchvektoren des zugehörigen buckets zu suchen, da der gesuchte Nächste-Nachbar auch in einem benachbarten bucket liegen kann. Dies liegt daran, daß sich das Voronoi-Gebiet eines Codebuchvektors durchaus über mehrere buckets erstrecken kann, da die Voronoi-Information nicht in das Design des K-d Baumes eingeflossen ist.
Der FBF-Algorithmus bestimmt daher zu dem Testvektor v zuerst den Nächsten-Nachbarn c_i innerhalb des gefundenen buckets durch eine vollständige lineare Suche. Sei nun

$$B_i = \{\mathbf{x} \in \mathcal{R}^K : d(\mathbf{v}, \mathbf{x}) \leq d(\mathbf{v}, \mathbf{c_i})\}$$

Dies ist die Hyperkugel mit Mittelpunktsvektor v und der Distanz zwischen v und bisherigem Nächsten-Nachbarn c_i als Radius. Im zweiten Schritt des FBF-Algorithmus wird nun eine Backtracking-Suche gestartet, um in buckets, die mit der Hyperkugel B_i überlappen, nach Codebuchvektoren zu suchen, die eventuell noch näher an v liegen.

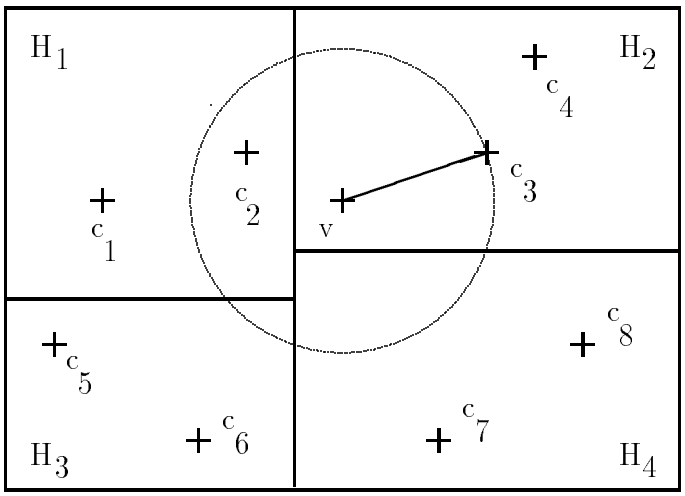

Abbildung 6: Nächster-Nachbar Suche beim FBF-Algorithmus

Genau diese Backtracking-Suche ist es, die beim FBF-Algorithmus einen hohen Aufwand verursacht, dominiert vom Hyperkugel-bucket Überlappungstest, der eine (teure) Vektordistanz-Berechnung enthält. Die Anzahl der Überlappungstests, die durchgeführt werden müssen, ist kaum vorhersagbar, jedoch hat man einen sehr hohen Aufwand im schlechtesten Fall (worst case) festgestellt.
Die exakte Analyse der Optimierungsstrategie des FBF-Algorithmus ist aufgrund der Backtracking-Suche sehr schwer. Zudem wird die Verteilung der zu quantisierenden Daten nicht berücksichtigt, was eine Analyse der Leistung dieses Algorithmus sehr ungenau macht.

4 Bucket Voronoi Intersection Suche

4.1 Voronoi-Informationen und K-d Bäume

Bei der Bucket Voronoi Intersection Suche (BVI-Suche) benutzt man genau wie beim FBF-Algorithmus einen K-d Baum, um den K-dimensionalen Raum in disjunkte buckets zu partitionieren. Da die Nächster-Nachbar Suche jedoch sehr eng mit der geometrischen Interpretation in Form von Voronoi-Gebieten der Codebuchvektoren zusammenhängt, darf man diese Information beim Design eines optimalen K-d Baumes nicht ignorieren.
Die buckets des K-d Baumes und die Voronoi-Gebiete zu den Codebuchvektoren liefern zwei unabhängige Partitionierungen des k-dimensionalen Suchraumes, die beide aus disjunkten Gebieten bestehen. Wie schon gezeigt wurde, kann man mit wenig Aufwand einen Testvektor $\mathbf{x}$ durch einen K-d Baum in einem der buckets lokalisieren. Das Problem beim FBF-Algorithmus ist nur, daß die Codebuchvektoren, die in dem lokalisierten bucket liegen, nicht die einzigen Kandidaten für einen Nächsten-Nachbarn sind.
Vielmehr muß man all diejenigen Vektoren aus dem Codebuch in Betracht ziehen, deren Voronoi-Gebiete mit dem lokalisierten bucket überlappen. Der Testvektor $\mathbf{x}$ kann nur in einem dieser Voronoi-Gebiete liegen, daher liefert die vollständige Suche unter den zugehörigen Codebuchvektoren den korrekten Nächsten-Nachbarn.
Die analytische Berechnung der Grenzen der Voronoi-Gebiete der Codebuchvektoren in hochdimensionalen Räumen ist sehr komplex. Man nähert daher die Voronoi-Gebiete durch Hyperquader an, die man durch Codieren einer genügend großen Zahl von Trainingsvektoren berechnet. Dabei nimmt man allerdings Fehler in Kauf, so daß die BVI-Suche nicht in jedem Fall den Nächsten-Nachbarn findet, sondern eventuell auch nur den zweit- oder drittnächsten...
Um diese Fehlerrate so klein wie möglich zu halten, muß man unter Umständen eine sehr große Menge an Trainingsvektoren zur Bestimmung der Voronoi-Näherungen zur Verfügung haben.
Je größer man den K-d Baum wählt, desto kleiner werden die buckets oder Hyperquader. Wenn die buckets kleiner werden, sinkt auch die Anzahl der Code-

buchvektoren, deren Voronoi-Gebiete mit einem bestimmten bucket überlappen, somit gibt es weniger Kandidaten, unter denen man noch suchen muß.
Mit steigender Tiefe des K-d Baumes sinkt also die Komplexität der Nächster-Nachbar Suche, wobei aber der Speicherplatzbedarf des Baumes exponentiell ansteigt, sodaß der Komplexitätsreduktion in der Praxis Grenzen gesetzt sind.

4.2 Beispiel zur BVI-Suche

Abbildung 7 illustriert die BVI Suche mit K-d Bäumen anhand eines Gebietes des $\mathcal{R}^2$, das durch den K-d Baum aus Abbildung 5 partitioniert wird. Der Testvektor $\mathbf{x} = (x_1, x_2)$ wird durch den K-d Baum im bucket IEQR lokalisiert. Um den Nächsten-Nachbarn von $\mathbf{x}$ zu finden, genügt es unter denjenigen Codebuchvektoren zu suchen, deren Voronoi-Gebiete mit dem bucket IEQR überlappen. Diese Codebuchvektoren sind in der Abbildung mit zusätzlichen Kreisen markiert.

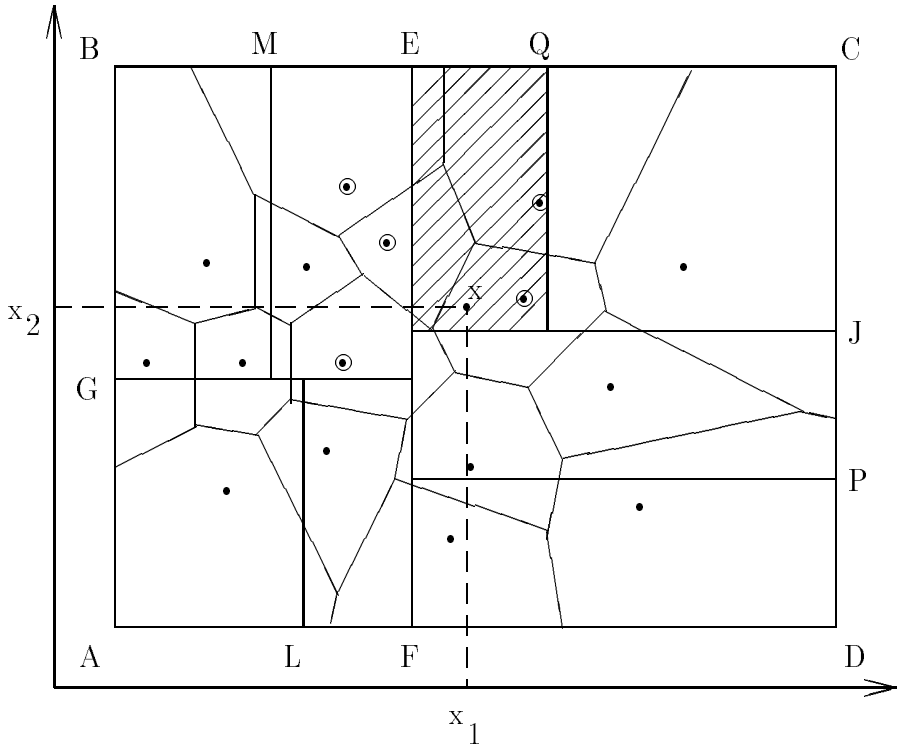

Abbildung 7: Bucket Voronoi Intersection Suche unter $N = 16$ Codebuchvektoren im $\mathcal{R}^2$ mit einem K-d Baum.

Im Blatt IEQR des K-d Baumes werden in einer Liste Referenzen auf diese 5 Codebuchvektoren gespeichert. Analog dazu werden in allen anderen Blättern des K-d Baumes Referenzen auf Codebuchvektoren gehalten, deren Voronoi-Gebiete mit dem zum jeweiligen Blatt gehörenden bucket überlappen.
Im Falle des Testvektors $\mathbf{x}$ muß man, nach vergleichsweise vernachlässigbaren 3

skalaren Vergleichen zur Bestimmung des buckets IEQR, den korrekten Nächsten-Nachbarn nur noch unter 5 statt ursprünglich 16 Codebuchvektoren suchen. Offensichtlich könnte man den Suchaufwand durch Erhöhen des K-d Baumes noch weiter senken, da die buckets dann noch kleiner werden und dadurch mit weniger Voronoi-Gebieten überlappen würden.
Der Kern des BVI Algorithmus ist die geschickte Wahl der Trennhyperbenen, also das Design des K-d Baumes, in der Vorverarbeitungsphase. Die eigentliche Vektorquantisierung mit dem fertigen K-d Baum ist vergleichsweise einfach. Der BVI-Algorithmus bringt aber nur dann signifikante Vorteile gegenüber anderen, schnellen Nächster-Nachbar Suchverfahren, wenn man geeignete Optimierungskriterien für den K-d Baum, im Sinne der BVI-Suche, findet. Das Design BVI-optimierter K-d Bäume steht deshalb im Vordergrund dieser Arbeit und ist Thema des nachfolgenden Kapitels.

5 Optimierung von K-d Bäumen zur BVI-Suche

5.1 Problemstellung

Gegeben sei ein Codebuch $C = \{\mathbf{c_1}, \ldots, \mathbf{c_N}\}$ bestehend aus N Vektoren $\mathbf{c_i}$ der Dimension K, $\mathbf{c_i} = (c_{i1}, \ldots, c_{iK})$. Zu diesem Codebuch soll ein optimierter K-d Baum der Tiefe d erstellt werden. Sei der K-dimensionale Hyperquader $R_0 = [a_1, b_1] \times \cdots \times [a_K, b_K]$ das initiale Gebiet, in dem alle zu quantisierenden Vektoren enthalten sind. Dies kann zum Beispiel der Einheitshyperwürfel $R_0 = [0,1]^K$ sein. Ein K-d Baum der Tiefe d zerlegt dieses initiale Gebiet vollständig in $M = 2^d$ disjunkte Hyperquader (buckets) $B_i, \quad i = 1, \ldots, M$. Sei $p_i = p(\mathbf{x} \in B_i)$ die Wahrscheinlichkeit dafür, daß der Vektor $\mathbf{x}$ im bucket B_i enthalten ist. Seien weiterhin $V_i, \quad i = 1, \ldots, N$ die zu den Codebuchvektoren korrespondierenden Voronoi-Gebiete, die ebenfalls das initiale Gebiet R_0 vollständig zerlegen und untereinander disjunkt sind.
Sei $n_i = \sharp\{V_j : V_j \cap B_i \neq \emptyset\}, \quad j = 1, \ldots, N$ die Anzahl der Voronoi-Gebiete, die mit dem bucket B_i überlappen. Unter der Vorraussetzung, daß $\mathbf{x} \in B_i$, genügt es den Nächsten-Nachbarn von $\mathbf{x}$ unter den n_i Codebuchvektoren zu suchen, was einem Aufwand von n_i entspricht. Der mittlere Aufwand für eine Suche im bucket B_i beträgt $p_i n_i$. Der Erwartungswert des mittleren Aufwands für die Suche nach einem beliebigen $\mathbf{x} \in R_0$ ist damit $E = \sum_{i=1}^{M} p_i n_i$. Diese Zielfunktion zu minimieren, entspricht einem globalen Optimierungsproblem mit $2M$ Variablen p_i und n_i, die ihrerseits wiederum von der Lage (j,h) jeder einzelnen Trennhyperebene in den inneren Knoten des Baumes abhängen. Das globale Minimum der Zielfunktion E zu finden, verlangt daher eine gemeinsame Optimierung der $2^d - 1$ Paare (j,h) des K-d Baumes. Die globale Optimierung ist offensichtlich extrem komplex und es bleibt einem nichts anderes übrig, als auf lokale Optimierungsverfahren auszuweichen, die aber nur ein Suboptimum (im Sinne des mittleren erwarteten

Suchaufwandes) finden.
Der hier verfolgte Ansatz minimiert den mittleren erwarteten Suchaufwand lokal in jedem inneren Knoten des Baumes. Man könnte sich aber auch andere lokale Optimierungsstrategien überlegen. Die in den nächsten beiden Kapiteln vorgestellten Kriterien optimieren die Lage der Trennhyperebenen lokal und unabhängig voneinander in jedem inneren Knoten. Dies erfordert nur die gemeinsame Optimierung der zwei Variablen (j, h), wobei j sogar nur ganzzahlige Werte im Intervall $[1, \ldots, K]$ annehmen kann (h ist reell).

5.2 Das Exact Optimization Criterion (EOC)

Das hier vorgestellte Optimierungskriterium ist für den Fall bekannter Testdatenverteilung anwendbar. Bei Anwendungen der Vektorquantisierung existieren große Mengen von Trainingsvektoren, die schon zur Bestimmung eines Codebuchs (z. B. durch den LBG-Algorithmus) benötigt wurden. Diese Trainingsvektoren werden nun auch zur näherungsweisen Bestimmung der Testdatenverteilung herangezogen. Dabei wird die Näherung natürlich um so besser, je mehr Trainingsvektoren verwendet werden. Die gesamte Optimierung ist liefert aber auch sehr schlechte Ergebnisse, wenn man die Testdatenverteilung durch eine zu kleine Menge an Trainingsvektoren viel zu ungenau bestimmt hat.
Sei $R = \{\mathbf{x} \in \mathcal{R}^K : a_j \leq x_j \leq b_j, j = 1, \ldots, K\}$ ein Hyperquader des $\mathcal{R}^K$ der durch eine noch zu bestimmende Hyperebene in zwei neue Hyperquader R_L und R_R aufgeteilt werden soll. Die Trennhyperebene $H = \{\mathbf{x} : x_j = h; a_j \leq h \leq b_j\}$ liegt senkrecht zu einer der K Koordinatenachsen und innerhalb von R. Im folgenden sei diese Hyperebene durch (j, h) repräsentiert. Abbildung 8 zeigt die Situation der lokal in einem Knoten des K-d Baumes stattfindenden Optimierung.

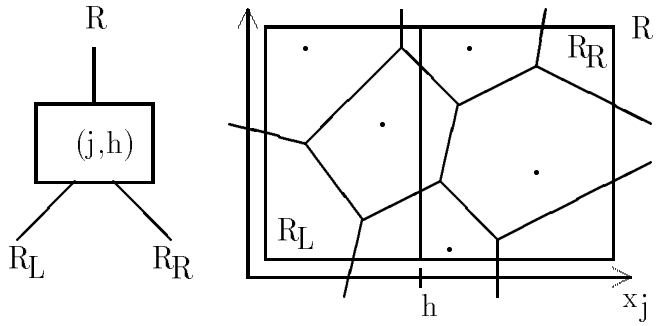

Abbildung 8: Lokaler Optimierungsschritt in einem inneren Knoten

Für die Hyperquader R_L und R_R mit

$$R_L(j,h) = \{\mathbf{x} \in \mathcal{R}^K : x_j \leq h\} \quad \text{und} \quad R_R(j,h) = \{\mathbf{x} \in \mathcal{R}^K : x_j > h\}$$

kann man nun die Wahrscheinlichkeiten angeben, mit denen ein Vektor $\mathbf{x}$, der in R liegt, auch in R_L oder R_R liegt:

$$p_L(j,h) = \frac{p(\mathbf{x} \in R_L)}{p(\mathbf{x} \in R)} \quad \text{und} \quad p_R(j,h) = \frac{p(\mathbf{x} \in R_R)}{p(\mathbf{x} \in R)} \quad \text{mit} \quad p_L + p_R = 1$$

Seien n,$n_L(j,h)$ und $n_R(j,h)$ die Anzahl der Voronoi-Gebiete, die mit dem Hyperquader R, R_L bzw. R_R überlappen, wobei n_L und n_R von der Lage der Trennhyperebene abhängen. Da die neuen Hyperquader R_L und R_R vollständig in R enthalten sind, gelten außerdem die folgenden Bedingungen:

$$n_L \leq n, \quad n_R \leq n, \quad n \leq n_L + n_R \leq 2n$$

Anhand eines Testvektors $\mathbf{x}$, der in R liegt, untersucht man nun die Reduktion der Nächster-Nachbar Suche. Durch einen Vergleich der Form $x_j \leq h$ lokalisiert man den Testvektor in einem der beiden Hyperquader R_L oder R_R. Um den Nächsten-Nachbarn zu bestimmen, muß man dann nur noch unter n_L bzw. n_R, statt ursprünglich n Codebuchvektoren suchen. Nach einem skalaren Vergleich hat man die Komplexität also entweder von n auf n_L oder von n auf n_R reduziert. Der Erwartungswert der Komplexität bei Verwendung der Trennhyperebene (j,h) reduziert sich von $E(R) = n$ auf

$$E(R,j,h) = p_L(j,h)n_L(j,h) + p_R(j,h)n_R(j,h)$$

Bei einer idealen Unterteilung durch eine Hyperebene (j,h) würde man durch $p_L = p_R = 1/2$ und $n_L = n_R = n/2$ erreichen, was aber in der Praxis absolut unwahrscheinlich ist, da durch die Trennhyperebene immer einige Voronoi-Gebiete durchtrennt werden. Im Idealfall würde die Komplexität nur noch $E(R,j,h) = n/2$ betragen, also die Hälfte der ursprünglichen Komplexität. Dies stellt die untere Grenze der erreichbaren Verbesserung in einem inneren Knoten des K-d Baumes dar.
Die optimale Trennhyperebene (j^*,h^*) ist diejenige, die den mittleren erwarteten Aufwand minimiert:

$$E(R,j^*,h^*) \leq E(R,j,h) \quad \forall (j,h) : 1 \leq j \leq K, a_j \leq h \leq b_j$$

Diese Bedingung wird *Exact Optimization Criterion (EOC)* genannt. Zur Bestimmung der optimalen Hyperebene kann man durch ein eindimensionales Optimierungsverfahren die Minima $E_j(R,h)$ von $E(R,j,h)$ für festes $j = 1,\ldots,K$ bestimmen, um dann in einem letzten Schritt das Minimum aller E_j zu suchen. Die zur Berechnung von $E(R,j,h)$ benötigten Funktionen $p_L(j,h)$, $n_L(j,h)$ und $n_R(j,h)$ können für das Gebiet R unter Verwendung einer großen Trainingsdatenmenge vorberechnet werden $(p_R(j,h) = 1 - p_L(j,h))$.
Zur Vorberechnung der Funktion

$$p_L(j,h) = p(\mathbf{x} \in R : x_j < h)$$

bestimmt man zuerst für jede Koordinatenachse j ein Histogramm mit den Trainingsvektoren, die im Gebiet R liegen. Ob ein Trainingsvektor in R liegt, kann man für jeden inneren Knoten durch den bereits berechneten Baum oberhalb dieses Knotens feststellen. Soll nun die Funktion $p_L(j,h)$ für ein Argument (j,h) ausgewertet werden, berechnet man mit dem Histogramm der j-ten Achse eine Näherung des unbekannten, korrekten Funktionswerts. Die Güte dieser Näherung hängt von der Größe der Trainingsdaten und von der Auflösung des Histogramms ab.
Um $n_L(j,h)$ und $n_R(j,h)$, die Anzahl der Voronoi-Gebiete die mit R_L und R_R überlappen zu bestimmen, benutzt man das schon vorgestellte Verfahren der Voronoi-Projektion. Projeziert man die n Voronoi-Gebiete, die mit dem Gebiet R überlappen auf eine Koordinatenachse j, so erhält man n überlappende Intervalle $(P_1^j, P_2^j, \ldots, P_n^j)$. Jedes Intervall P_i^j besteht aus einer Untergrenze $P_{i,L}^j$ und einer Obergrenze $P_{i,U}^j$ die wie folgt definiert sind:

$$P_{i,L}^j = \min_{\mathbf{x}\in R} x_j : d(\mathbf{x}, \mathbf{c_i}) \leq d(\mathbf{x}, \mathbf{c_l}) \quad \forall l = 1, \ldots, N$$

$$P_{i,U}^j = \max_{\mathbf{x}\in R} x_j : d(\mathbf{x}, \mathbf{c_i}) \leq d(\mathbf{x}, \mathbf{c_l}) \quad \forall l = 1, \ldots, N$$

Diese beiden Werte können wiederum nur näherungsweise, durch Kodieren einer großen Zahl von Trainingsvektoren gewonnen werden. Aus diesen Projektionsgrenzen kann man nun zu gegebenem (j,h) direkt die Funktionen n_L und n_R bestimmen. $n_L(j,h)$ ist gleich der Anzahl der Obergrenzen der Projektionen P^j, die kleiner als h sind. $n_R(j,h)$ ist gleich der Anzahl der Untergrenzen der Projektionen P^j, die größer als h sind:

$$n_L(j,h) = \sharp\{P_{i,L}^j : P_{i,L}^j < h\} \qquad n_R(j,h) = \sharp\{P_{i,U}^j : P_{i,U}^j > h\}$$

Die Bestimmung der Histogramme und der Voronoi-Projektionen kann zusammen durchgeführt werden, damit man für jeden inneren Knoten des K-d Baumes nur einmal durch die Menge der Trainingsvektoren durchgehen muß. Obwohl beide Informationen, Histogramme und Voronoi-Projektionen, nur näherungsweise bestimmt werden können, zeigt sich in der Praxis, daß die dadurch verursachten Fehler akzeptiert werden können, wenn die Trainingsdatenmenge groß genug gewählt wurde.
Abbildung 9 zeigt die Projektion von Voronoi-Gebieten auf eine Achse x_j. Es sind alle entstehenden Intervalle sowohl einzeln, als auch gemeinsam auf der Achse dargestellt. Projektionslinien wurden, der Übersicht halber, nur zu 3 Voronoi-Gebieten eingezeichnet. Intervalluntergrenzen der Projektion des Voronoi Gebiets V_i sind mit einem L_i, Intervallobergrenzen mit einem U_i gekennzeichnet.

5.3 Das Generalized Optimization Criterion (GOC)

Im Falle unbekannter Testdatenverteilung kann man zur Bestimmung einer optimalen Trennhyperebene nur die Voronoi-Informationen in Form der Funktionen

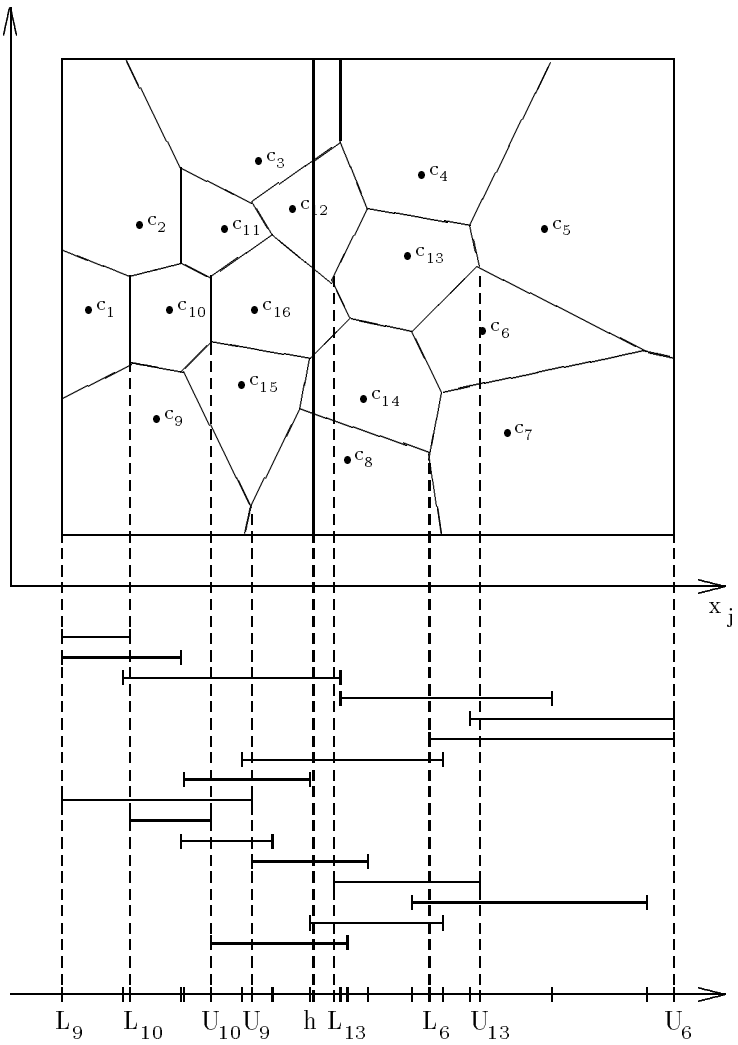

Abbildung 9: Voronoi-Projektionen zur Bestimmung von n_L und n_R

$n_L(h)$ und $n_R(h)$ heranziehen. Es stellt sich also die Frage, nach welcher Strategie man jetzt die Position der Hyperebene bestimmt. Eine "gute" Position h für die Trennebene ist sicher dann erreicht, wenn dadurch sowohl $n_L(h)$, als auch $n_R(h)$ so klein wie möglich werden. Außerdem hat man ja auch eine Wahl zwischen den K möglichen Koordinatenrichtungen zu wählen, zu der die Trennebene senkrecht stehen soll.
Eine beliebige Trennhyperebene durchschneidet stets eine gewisse Anzahl von Voronoi-Gebieten im $\mathcal{R}^K$. Dies ist in der Praxis unvermeidbar aber man kann zumindest versuchen, diese Anzahl so klein wie möglich zu halten, da der Codebuchvektor zu jedem durchschnittenen Voronoi-Gebiet in beiden Teilräumen berücksichtigt werden muß und somit nicht zur Reduktion der Suchkomplexität beiträgt. Um zu einem guten Optimierungskriterium für den Fall unbekannter

Testdatenverteilung zu gelangen, wird nun im folgenden das Verhalten der Funktionen $n_L(h)$ und $n_R(h)$ näher untersucht und geometrisch interpretiert.
Sei n die Anzahl der Voronoi-Gebiete, die mit dem vorgegebenen Hyperquader R überlappen. Durch Projektion dieser Voronoi-Gebiete auf eine der insgesamt K Koordinatenachsen erhält man n sich überlappende Intervalle $(P_1, \ldots, P_n)$ (Abb. 10). Die $2n$ Grenzen dieser Intervalle werden mit einer Marke (Label) versehen, die sie als Unter- oder Obergrenze auszeichnen (L für eine Untergrenze und U für eine Obergrenze).
Die Labels der aufsteigend sortierten $2n$ Grenzen der Intervalle bilden eine Folge der Art $(LLULU\ldots)$. Zur Vereinfachung wird angenommen, daß die Grenzen der Interalle paarweise verschieden sind. Dies ist in der Praxis nicht der Fall, spielt aber in der theoretischen Betrachtung keine Rolle. Angenommen, wir betrachten die Projektionen der Voronoi-Gebiete auf die j-te Koordinatenachse. Die $2n$ aufsteigend sortierten Projektionsgrenzen bilden $2n-1$ aneinandergereihte, disjunkte Intervalle $(I_1, I_2, \ldots, I_{2n-1})$ im Bereich (a_j, b_j) (siehe Abb. 10).

Betrachten wir nun das Verhalten der Funktionen $n_L(h)$ und $n_R(h)$, wenn h sich im Intervall (a_j, b_j) von links nach rechts bewegt. Für $h = a_j$ ist $n_L(h) = 0$ und $n_R(h) = n$. Bewegt sich h nur um einen winzigen Betrag von a_j weg, so ist $n_L(h) = 1$ und $n_R(h) = n - 1$. Wann immer h sich über eine Intervallgrenze hinwegbewegt, ändert sich eine der beiden Funktionen:

- Ist die Intervallgrenze eine Untergrenze (L), so kommt ein neues Voronoi-Gebiet hinzu und n_L wird um 1 erhöht. Der Wert von n_R bleibt gleich, da sich die Anzahl der Voronoi-Gebiete, die rechts von h liegen nicht verändert.

- Ist die Intervallgrenze eine Obergrenze (U), so wird ein Voronoi- Gebiet ausgeschlossen und n_R wird um 1 vermindert. Der Wert von n_L bleibt gleich, da sich die Anzahl der Voronoi-Gebiete, die links von h liegen nicht verändert.

Bewegt sich h zwischen zwei aufeinanderfolgenden Intervallgrenzen, so ändert sich weder der Wert von $n_L(h)$ noch von $n_R(h)$. Am rechten Rand, in einem sehr kleinen Abstand von b_j ist $n_L(h) = n - 1$ und $n_R(h) = 1$, für $h = b_j$ ist $n_L(h) = n$ und $n_R(h) = 0$.
Im untersten Diagramm der Abbildung 10 wird das Verhalten der Funktion $(n_L - n_R)(h)$ für h zwischen a_j und b_j veranschaulicht. Solange sich h zwischen zwei aufeinanderfolgenden Intervallgrenzen bewegt ändert sich weder $n_L(h)$ noch $n_R(h)$ und somit bleibt auch die Differenz der Beiden konstant. Sobald sich h jedoch über eine Intervallgrenze hinweg bewegt, wird entweder n_L um 1 größer oder n_R um 1 kleiner. In beiden Fällen erhöht sich daher die Differenz der beiden Werte um 1. Somit ist die Funktion $(n_L - n_R)(h)$ monoton steigend und besitzt einen treppenförmigen Verlauf. Am linken Rand bei $h = a_j$ hat die Funktion den Wert $-n$, am rechten Rand bei $h = b_j$ den Wert n. Da sich $(n_L - n_R)(h)$ nur an Intervallgrenzen ändert, hat sie innerhalb eines beliebigen Intervalls I_l den Wert

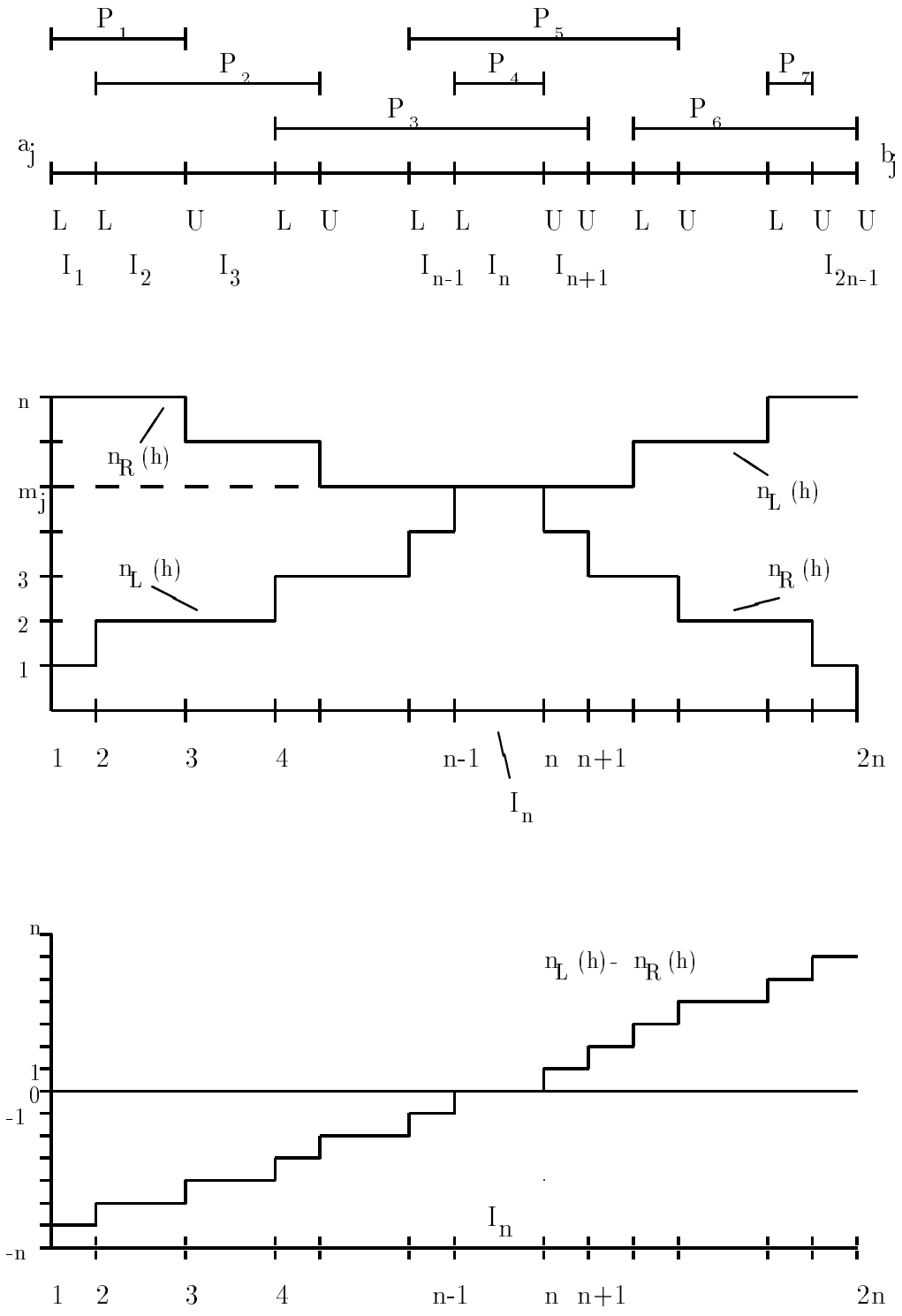

Abbildung 10: Typischer Verlauf der Funktionen n_L, n_R und $(n_L - n_R)$ zu gegebenen Voronoi-Projektionen, abhängig von der Position h der Trennhyperebene

$-n + l$, insbesondere nimmt die Funktion innerhalb des Intervalls I_n den Wert 0 an. Für h innerhalb I_n sind also n_L und n_R gleich groß, was einer balancierten Unterteilung entspricht, bei der sich gleich viele Voronoi-Regionen links wie rechts der Trennhyperebene befinden.
Für $h \in I_n$ beträgt der erwartete Suchaufwand sowohl links, als auch rechts der Trennhyperebene $E(h) = p(h)n_L(h) + (1 - p(h))n_R(h) = m_j$. Wählt man h innerhalb I_n, so kann man nicht erwarten die optimale Position der Trennhyperebene auf der Achse j gefunden zu haben, da keinerlei Information über $p(h)$ vorliegt. Trotzdem ist diese Position allen anderen vorzuziehen, da sie zu einem balancierten K-d Baum führt.
Wie sieht nun ein lokaler Optimierungsschritt innerhalb eines Knotens des K-d Baumes aus? Man bestimmt für jede der K Koordinatenachsen eine Position h so, daß eine balancierte Unterteilung entsteht ($n_L(h) = n_R(h)$) bei der Suchaufwand $m_j = n_L(h) = n_R(h)$ betragen würde. Man erhält so eine Menge von K möglichen Positionen für eine Hyperebene mit zugehörigem Suchaufwand : $\{(h_1, m_1), \ldots, (h_j, m_j), \ldots, (h_K, m_K)\}$. Die optimale Achse für die Positionierung einer Trennhyperebene ist diejenige, für die m_j minimal wird. Dies ist jedoch nicht immer eindeutig, da der minimale Suchaufwand m_j eventuell bei mehreren Achsen erreicht werden kann.
Im folgenden soll nun das beschriebene Optimierungskriterium noch weiter verfeinert werden. Wir wählen dazu eine geometrische Betrachtungsweise der Funktionen n_L und n_R. Zu beliebigem h kann man die Funktionswerte der beiden Funktionen zu einem Tupel zusammenfassen und in einem 2-dimensionalen Koordinatensystem darstellen. Auf der x-Achse trägt man den Wert von $n_L(h)$ und auf der y-Achse den Wert von $n_R(h)$ ab. Für $a_j \leq h \leq b_j$ erhält man auf diese Weise für jede der K Koordinatenachsen eine durch h parametrisierte Kurve in der n_L-n_R-Ebene.

Abbildung 11 veranschaulicht dies in einem n_L-n_R-Koordinatensystem, in dem allerdings nur zwei Kurven eingezeichnet sind (eine davon gestrichelt). Für alle Kurven gilt, daß $0 \leq n_L(h), n_R(h) \leq n$. Eine Kurve kann sich also nur innerhalb des dadurch bestimmten Quadrates bewegen.
Betrachten wir nun die zu einer Koordinatenachse j gehörende Kurve. Seien $N_L(h)$ und $N_R(h)$ die Anzahl der Voronoi-Gebiete, die vollständig links bzw. rechts der Hyperebene $x_j = h$ liegen. Des weiteren sei $N_S(h)$ die Anzahl der Voronoi-Gebiete die durch die Trennhyperebene gespalten werden. Offensichtlich gelten dann für ein beliebiges h folgende Bedingungen:

$$N_L + N_R + N_S = n \quad , \quad N_L + N_S = n_L \quad , \quad N_R + N_S = n_R$$

Bestimmen wir nun daraus das Gebiet in der n_L-n_R-Ebene, in dem sich eine beliebige Kurve befinden kann. Aus den obigen Gleichungen folgt direkt $n_L + n_R = n + N_S$. Durch eine beliebige Hyperebene $x_j = h$ werden minimal kein Einziges und maximal alle Voronoi-Gebiete gespalten, d. h. $0 \leq N_S \leq n$. Damit kann man den Wert von $n_L + n_R$ nach oben und unten abschätzen und erhält damit

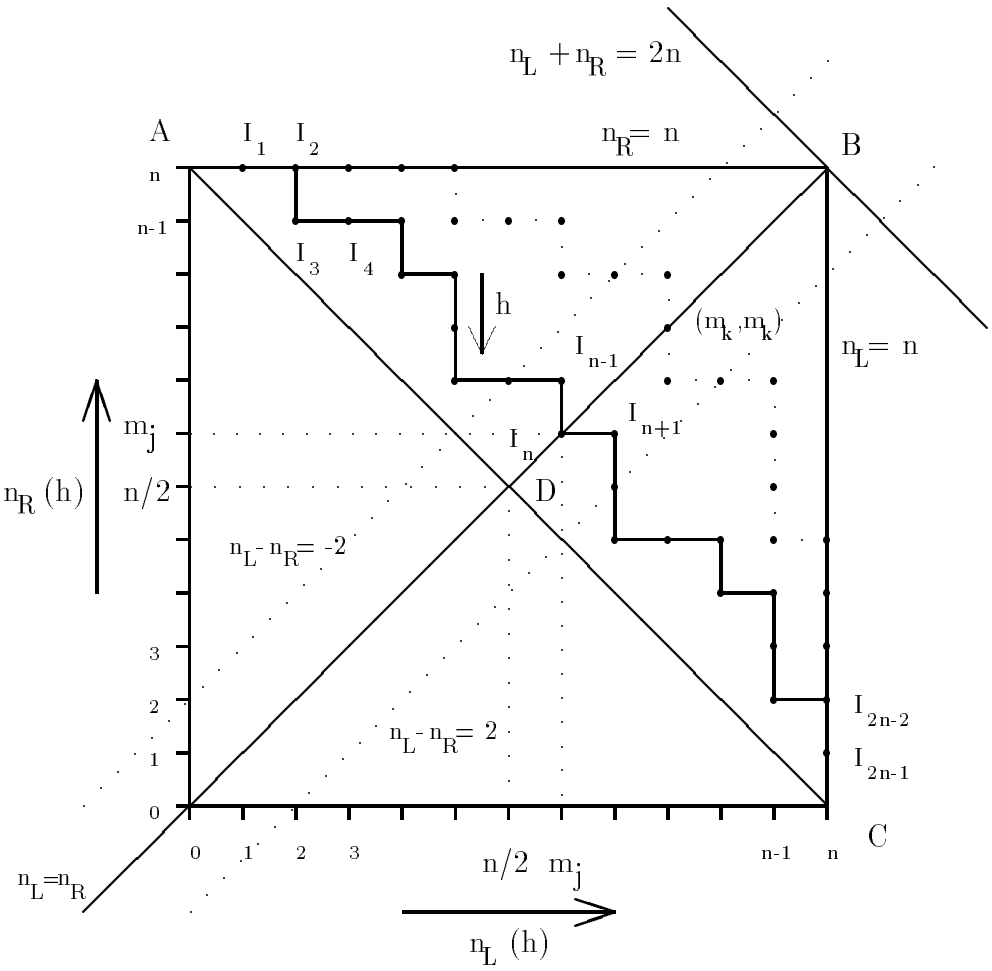

Abbildung 11: Geometrische Interpretation des GOC Optimierungskriteriums dargestellt in der n_L-n_R-Ebene

$n \leq n_L + n_R \leq 2n$. Diese Bedingung definiert einen diagonalen Streifen in der n_L-$n - R$-Ebene. Zusammen mit den oben schon genannten Bedingungen gilt für alle Kurven:

$$0 \leq n_L \leq n \quad , \quad 0 \leq n_R \leq n \quad , \quad n \leq n_L + n_R \leq 2n$$

Dieses dreieckige Gebiet ist in Abbildung 11 durch die Punkte ABC gekennzeichnet.
Für $h = a_j$ startet jede Kurve im Punkt A und endet für $h = b_j$ im Punkt C. Der Verlauf einer Kurve zwischen diesen beiden Endpunkten ist durch die Lage der Intervallgrenzen der Voronoi-Projektionen in der entsprechenden Koordinate bestimmt, genauer gesagt von der sortierten Folge der Labels der Unter- und Obergrenzen der Intervalle. Da diese im Allgemeinen für jede Koordinate unterschiedlich sind, werden auch die Kurven einen unterschiedlichen Verlauf zeigen.

Grundsätzlich ändert sich die Lage des Punktes (n_L, n_R), dessen Verlauf die Kurve bestimmt, zwischen zwei aufeinanderfolgenden Intervallgrenzen nicht. Interessant ist nur der Fall, daß h eine Intervallgrenze passiert:

- Falls es sich um eine Untergrenze (L) handelt, so wird die Anzahl der linken Voronoi-Gebiete n_L um eins erhöht, während sich die Anzahl der rechten Voronoi-Gebiete n_R nicht verändert. Die Kurve verläuft also vom Punkt (n_L, n_R) horizontal nach rechts zum Punkt $(n_L + 1, n_R)$.

- Falls es sich um eine Obergrenze (U) handelt, so sinkt die Anzahl der rechten Voronoi-Gebiete n_R um eins, während die Anzahl der linken Voronoi-Gebiete n_L konstant bleibt. In diesem Fall verläuft die Kurve vom Punkt (n_L, n_R) vertikal nach unten zum Punkt $(n_L, n_R - 1)$.

Erreicht h das Intervall I_n, so befindet sich der Punkt (n_L, n_R) auf der diagonalen Geraden $n_L = n_R$. Wie weiter oben bereits erläutert wurde, entspricht dies einer ausgeglichenen Unterteilung bei der gleich viele Voronoi-Gebiete links wie rechts der Hyperebene liegen. Eine solche Unterteilung wäre optimal, wenn dadurch kein einziges Voronoi-Gebiet gespalten würde, also $N_S = 0$ und damit $n_L + n_R = n$. Eine Kurve wäre also optimal, wenn sie durch den Punkt $(n/2, n/2)$ verlaufen würde. Dies ist wie gesagt in der Praxis nicht erreichbar, aber beim GOC-Kriterium wählt man aus den K möglichen Koordinatenachsen diejenige zur Unterteilung aus, bei der der Schnittpunkt der zugehörigen Kurve mit der Geraden $n_L = n_R$ am nächsten am Punkt $(n/2, n/2)$ liegt.
In Abb. 11 sind zwei Kurven zu den Koordinatenachsen j (durchgezogene Linie) und k (gestrichelt) eingezeichnet. Für $h \in I_n$ passieren die Kurven die Punkte (m_j, m_j) bzw. (m_k, m_k). In diesem Fall wird man sich für eine Trennhyperebene senkrecht zur Achse j entscheiden, da der Punkt (m_j, m_j) näher an $(n/2, n/2)$ liegt als der Punkt (m_k, m_k).
Bevor nun das Generalized Optimization Criterion exakt definiert werden kann, muß man noch die Einschränkung aufheben, daß die Grenzen der Intervalle der projezierten Voronoi-Gebiete paarweise verschieden sind. Man muß auf diese Einschränkung verzichten, da beim Aufbau eines K-d Baumes die zu den Knoten gehörenden Hyperquader rekursiv immer weiter unterteilt werden, so daß irgendwann die meisten Voronoi-Gebiete größer als die Hyperquader werden. Die Projektionsintervalle ragen dann über die Ränder des Hyperquaders hinaus und werden auf die Intervalle (a_j, b_j) abgeschnitten. Dadurch liegen mehrere Intervalluntergrenzen (mit L gelabelt) am linken Rand bei a_j und auch mehrere Intervallobergrenzen (mit U gelabelt) am rechten Rand bei b_j. Aber auch sonst kann der Fall auftreten, daß mehrere Intervallgrenzen in einem Punkt zusammenfallen.
In unserer geometrischen Sicht in der n_L-n_R-Ebene entspricht dies aber dem Fall, daß sich ein Punkt $(n_L(h), n_R(h))$ auf einer Kurve nicht nur um einen Schritt nach rechts oder nach unten bewegen kann, wenn h eine Intervallgrenze passiert, sondern auch um mehrere Schritte nach rechts oder nach unten, wenn h mehrere

Intervallgrenzen gleichzeitig passiert.
Das Problem ist nun, daß eine beliebige Kurve im allgemeinen keinen stationären Punkt mehr auf der Geraden $n_L = n_R$ besitzt, sodaß man die optimale Kurve nicht mehr anhand der Lage der Kreuzungspunkte der Kurven mit der $n_L = n_R$-Geraden bestimmen kann. Außerdem ist dann eine balancierte Unterteilung mit gleicher Anzahl an Voronoi-Gebieten links und rechts der Trennhyperebene nicht mehr möglich.
Mit anderen Worten, die Funktion $(n_L - n_R)(h)$ hat für ein h im Intervall I_n nicht immer den Funktionswert 0, so daß man im n_L-n_R- Schaubild nicht nur Schnittpunkte der Kurven mit der Geraden $n_L = n_R$, sondern auch mit den Geraden $n_L - n_R = c$ und $n_L - n_R = -c$ berücksichtigen muß, die in geringem Abstand parallel zu $n_L = n_R$ verlaufen.
Ein allgemeineres Kriterium sollte also als Bedingung für einen Punkt mit optimaler Unterteilung nicht $n_L(h) = n_R(h)$, sondern $\min |n_L(h) - n_R(h)|$ verlangen, um gute unbalancierte Unterteilungen zu erhalten, die so nahe wie möglich an der (unerreichbaren) balancierten Unterteilung liegen. Als Maß für die "Nähe" einer Unterteilung zur Optimalen kann man zum Beispiel den Euklidischen Abstand zwischen dem Punkt $(n_L(h), n_R(h))$ und dem Punkt $(n/2, n/2)$ wählen. Das Generalized Optimization Criterion (GOC) präsentiert sich daher wie folgt:

1.) Bestimme zu jeder der K Koordinatenachsen die optimale Position h_j^* einer Trennhyperebene, sodaß $|n_L(h_j^*) - n_R(h_j^*)|$ minimiert wird.

2.) Aus den K Kandidaten $(j, h_j^*), j = 1, \ldots, K$ für eine Trennhyperebene wird das Paar (k, h_k^*) ausgewählt, für das der Abstand des zugehörigen Punktes $(n_L(h_k^*), n_R(h_k^*))$ zum Punkt $(n/2, n/2)$ minimal im Sinne des Euklidischen Abstandsmaßes ist. n ist dabei die Anzahl der Voronoi-Gebiete, die mit dem Hyperquader überlappen, für den eine Trennhyperebene bestimmt werden soll.

Genau wie beim Exact Optimization Criterion (EOC) berechnet man die Werte der Funktionen $n_L(h)$ und $n_R(h)$ für den zu unterteilenden Hyperquader mittels der Projektionen der Voronoi-Gebiete.

5.4 Design von K-d Bäumen mit EOC oder GOC

Mit den vorgestellten Optimierungskriterien kann man lokal für einen Knoten des K-d Baumes eine optimale Hyperebene bestimmen. Vorraussetzung für einen solchen Optimierungsschritt ist nur, daß die Vorfahren des aktuellen Knotens im Baum bereits berechnet wurden. Daher wird der K-d Baum beginnend mit dem Wurzelknoten im Stile einer Breiten- oder Tiefensuche abgearbeitet und dabei gleichzeitig aufgebaut.
Am Ende erhält man einen K-d Baum der Tiefe d mit 2^d buckets als Blätter.

Jedes dieser buckets repräsentiert einen Hyperquader, zu dem die Liste von Codebuchvektoren bestimmt wird, deren Voronoi-Gebiete mit diesem Hyperquader überlappen. Diese Liste wird auch *bucket-Voronoi intersection list (BVI list)* genannt.
Die Nächster-Nachbar Suche zu einem vorgegebenen Testvektor $\mathbf{x}$ verläuft wie bei allen Verfahren, die K-d Bäume verwenden, in zwei Schritten:

Schritt 1: Mit d skalaren Vergleichen wird das bucket im K-d Baum bestimmt, das den Testvektor $\mathbf{x}$ beinhaltet.

Schritt 2: Anhand der BVI Liste des gefundenen buckets wird eine lineare, vollständige Suche unter den verbliebenen Kandidaten des Codebuchs durchgeführt.

Der mittlere Suchaufwand entspricht der mittleren Anzahl an Codebuchvektoren in den buckets, also der mittleren Größe der BVI-Listen. Im schlechtesten Fall hat der Suchalgorithmus einen Aufwand proportional zur größten BVI-Liste im Baum.
Der Suchaufwand sinkt streng monoton mit wachsender Tiefe des K-d Baumes. Der Tiefe eines K-d Baumes sind aber in der Praxis durch den exponentiellen Speicherbedarf Grenzen gesetzt. Bezeichnet man die mittlere Größe der BVI-Listen mit $\bar{b}$ und die Tiefe des K-d Baumes wie bisher mit d, so erhält man für den Speicherbedarf M:

$$M = 2(2^d - 1) + (\bar{b} + 1)2^d \approx (3 + \bar{b})2^d$$

Erhöht man einen K-d Baum um eine Schicht, so ist bei der NN-Suche nur ein zusätzlicher skalarer Vergleich erforderlich, der Speicherbedarf des Baumes wird aber nahezu verdoppelt.
Falls man Informationen über die Testdatenverteilung besitzt, wie es beim EOC Kriterium der Fall ist, so kann man die Suchkomplexität weiter verringern, indem man die Codebuchvektoren in den buckets in absteigender Reihenfolge entsprechend ihrer Wahrscheinlichkeit, Nächster-Nachbar zu sein, anordnet und bei der erschöpfenden Suche eine partielle Distanzberechnung verwendet.

6 Implementierung des BVI-Algorithmus

Nach den bisher eher theoretischen Ausführungen soll nun die im Rahmen dieser Studienarbeit erfolgte Implementierung der BVI-Suche vorgestellt werden. Dabei zeigt sich, daß eine Fülle von praktischen Problemen zu lösen sind, die bei einer theoretischen Betrachtung keine Rolle spielen.

6.1 Übersicht

Die Optimierung der Trennhyperebene mittels EOC oder GOC erfordert einen kompletten Durchlauf durch die Trainingsvektorenmenge in jedem inneren Knoten, um die Projektion der Voronoi-Gebiete und eventuell auch noch die Histogramme der Trainingsvektorenverteilung zu bestimmen. Bei einer sehr großen Trainingsmenge (und die braucht man), wird daher der Aufwand einer sequentiellen Immplementierung extrem hoch.
Aus diesem Grund wurden für diese Studienarbeit zwei verschiedene Implementierungen erstellt. Einmal die korrekten EOC- und GOC-Algorithmen, wie sie bisher vorgestellt wurden. Zum anderen wurde aber in einer alternativen Implementierung untersucht, ob man die sehr aufwendigen Durchläufe durch die Trainingsvektorenmenge in eine Vorverarbeitungsphase auslagern kann.
Dabei werden die Voronoi-Gebiete und Histogramme nur ein einziges Mal für das initiale Gebiet (Hyperwürfel) berechnet, und im weiteren Verlauf nur noch an das jeweilige bucket angepaßt. Dabei sind die Projektionen der Voronoi-Gebiete aber größer als eigentlich notwendig, man verschenkt also Möglichkeiten zur Optimierung. Die damit erzeugten K-d Bäume bieten eine geringere Reduktion der Suchkomplexität, als die mit dem korrekten Algorithmus erzeugten, können aber wesentlich schneller berechnet werden.
Die BVI-Suche wurde zum Zwecke der Vektorquantisierung von Fourier- transformierten Sprachdaten implementiert. Die beschleunigenden Eigenschaften des BVI-Verfahrens wurden bislang nur anhand von Merkmalsräumen der Dimension ≤ 10 gezeigt (siehe [1]). Für diese Studienarbeit wurden aber ausschließlich Merkmalsräume der Dimension 16 verwendet. Ob der BVI-Algorithmus auch in diesen relativ hochdimensionalen Räumen effektive Gewinne erzielen kann, wird in den Experimenten im nächsten Kapitel untersucht werden.
Die Grundlage für das Design von BVI-optimierten K-d Bäumen bildet ein Codebuch und eine große Anzahl an Trainingsvektoren. Es waren dazu 49 verschiedene, vorgefertigte Codebücher mit jeweils 50 Vektoren der Dimension 16 verfügbar. Die Codebücher stammen aus dem Spracherkenner des JANUS-Systems und enthalten typische Vektoren zu jeweils einem bestimmten von 49 Phonemen. Die Phoneme (und die dazu vorhandenen Codebücher) werden mit folgenden Symbolen bezeichnet:

SIL	AA	AE	AH	AO	AW	AX
AY	B	CH	D	DD	DH	DX
EH	EN	ER	EY	F	G	HH
IH	IX	IY	JH	K	KD	L
M	N	NG	OW	OY	P	PD
R	S	SH	T	TD	TS	TH
UH	UW	V	W	Y	Z	ZH

Als Trainingsvektoren stand eine sehr große Datenbasis *(resources management)* mit insgesamt fast 2 Millionen Vektoren zur Verfügung, die durch Digitalisierung und anschließende Fast-Fourier-Transformation (FFT) aus über 5000, von verschiedenen Sprechern gesprochenen Sätzen gewonnen wurden. Es wurde mit zwei Arten von Merkmalsvektoren *(streams)* gearbeitet. Zum einen mit den bereits erwähnten FFT-Koeffizienten, zum anderen aber auch mit den differentiellen FFT-Koeffizienten, so daß zu jedem Phonem zwei Codebücher existieren, die zur Unterscheidung mit einer Null bzw. einer Eins gekennzeichnet werden (z. B. SIL-0 und SIL-1). Der Wertebereich der FFT-Koeffizienten ist genormt auf das Intervall $(0, 1)$, so daß in diesem Fall das initiale bucket für den BVI-Algorithmus der Einheitshyperwürfel ist. Dahingegen können die Werte der differentiellen FFT-Koeffizienten prinzipiell Werte in $(-1, 1)$ annehmen. Erfahrungsgemäß liegen die Werte aber nur im Intervall $(-0.3, 0.3)$ da eine gewisse Stetigkeit zwischen aufeinanderfolgenden Merkmalsvektoren vorliegt.
Im folgenden wird nun auf Details der erfolgten Implementierungen eingegangen, wobei zwischen der korrekten, rechenintensiven Optimierung und der approximativen, schnellen Optimierung unterschieden wird.

6.2 Realisierung der Datenstruktur K-d Baum

Die rechnerinterne Realisierung eines K-d Baumes in Form einer problemgerechten Datenstruktur ist sowohl für die Design-Algorithmen, als auch für den Suchalgorithmus (BVI Suche) von entscheidender Bedeutung. Dabei spielen Faktoren, wie die Komplexität der Algorithmen und der Speicherplatzbedarf der Datenstruktur eine Rolle.

Abbildung 12 illustriert die in den hier vorgestellten Implementierungen verwendete Datenstruktur für K-d Bäume. Dabei werden die inneren Knoten des K-d Baumes getrennt von den Blättern, in Form eines Heap, in einem 1-dimensionalen Array gespeichert.
Um die Berechnung des linken und rechten Sohns eines Knotens effizient zu gestalten, beginnt das Array der Knoten erst mit dem Index 1 (Wurzelknoten). Der linke Sohn eines beliebigen Knotens i ist dann der Knoten mit Index $2i$ und der rechte Sohn ist der Knoten mit Index $2i + 1$. Umgekehrt kann der Vater eines Knotens durch einfache Division durch 2 ohne Rest bestimmt werden.

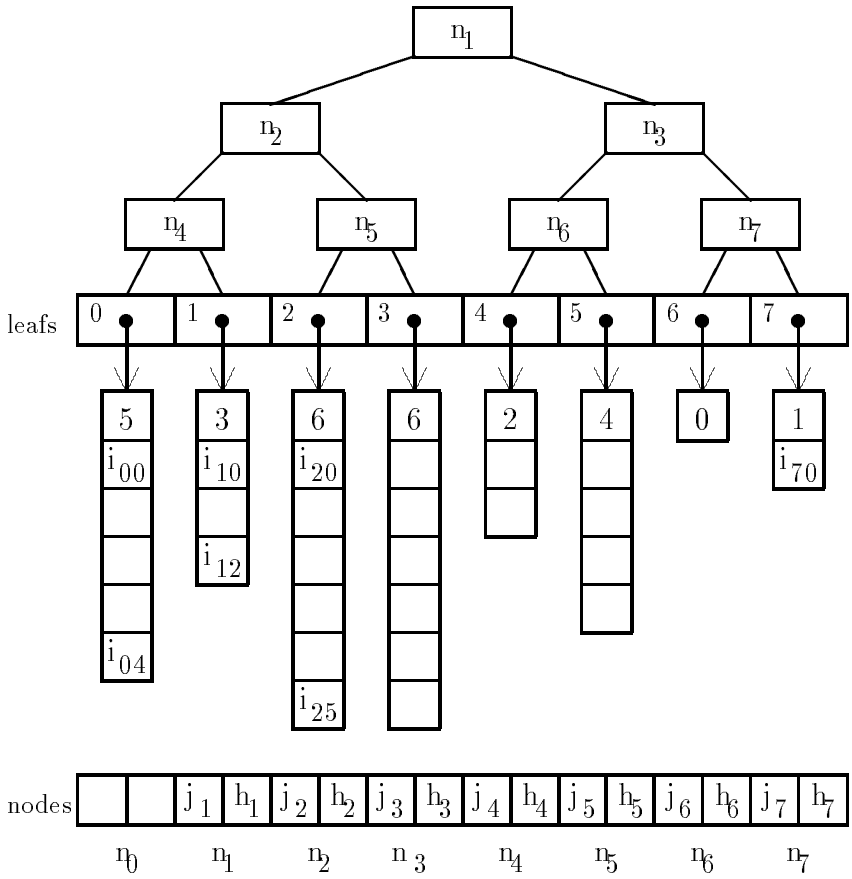

Abbildung 12: Verwendete Datenstruktur für K-d Bäume

Die Blätter sind durch ein Array von Zeigern auf die zugehörigen BVI Listen implementiert. Jede BVI Liste besteht aus $b + 1$ Elementen, wobei b die Anzahl der Codebuchvektoren ist, die mit dem jeweiligen bucket überlappen. Im ersten Element der Liste steht die Anzahl b, in den restlichen b Elementen stehen die Indizes der jeweiligen Codebuchvektoren.
Für die BVI-Suche mit dem K-d Baum werden dann noch die Codebuchvektoren explizit benötigt. Diese sind aber schon in dem Codebuch gespeichert, das als Ausgangsbasis für die Berechnung der Voronoi-Projektionen diente, und werden deshalb nicht noch einmal in der K-d Baum Datenstruktur gespeichert.

6.3 Korrekte Optimierung

Im folgenden wird die Implementierung vorgestellt, die korrekt im Sinne der vorgestellten Optimierungsverfahren GOC und EOC optimiert. Die für das BVI Verfahren benötigten Parameter werden als Kommandozeilenparameter an die Programme übergeben.

6.3.1 Spezifikation der Programm-Parameter

- **BVItree** *cbfile cbname depth optimizer trainlist trsize resN*

  cbfile Name des Files, das mehrere Codebücher enthält und von denen zu einem ein K-d Baum bestimmt werden soll.

  cbname Name des Codebuchs, zu dem ein K-d Baum bestimmt werden soll.

  depth Höhe des zu berechnenden K-d Baumes.

  optimizer Angabe des Optimierungskriteriums (EOC oder GOC).

  trainlist Name des Files, das eine Liste von Filenamen enthält, die die Trainingsvektoren enthalten.

  trsize Anzahl der Trainingsvektoren, die zur Bestimmung der Voronoi-Projektionen (und der Histgramme) quantisiert werden sollen.

  resN Auflösung der Histogramme (falls mit EOC optimiert wird).

- **BVItrees** *cbfile depth optimizer trainlist trsize resN*

  cbfile Name des Files, das die Codebücher enthält, zu denen K-d Bäume berechnet werden sollen.

  depth Höhe des zu berechnenden K-d Baumes.

  optimizer Angabe des Optimierungskriteriums (EOC oder GOC).

  trainlist Name des Files, das eine Liste von Filenamen enthält, die die Trainingsvektoren enthalten.

  trsize Anzahl der Trainingsvektoren, die zur Bestimmung der Voronoi-Projektionen (und der Histgramme) quantisiert werden sollen.

  resN Auflösung der Histogramme (falls mit EOC optimiert wird).

Die beiden Programme unterscheiden sich lediglich dadurch, daß das erste genau einen K-d Baum zu einem bestimmten Codebuch erzeugt, während das zweite zu allen vorliegenden Codebüchern K-d Bäume erstellt.

6.3.2 Implementierung des BVI Algorithmus

Die gewählte Vorgehensweise beim Erzeugen eines K-d Baumes mit den angegebenen Optimierungskriterien soll nun anhand Abbildung 13 veranschaulicht werden. Dicke Pfeile entsprechen dabei dem Kontrollfluß, während dünne Pfeile den Datenfluß symbolisieren. Ausgangsbasis der K-d Baum Erzeugung ist eine Menge von Trainingsvektoren zusammen mit einem dazugehörigen Codebuch, wie in der Abbildung oben links zu sehen.

In einem ersten, vorverarbeitenden Schritt werden nun alle Trainingsvektoren mittels einer erschöpfenden Suche auf ihren Nächsten-Nachbarn im Codebuch quantisiert und diese Codes dann zur weiteren Verwendung in einer Tabelle gespeichert. Dies macht Sinn, da die Nächster-Nachbar Codes im Verlauf des Optimierungsprozesses zur Bestimmung der Voronoi-Gebiete für jeden inneren Knoten des Baumes aufs Neue benötigt werden.
Als Nächstes wird der zu erzeugende K-d Baum initialisiert. Dazu muß der Wurzelknoten und die Randwerte des zugehörigen initialen buckets bestimmt werden. Der Baum wird nun sukzessive Ebene für Ebene von links nach rechts im Stile einer Breitensuche aufgebaut, in dem jeder Knoten lokal optimiert wird. Das Abarbeiten des Baumes bei gleichzeitiger Optimierung und Erzeugung der Knoten ist in Abbildung 13 durch die Schleife im Kontrollfluß dargestellt.

Um einen bestimmten inneren Knoten zu optimieren, das heißt eine optimale Trennhyperebene zu bestimmen, müssen zuerst einmal die Voronoi-Projektionen und gegebenenfalls auch die Histogramme der Trainingsvektorverteilungen bestimmt werden. Dies ist sehr zeitaufwendig, da man diese Informationen nicht analytisch sondern durch eine mehr oder weniger große Trainingsvektormenge approximativ bestimmt. Für jeden Trainingsvektor untersucht man, ob dieser innerhalb des zum aktuellen inneren Knoten gehörenden Hyperquaders liegt. Ist dies der Fall, so bestimmt man durch einfaches table-look-up den Nächsten Nachbarn im Codebuch und verändert gegebenenfalls dessen Voronoi-Projektionen (falls diese sich durch den hinzugekommenen Trainingsvektor überhaupt verändern).
Der Vektor-in-Hyperquader-Test, der im Falle einer Vektordimension von k eigentlich $2k$ Vergleiche benötigt, kann durch den schon erzeugten Teilbaum viel schneller erledigt werden. Beginnend mit dem Wurzelknoten verzweigt man entsprechend dem Ausgang des Hyperebenen-Tests in jedem Knoten nach links oder rechts. Nur wenn man dabei im aktuellen Knoten landet, liegt der Vektor auch im aktuellen Hyperquader. Durch einen Lauf durch die gesamte Trainingsvektormenge erhält man durch das beschriebene Verfahren die (approximativen) Voronoi-Projektionen der Codebuchvektoren und eventuell (im Falle des EOC Kriteriums) auch noch die Histogramme der Trainingsvektorverteilung.
Dabei tritt jedoch ein Problem auf. Mit fortschreitender Tiefe des K-d Baumes werden die zu den Knoten gehörenden Hyperquader immer kleiner. Aus der zwar sehr großen Menge an Trainingsvektoren liegen dabei immer weniger Vektoren noch innerhalb des aktuellen Hyperquaders, so daß die daraus gewonnenen Voronoi-Projektionen und Histogramme mit fortschreitender Tiefe immer ungenauer werden. Irgendwann liegen sogar gar keine Trainingsvektoren mehr in einigen Hyperquadern, so daß man eigentlich gar keine Aussage mehr über die Voronoi- Projektionen machen kann.
Dieser Fall wird in der vorliegenden Implementierung dadurch abgefangen, daß im Falle eines "leeren" Hyperquaders einfach zusätzliche Trainingsvektoren zufällig erzeugt werden, die innerhalb des Hyperquaders liegen und damit eine Bestim-

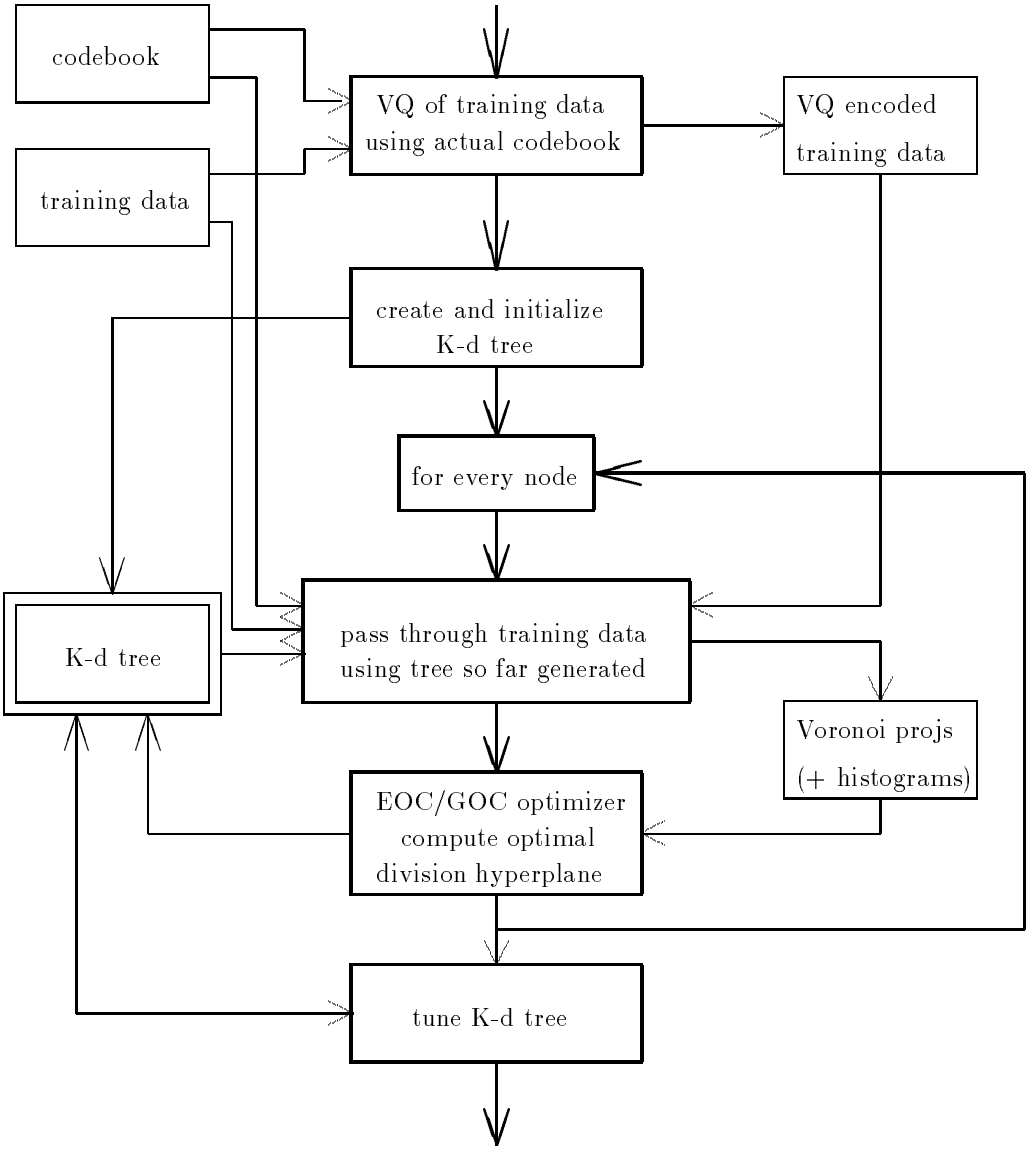

Abbildung 13: Blockdiagramm zum K-d tree Designer BVItree

mung der Voronoi-Projektionen erlauben. Allerdings schleicht sich durch diesen Trick ein Fehler ein: Die Trainingsvektorverteilung wird durch die künstlichen Trainingsvektoren etwas verfälscht, was aber nur im Falle des EOC Kriteriums kritisch werden kann.
Sind die Voronoi-Projektionen und die Histogramme der Trainingsvektorverteilung bestimmt, wird im nächsten Schritt entweder durch den GOC- oder den EOC- Optimierer eine Trennhyperebene bestimmt und deren Parameter im aktuellen Knoten des K-d Baumes gespeichert. Die Bestimmung optimaler Trennhyperebenen wurde bereits in den vorangegangenen Kapiteln ausführlich besprochen und soll daher hier nicht näher erläutert werden.

Nach Optimierung aller Knoten bis zur gewünschten Tiefe d müssen nun nur noch die BVI Listen in den Blättern des K-d Baumes erzeugt werden. Dazu werden für jedes Blatt alle Codebuchvektoren darauf getestet, ob sie innerhalb des zugehörigen Hyperquaders (buckets) liegen. Ist dies der Fall wird der Index des Codebuchvektors in die BVI Liste des jeweiligen Blattes aufgenommen.
Prinzipiell ist der Nächster-Nachbar Suchbaum jetzt fertig und könnte schon zur schnellen Vektorquantisierung benutzt werden. Durch einen geringen zusätzlichen Aufwand verschafft man sich aber eine noch bessere Performanz des Baumes.
In einem zusätzlichen Durchlauf durch die Trainingsvektorenmenge bestimmt man für jede BVI Liste (jedes Blatt) getrennt die Trainingsvektoren, die innerhalb des zugehörigen buckets liegen. Mit Hilfe dieser Trainingsvektoren bestimmt man nun zu jedem in der BVI Liste vorkommenden Codebuchvektor die Wahrscheinlichkeit dafür, daß er Nächster-Nachbar eines Vektors ist, unter der Vorraussetzung, daß dieser Vektor innerhalb des aktuellen buckets liegt.
Durch einfaches Umsortieren der Indizes der Codebuchvektoren in den BVI Listen nach absteigender Wahrscheinlichkeit erzielt man bei der späteren sequentiellen Nächster-Nachbar Suche innerhalb der BVI Listen eine Beschleunigung, da die wahrscheinlicheren Codebuchvektoren zuerst getestet werden. Vorraussetzung ist aber die Verwendung des Partial Distance Search Algorithmus zur Nächster-Nachbar Suche. Diese Umordnung der BVI Listen wird in dieser Studienarbeit als *tuning* bezeichnet.

6.4 Approximative Optimierung durch Vorberechnung

Aufgrund des sehr hohen Aufwandes der korrekten Optimierung wurde in einer zweiten Implementierung die Berechnung der Voronoi-Projektionen und der Histogramme der Trainingsvektorverteilung nur ein einziges mal in einer Vorberechnungsphase durchgeführt und dann zur Optimierung des gesamten Baumes verwendet. Die damit erzielbare Komplexitätsreduktion ist erwartungsgemäß geringer als die mit dem korrekten Verfahren erreichbare. Dafür erhält man jedoch einen schnelleren Algorithmus zur Erzeugung von suboptimalen Nächster-Nachbar Suchbäumen, die eventuell sogar zur Beschleunigung des Trainings eines Spracherkenners eingesetzt werden können, wo sich die Codebuchvektoren in jeder Trainingsepoche verändern, so daß man jeweils neue Suchbäume berechnen muß, was nur dann Vorteile bei der Nächster-Nachbar-Suche bringt, wenn diese Suchbäume relativ schnell berechnet werden können.
Im folgenden werden nun die entstandenen Programme und deren Parameter kurz vorgestellt und erläutert.

6.4.1 Vorberechnung

- **vtrain** *cbfile trsize*

cbname Name des Codebuchs, zu dem ein K-d Baum bestimmt werden soll.

cbfile Name des Files, das die Codebücher enthält, zu denen Voronoi-Projektionen bestimmt werden sollen.

trsize Anzahl der Trainingsvektoren, die zur Bestimmung der Voronoi-Projektionen quantisiert werden sollen.

- **vp_view** *vpname cbname*

vpname Name des von vtrain erzeugten Files, das die Voronoi- Projektionsintervalle zu einer Menge von Codebüchern enthält.

cbname Name des Codebuchs aus dem File *vpname*, dessen Voronoi-Projektionen visualisiert werden sollen. Die Namen der Codebücher entsprechen den Symbolen für die Phoneme (s. o.).

Das zur Vorberechnungsphase zählende Programm **vtrain** bestimmt zu jedem der Codebücher aus *cbfile* näherungsweise die Projektionen der Voronoi-Gebiete seiner Codebuchvektoren auf die $K = 16$ Achsen. Dazu werden *trsize* Trainingsvektoren mittels Partial Distance Search auf ihren Nächsten-Nachbarn im jeweiligen Codebuch quantisiert. Zu jedem der N Codebuchvektoren $\mathbf{c_i}$ entsteht dabei eine Menge T_i von Trainingsvektoren, die näher an $\mathbf{c_i}$, als an allen anderen Codebuchvektoren liegen. Nun wird näherungsweise die Projektion des Voronoi-Gebietes V_i des Codebuchvektors $\mathbf{c_i}$ auf eine Achse k bestimmt, indem das Minimum und Maximum der k-ten Komponente der Vektoren aus T_i gebildet wird:

$$P^k_{i,L} = \min t_k : \mathbf{t} \in T_i \quad , \qquad P^k_{i,U} = \max t_k : \mathbf{t} \in T_i$$

Das Intervall $(P^k_{i,L}, P^k_{i,U})$ ist dann eine Näherung an die Projektion des Voronoi-Gebietes des Codebuchvektors $\mathbf{c_i}$ auf die Achse k. Je größer die Zahl der Trainingsvektoren *trsize* gewählt wird, desto exakter können die Voronoi-Projektionen bestimmt werden. Abbildung 14 veranschaulicht dies im $\mathcal{R}^2$.

Das vtrain Programm erzeugt als Ausgabe ein File, das die Unter- und Obergrenzen der Projektionsintervalle aller Codebuchvektoren aller Codebücher enthält. Dieses File bildet die Grundlage der GOC- und EOC-Optimierung von K-d Bäumen.

Das Programm **vp_view** stellt die projezierten Voronoi-Gebiete zu einem bestimmten Codebuch als Intervalle dar, und dient dazu die Auswirkungen der Größe der verwendeten Trainingsvektorenmenge anschaulich interpretieren zu können.

- **htrain** *resolution trsize*

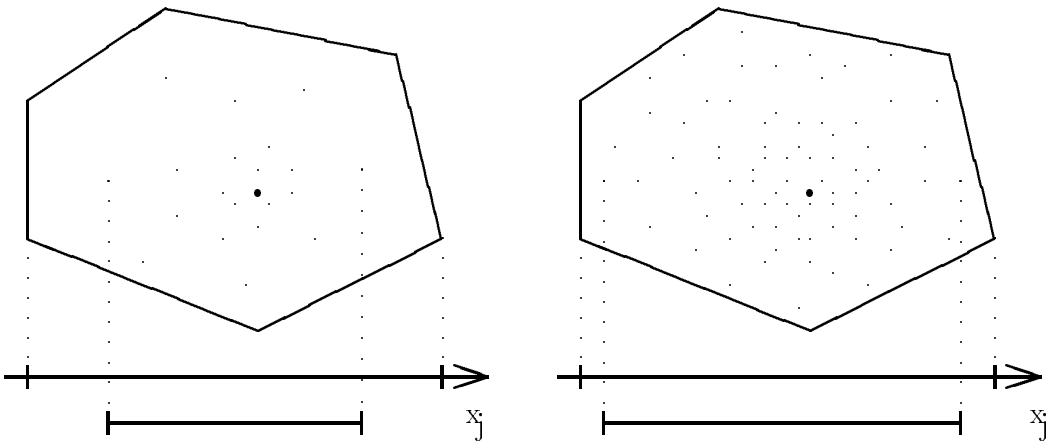

Abbildung 14: Approximation der Projektion des Voronoi-Gebietes eines Codebuchvektors mit wenigen (links) und vielen (rechts) Trainingsvektoren

resolution Auflösung des diskreten Rasters, mit dem die Histogramme der Trainingsdatenverteilungen erstellt werden sollen.

trsize Anzahl der Trainingsvektoren, die zur Bestimmung der Trainingsdatenverteilungen herangezogen werden sollen.

- **h_view** *histfile*

  histfile Name des von htrain erzeugten Files, das die zu visualisierenden Histogramme enthält.

Das Programm **htrain** erzeugt für jede der $K = 16$ Dimensionen ein Histogramm das eine diskrete Näherung an die Verteilungsfunktion der Trainingsvektoren in der jeweiligen Koordinate darstellt. Dazu wird in jeder Koordinate das initiale Intervall $(0, 1)$ mit der Auflösung *resolution* gerastert. Hier gilt genauso wie bei vtrain, daß die Approximation der realen Verteilungsfunktion um so besser wird, je mehr Trainingsvektoren zur Bestimmung herangezogen werden. Abbildung 14 illustriert dies im $\mathcal{R}^2$ wobei nur das Histogramm für eine Koordinate dargestellt ist (nach unten). Links ist eine sehr schlechte Approximation mit einer Auflösung von 9 dargestellt, wobei nur wenige Trainingsvektoren verwendet wurden. Das Histogramm auf der rechten Seite wurde mit der doppelten Auflösung und unter Verwendung von wesentlich mehr Trainingsvektoren erzeugt, was zu einem deutlich besseren Ergebnis führt.

Die erzeugten Histogramme werden zum Design EOC-optimierter K-d Bäume benötigt. Wie bereits erwähnt wurde, genügt dazu ein einziger Aufruf von htrain, das die Histogramme in einem File zur Verfügung stellt. Dies entspricht einem einzigen Durchlauf durch die Trainingsvektorenmenge. Dabei ist jedoch zu beachten, daß die Auflösung der Histogramme groß genug gewählt wird (*resolution*

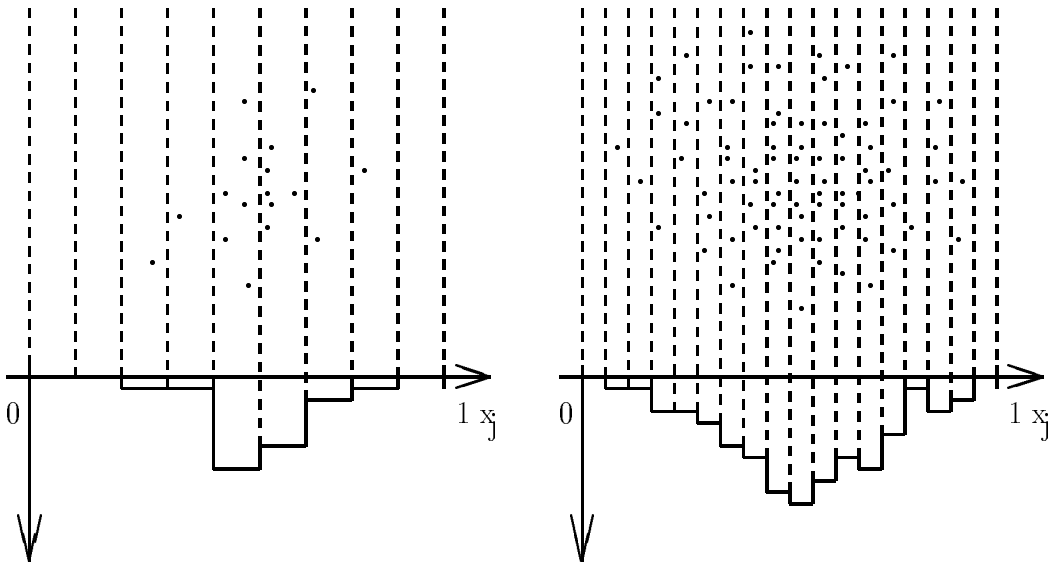

Abbildung 15: Qualitative Unterschiede in der Approximation der Trainingsdatenverteilung durch zwei verschiedene Histogramme

≥ 1000). Mehr dazu in dem Kapitel über die Wahl der Parameter bei der EOC-Optimierung.

Das Programm **h_view** visualisiert die von htrain erzeugten Histogramme am Bildschirm, was natürlich nur mit einer reduzierten Auflösung möglich ist. Das Programm ist, ebenso wie vp_view, nur als Tool gedacht, um die Ergebnisse von vtrain und htrain anschaulich interpretieren zu können.

6.4.2 Designphase

Zum Erzeugen von K-d Bäumen stehen die folgenden 4 Programme zur Verfügung, jeweils 2 für das Optimieren mit GOC und 2 für EOC. Da sich die approximative von der korrekten Optimierung nur dadurch unterscheidet, daß sie die Voronoi-Projektionen und die Histogramme der Trainingsvektorverteilung einmal im Voraus bestimmt, wird hier auf eine detailierte Erläuterung der Programme verzichtet.

- **goc_tree** *vpname cbname maxdepth breakval*

  vpname Name des Files, das mehrere Codebücher enthält, unter anderem auch das für diese Routine verwendete.

  cbname Name des Codebuchs aus dem File *vpname*, zu dem ein K-d Baum berechnet werden soll.

maxdepth Maximale Tiefe des Designs des K-d Baumes. Wählt man für *breakval* den Wert 0.0 so ist gewährleistet, daß die maximale Tiefe auf jeden Fall erreicht wird.

breakval Dieser Wert dient zur adaptiven Kontrolle des Optimierungsverfahrens. Falls die Abnahme der mittleren bucket-Größe von einem Niveau zum Nächsten kleiner als der Wert *breakval* wird, bricht der Vorgang ab und der bis dahin berechnete Baum wird ausgegeben.

- **goc_trees** *vpname maxdepth breakval*

  vpname Name des Files, das die Codebücher enthält, für die K-d Bäume berechnet werden sollen.

  maxdepth Maximale Tiefe des Designs des K-d Baumes. Wählt man für *breakval* den Wert 0.0 so ist gewährleistet, daß die maximale Tiefe auf jeden Fall erreicht wird.

  breakval Dieser Wert dient zur adaptiven Kontrolle des Optimierungsverfahrens. Falls die Abnahme der mittleren bucket-Größe von einem Niveau zum Nächsten kleiner als der Wert *breakval* wird, bricht der Vorgang ab und der bis dahin berechnete Baum wird ausgegeben.

- **eoc_tree** *vpname hname cbname maxdepth breakval*

  vpname Name des Files, das mehrere Codebücher enthält, unter anderem auch das für diese Routine verwendete.

  hname Name des Files, das Histogramme für alle Koordinatenachsen enthält (kann mit htrain berechnet werden).

  cbname Name des Codebuchs aus dem File *vpname*, zu dem ein K-d Baum berechnet werden soll.

  maxdepth Maximale Tiefe des Designs des K-d Baumes. Wählt man für *breakval* den Wert 0.0 so ist gewährleistet, daß die maximale Tiefe auf jeden Fall erreicht wird.

  breakval Dieser Wert dient zur adaptiven Kontrolle des Optimierungsverfahrens. Falls die Abnahme der mittleren bucket-Größe von einem Niveau zum Nächsten kleiner als der Wert *breakval* wird, bricht der Vorgang ab und der bis dahin berechnete Baum wird ausgegeben.

- **eoc_trees** *vpname hname maxdepth breakval*

  vpname Name des Files, das die Codebücher enthält, zu denen K-d Bäume berechnet werden sollen.

  hname Name des Files, das Histogramme für alle Koordinatenachsen enthält (kann mit htrain berechnet werden).

maxdepth Maximale Tiefe des Designs des K-d Baumes. Wählt man für *breakval* den Wert 0.0 so ist gewährleistet, daß die maximale Tiefe auf jeden Fall erreicht wird.

breakval Dieser Wert dient zur adaptiven Kontrolle des Optimierungsverfahrens. Falls die Abnahme der mittleren bucket-Größe von einem Niveau zum Nächsten kleiner als der Wert *breakval* wird, bricht der Vorgang ab und der bis dahin berechnete Baum wird ausgegeben.

6.4.3 Wahl der Parameter bei der EOC-Optimierung

Während des Designs eines K-d Baumes werden ja die zu den inneren Knoten gehörenden Hyperquader mit steigender Tiefe immer kleiner. Dadurch werden auch die bei der Optimierung betrachteten Intervalle (a_j, b_j) auf den Achsen j immer kleiner, so daß eine korrekte Berechnung der Wahrscheinlichkeiten $p_L(h)$ und $p_R(h)$ für jeden Knoten separat einen Durchlauf durch die Trainingsvektorenmenge erfordert.
In der vorliegenden Implementierung wurde jedoch aus Effizienzgründen auf die korrekte Berechnung der Wahrscheinlichkeiten verzichtet. Statt dessen werden während der gesamten Design-Phase des K-d Baumes mit EOC die gleichen (vorberechneten) Histogramme verwendet. Dabei kann es passieren, daß die Länge der bei der Optimierung betrachteten Intervalle die Größenordnung des Auflösungsintervalls der Histogramme annehmen. In diesem Fall muß man lokal zum GOC-Kriterium übergehen, da die Fehler bei der approximativen Berechnung der Wahrscheinlichkeiten zu groß werden.
Damit stellt die vorliegende Implementierung einen Kompromiß zwischen exakter Optimierung und Effizienz der Berechnungen dar, sozusagen eine Mischung aus EOC und GOC, wobei man empirisch eine geschickte Wahl der Auflösung der Histogramme bestimmen muß.
Bei einer geringen Auflösung der Histogramme optimiert das EOC-Programm eher mit GOC als mit EOC, ist aber sehr schnell. Bei einer sehr hohen Auflösung konvergiert das EOC-Programm immer mehr zu einm reinen EOC, die Resultate werden besser, aber die Laufzeit steigt exponentiell an.

6.5 Vektorquantisierung durch BVI-Suche

Zur Analyse der schnellen Nächster-Nachbar Vektorquantisierung mittels vom BVI Algorithmus erzeugten K-d Bäumen wurde folgendes Programm implementiert, welches eine vorgegebene Anzahl von Testvektoren sowohl konventionell durch erschöpfende Suche unter allen Codebuchvektoren, als auch durch schnelle Nächster-Nachbar Suche mit dem vorberechneten K-d Baum quantisiert und dabei verschiedene Messungen vornimmt.
Dabei ist besonders zu beachten, daß die Testvektorenmenge verschieden von der

Trainingsvektorenmenge gewählt wird, die zum Design des K-d Baumes verwendet wurde. Nur auf diese Weise kann man die Fehlerrate der BVI Quantisierung messen (auf den Trainingsdaten quantisiert das BVI Verfahren fehlerfrei).

- **BVIsearch** *cbfile codebook Kdtree testlist testsize*

  cbfile Name des Files, das mehrere Codebücher enthält, unter anderem auch das Codebuch zum verwendeten K-d Baum.

  codebook Name des Codebuchs, das verwendet werden soll.

  Kdtree Name des Files, das einen (oder auch mehrere hintereinander) K-d Baum enthält, der zur Nächster-Nachbar-Suche benutzt werden soll.

  testlist Name des Files, das eine Liste von Filenamen enthält, aus denen die Testvektoren entnommen werden.

  testsize Anzahl an Testvektoren, die zum Zwecke der Analyse sowohl konventionell, als auch mit BVI-Suche quantisiert werden sollen.

Die von BVIsearch während eines Durchlaufs durch die Testvektorenmenge durchgeführten Messungen werden in Form einer Tabelle ausgegeben. Konkret erhält man Aufschluß über folgende Werte:

- Anzahl der durchsuchten Codebuchvektoren um den Nächsten-Nachbarn zu finden. Dies entspricht der mittleren Anzahl an Codebuchvektoren in den BVI-Listen an den Blättern des K-d Baums.

- Speed Up des BVI Algorithmus gegenüber dem konventionellen Verfahren. Dieser Wert wird direkt durch einen Vergleich der benötigten Zeit mittels elementarer Zeitmessungsroutinen bestimmt.

- Fehlerrate der BVI Suche. Dies ist der Prozentsatz der Testvektoren, die von dem Verfahren nicht auf den korrekten (im Sinne eines minimalen euklidischen Abstandes) Nächsten-Nachbarn quantisiert werden konnten.

- Signal-to-Noise ratio. Vergleich der SNR der beiden Quantisierungsverfahren. Die SNR gibt an, wieweit sich der Signalpegel vom Rauschen abhebt und wird in Dezibel gemessen. Die SNR ist wie die Fehlerrate ein Maß für die Quantisierungsgüte der Suchbäume. Sie ist jedoch aussagekräftiger da sie das unvermeidbare Rauschen jedes Quantisierers berücksichtigt und die Fehlerrate in Beziehung zum Fehlerabstand (Euklidische Distanz) setzt. Die SNR ist wie folgt definiert:

  $$SNR = 10 \log_{10} \frac{E(X^2)}{E((X - Q(X))^2)} \quad (db)$$

  Dies ist das logarithmierte Verhältnis zwischen der erwarteten Leistung des Signals und des mittleren quadratischen Quantisierungsfehlers.

- Histogramm der Verteilung der von der BVI Suche falsch quantisierten Testvektoren. Dieses Histogramm wird folgendermaßen erstellt. Falls ein Testvektor statt auf den korrekten Nächsten-Nachbarn $\mathbf{c_i}$ auf einen anderen Codebuchvektor $\mathbf{c_j}$ quantisiert wird, stellt man fest der wievielt-nächste Codebuchvektor $\mathbf{c_j}$ ist. Dies geschieht durch Sortieren der Abstände aller Codebuchvektoren zum Testvektor. Die Plazierung des Codebuchvektors $\mathbf{c_j}$ ist dann ein relatives Maß für den Fehler. Im Histogramm wird die Verteilung der Plazierungen der "falschen" Codebuchvektoren angegeben.

Abschließend soll hier noch beispielhaft eine Ausgabe von BVIsearch gezeigt werden, wobei allerdings das Histogramm der Übersichtlichkeit wegen nicht mit angegeben ist. Dargestellt ist das Resultat der Quantisierung von 100000 Testvektoren in 10000-er Schritten mittels eines Codebuchs zum Phonem TS-0 und einem dazu durch GOC-Optimierung erzeugten K-d Suchbaums der Tiefe 10. Der Baum wurde unter Verwendung von 100000 Trainingsvektoren generiert.
$t1$ und $n1$ sind Zeitdauer und Anzahl durchsuchter Codebuchvektoren beim Quantisieren durch erschöpfende Suche. $t2$ und $n2$ sind Zeitdauer und Anzahl durchsuchter Codebuchvektoren beim BVI Verfahren. $t\star = t1/t2$ und $n\star = n1/n2$ geben den Speed Up des BVI Verfahrens an. err ist die Fehlerrate in Prozent.

```
  ============================================================
    BVISEARCH         Fast Nearest Neighbor VQ with K-d trees
  ============================================================

  - reading codebook ´TS-0´ from ´cb+1´
  - reading test vector set (100000 vectors)
  - reading K-d tree ´TS-0´ from file ´TS-0.100k.10.goc´
  - bucket sizes (min,ave,max) : ( 1 , 3.394531 , 8 )

#vecs  |   t1     n1   |   t2     n2   |   t*     n*    |  err
-------------------------------------------------------------------
 10000     4.88  50.00    1.05   5.93     4.626  8.429    2.590
 20000     9.95  50.00    2.14   6.03     4.639  8.298    2.665
 30000    15.19  50.00    3.26   6.06     4.663  8.246    2.723
 40000    20.35  50.00    4.33   6.03     4.698  8.287    2.815
 50000    25.43  50.00    5.40   6.01     4.711  8.314    2.788
 60000    30.22  50.00    6.41   5.98     4.712  8.354    2.695
 70000    35.15  50.00    7.48   6.01     4.702  8.319    2.589
 80000    40.24  50.00    8.59   6.06     4.685  8.251    2.506
 90000    45.24  50.00    9.65   6.03     4.687  8.288    2.522
100000    50.18  50.00   10.72   6.01     4.681  8.318    2.536

Total number of encoded vectors : 100000
Encoding with Full Search   SNR =   12.45947 dB
Encoding with BVI  Search   SNR =   12.43831 dB
Difference Full-BVI         SNR =    0.02116 dB
```

7 Experimente und Resultate

7.1 Übersicht

In den Experimenten wurde untersucht, wie stark sich die Komplexität der Nächster-Nachbar Suche durch den Einsatz von BVI-optimierten K-d Bäumen reduzieren läßt und wie stark sich dabei Quantisierungsfehler bemerkbar machen. Dazu stand eine große Datenbasis an aus Sprachsamples gewonnenen FFT-Vektoren der Dimension 16 zur Verfügung, die auch für den Spracherkenner JANUS-2 verwendet werden *(resources management task)*.
Zu jedem der dabei modellierten 49 Phoneme stand ein Codebuch mit 50 Prototypen als Ausgangsbasis des BVI-Algorithmus zur Verfügung. Es standen zwar auch Codebücher mit Delta Koeffizienten zur Verfügung, diese wurden aber in diesen Experimenten nicht verwendet.
Mit beiden bereits vorgestellten Implementierungen des BVI-Algorithmus wurden umfangreiche Tests durchgeführt und die Ergebnisse analysiert und miteinander verglichen.

7.2 Korrekte Optimierung

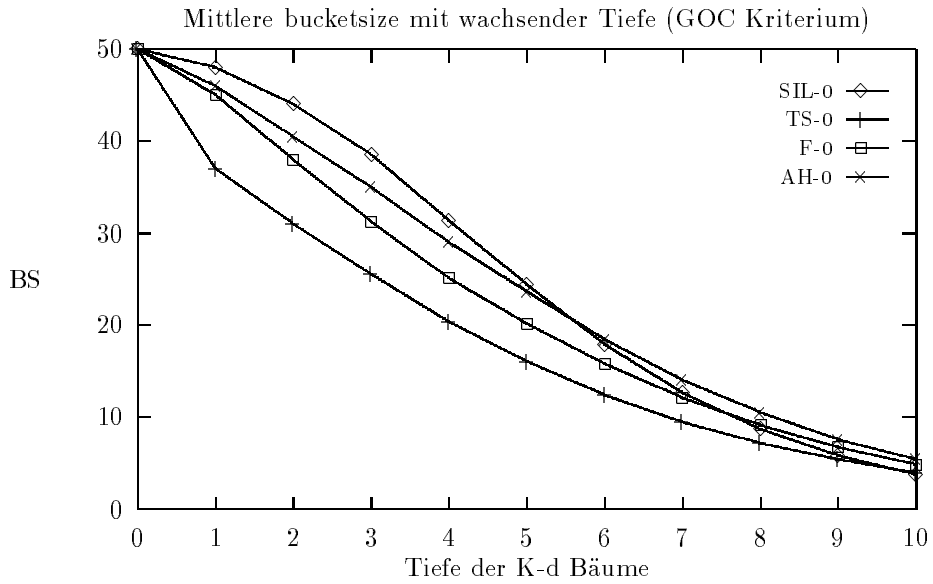

Abbildung 16: Verlauf der mittleren bucketsize bei K-d Bäumen verschiedener Tiefe (dargestellt sind die Kurven zu 4 Codebüchern)

Die sehr aufwendige korrekte Optimierung von K-d Bäumen lieferte K-d Bäume, mit denen sich die Komplexität der Nächster-Nachbar Suche am stärksten reduzieren ließ. Ein erstes Maß für den bei K-d Bäumen zu erwartenden Suchaufwand ist die mittlere bucketsize. Dies ist die mittlere Anzahl an Codebuchvektoren, die

sich in einem bucket des K-d Baumes befinden und die noch als Kandidaten für den Nächsten-Nachbarn in Frage kommen.
Abbildung 16 zeigt an insgesamt 4 Codebüchern wie die mittlere bucketsize von der Tiefe des benutzten Baumes abhängt. Man erkennt, daß sich durch Verwendung eines Baumes größerer Tiefe stets noch eine weitere Dezimierung der mittleren bucketsize erreichen läßt, da sie eine streng monoton fallende Funktion der Baumtiefe ist.
Abhängig von der Verteilung der Codebuchvektoren eines Codebuchs im 16-dimensionalen Raum ergeben sich, wie man sieht, unterschiedliche Verläufe der mittleren bucketsize. Besonders auffallend ist dabei der Verlauf der Kurve für das Codebuch SIL-0, das Prototypen für die Stille in den Aussprachepausen enthält. Im Gegensatz zu den Prototypen der einzelnen Phoneme variieren diese Codebuchvektoren kaum; sie bilden einen Cluster von Vektoren. Die hier verwendeten Codebücher bestehen alle aus konstant 50 Codebuchvektoren, obwohl man die Anzahl der Codebuchvektoren eigentlich an die Verteilung der zu modellierenden Merkmalsvektoren anpassen sollte. Auf den besonderen Verlauf der Kurve des Codebuchs SIL-0 wird später noch näher eingegangen.

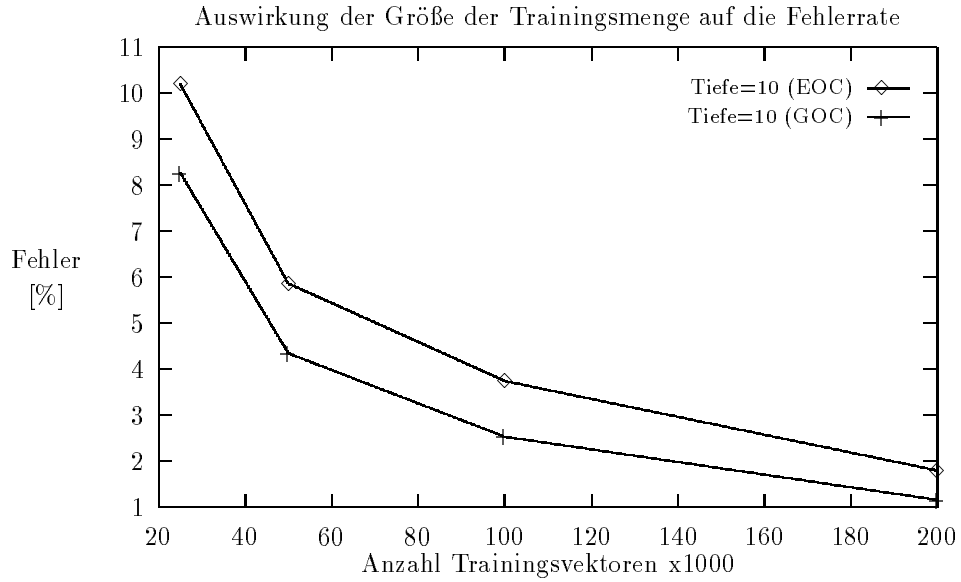

Abbildung 17: "There is no data like more data !"

Abbildung 17 zeigt die Auswirkungen der Anzahl der zur Berechnung der Voronoi-Projektionen verwendeten Trainingsvektoren auf die Fehlerrate bei der Vektorquantisierung mit den erzeugten K-d Bäumen. Die Fehlerrate ist der Prozentsatz an Testvektoren, die durch das BVI Verfahren nicht auf den korrekten Nächsten-Nachbarn quantisiert werden konnten. Die Bäume wurden mit einer Tiefe von 10 erzeugt und mit 100000 Testvektoren getestet, die nicht beim Optimieren als Trainingsvektoren verwendet wurden. Erwarungsgemäß sinkt die Feh-

lerrate mit wachsender Zahl an Trainingsvektoren ab. Man kann an den Kurven der Abbildung 17 aber auch erkennen, daß die mit dem EOC-Kriterium erzeugten Bäume bei gleicher Zahl an verwendeten Trainingsvektoren stets eine höhere Fehlerrate verursachen, als die mit dem GOC-Kriterium erzeugten. Das EOC Kriterium benötigt also eine größere Anzahl an Trainingsvektoren zur Erzielung der gleichen Fehlerrate. Dies liegt daran, daß sich beim EOC-Kriterium zusätzliche Fehler durch die nur näherungsweise bestimmbare Verteilung der Vektoren in Form von Histogrammen ergeben.
Die Fehlerrate läßt sich also durch Hinzunahme weiterer Trainingsvektoren immer weiter senken. Dem sind aber in der Praxis aus zwei Gründen Grenzen gesetzt:

- Erstens hat man nicht beliebig viele Trainingsvektoren zur Verfügung.
- Zweitens steigt der Aufwand zur Erzeugung der Suchbäume linear mit der verwendeten Anzahl an Trainingsvektoren an.

Als nächstes wurden zu allen verfügbaren 49 Codebüchern mit dem GOC Optimierungskriterium und unter Verwendung von 200000 Trainingsvektoren K-d Suchbäume der Tiefe 10 erzeugt und deren Leistung beim Quantisieren von 100000 Testvektoren verglichen. Ein besseres Maß für die mit K-d Bäumen erzielbare Beschleunigung in der Nächster-Nachbar Suche als die mittlere bucketsize ist der Speed-Up. Er wird durch Division der benötigten Zeit für eine Quantisierung mit erschöpfender Suche durch die benötigte Zeit der BVI Suche berechnet. Auf den nächsten beiden Seiten sind zwei Tabellen mit den gemessenen Werten abgedruckt, die sich nur in der unterschiedlichen Sortierung nach Speed-Up und Fehlerrate unterscheiden. Interessant ist, daß ein hoher Speed-Up nicht unbedingt eine hohe Fehlerrate nach sich zieht. Beispiele hierfür sind die Codebücher TS-0, S-0 und CH-0. Andererseits gibt es auch Fälle, wo man nur einen vergleichsweise kleinen Speed-Up erzielt und zudem noch die Fehlerrate relativ groß ist, wie man beispielsweise am Codebuch UH-0 sieht.
Bemerkenswert ist außerdem, daß die K-d Bäume zu den Vokalen (AE-0, IH-0, EY-0, EH-0, UH-0, u. s. w.) die höchsten Fehlerraten liefern. Dies läßt vermuten, daß in diesen Fällen die Verwendung von Voronoi-Projektionen (Hyperquader) die wirklichen Voronoi-Gebiete am schlechtesten approximiert.
Als Sonderfall zeigt sich auch hier das Codebuch SIL-0, dessen K-d Baum den höchsten Speed-Up liefert. Der Grund hierfür ist wie bereits erwähnt die Konzentration der Codebuchvektoren auf ein vergleichsweise kleines Gebiet, so daß man durch das BVI Verfahren außerhalb dieses Gebiets meistens nur einen Kandidatenvektor und somit unmittelbar den Nächsten-Nachbarn erhält. Mit anderen Worten, die Voronoi-Gebiete der Codebuchvektoren belegen entweder sehr große Gebiete des Merkmalsraums und sind daher sehr gut durch Hyperquader anzunähern, oder sie sind so klein, daß eine Näherung einen vergleichsweise kleinen Beitrag zum Gesamtfehler erbringt.

Codebuch	mittl. Bucketsize	Speed-up*	Fehlerrate	SNR Differenz
SIL-0	7.50	5.591	1.382	0.00033
S-0	7.89	4.732	1.548	0.01075
OY-0	8.95	4.389	2.625	0.00353
CH-0	6.67	4.200	1.402	0.01008
TS-0	7.35	4.152	1.153	0.00843
JH-0	7.45	4.108	1.451	0.00831
AY-0	10.47	4.053	2.462	0.01003
T-0	7.18	3.998	1.747	0.03671
AO-0	10.81	3.970	2.230	0.00859
EY-0	11.59	3.879	3.067	0.01477
OW-0	10.99	3.829	2.477	0.01018
SH-0	8.91	3.822	1.207	0.00486
Z-0	9.00	3.770	1.703	0.01029
ER-0	9.38	3.679	1.987	0.01303
IY-0	10.79	3.667	2.832	0.09104
AW-0	9.22	3.655	2.753	0.01652
L-0	10.18	3.645	2.407	0.01468
Y-0	9.96	3.564	2.010	0.01357
TD-0	8.83	3.553	1.351	0.01563
AH-0	11.49	3.535	2.610	0.01127
IX-0	10.53	3.500	2.495	0.02289
AE-0	14.05	3.451	3.171	0.01151
ZH-0	9.36	3.449	1.804	0.01128
AA-0	10.77	3.419	2.344	0.01033
DX-0	11.04	3.409	2.448	0.02045
R-0	10.75	3.407	1.968	0.01358
TH-0	8.87	3.322	1.665	0.01029
UW-0	13.12	3.283	2.074	0.01209
F-0	9.00	3.279	1.766	0.01143
D-0	8.12	3.254	1.601	0.01495
IH-0	11.27	3.247	3.072	0.02600
EH-0	13.62	3.198	3.017	0.01924
K-0	9.59	3.177	1.463	0.01163
PD-0	9.02	3.137	1.310	0.01037
W-0	10.26	3.125	1.501	0.01166
M-0	12.99	3.090	1.909	0.01211
AX-0	11.47	3.057	2.464	0.01848
DH-0	9.76	3.039	2.672	0.02402
NG-0	13.13	3.025	2.421	0.01420
KD-0	10.68	2.989	1.333	0.00934
HH-0	10.64	2.983	1.808	0.01439
UH-0	14.81	2.921	2.960	0.15781
G-0	11.18	2.919	1.901	0.04936
N-0	13.43	2.836	2.199	0.01117
B-0	11.20	2.836	1.354	0.00996
P-0	11.48	2.706	1.342	0.00870
DD-0	12.26	2.704	1.684	0.01507
V-0	13.17	2.690	2.197	0.01481
EN-0	13.30	2.659	2.206	0.01613

Abbildung 18: Tabelle aller verwendeten Codebücher eines streams (FFT-Koeffizienten), nach erreichtem Speed-Up sortiert

Codebuch	mittl. Bucketsize	Speed-Up	Fehlerrate*	SNR Differenz
AE-0	14.05	3.451	3.171	0.01151
IH-0	11.27	3.247	3.072	0.02600
EY-0	11.59	3.879	3.067	0.01477
EH-0	13.62	3.198	3.017	0.01924
UH-0	14.81	2.921	2.960	0.15781
IY-0	10.79	3.667	2.832	0.09104
AW-0	9.22	3.655	2.753	0.01652
DH-0	9.76	3.039	2.672	0.02402
OY-0	8.95	4.389	2.625	0.00355
AH-0	11.49	3.535	2.610	0.01127
IX-0	10.53	3.500	2.495	0.02289
OW-0	10.99	3.829	2.477	0.01018
AX-0	11.47	3.057	2.464	0.01848
AY-0	10.47	4.053	2.462	0.01003
DX-0	11.04	3.409	2.448	0.02045
NG-0	13.13	3.025	2.421	0.01420
L-0	10.18	3.645	2.407	0.01468
AA-0	10.77	3.419	2.344	0.01033
AO-0	10.81	3.970	2.230	0.00859
EN-0	13.30	2.659	2.206	0.01613
N-0	13.43	2.836	2.199	0.01117
V-0	13.17	2.690	2.197	0.01481
UW-0	13.12	3.283	2.074	0.01209
Y-0	9.96	3.564	2.010	0.01357
ER-0	9.38	3.679	1.987	0.01303
R-0	10.75	3.407	1.968	0.01358
M-0	12.99	3.090	1.909	0.01211
G-0	11.18	2.919	1.901	0.04936
HH-0	10.64	2.983	1.808	0.01439
ZH-0	9.36	3.449	1.804	0.01128
F-0	9.00	3.279	1.766	0.01143
T-0	7.18	3.998	1.747	0.03671
Z-0	9.00	3.770	1.703	0.01029
DD-0	12.26	2.704	1.684	0.01507
TH-0	8.87	3.322	1.665	0.01029
D-0	8.12	3.254	1.601	0.01495
S-0	7.89	4.732	1.548	0.01075
W-0	10.26	3.125	1.501	0.01166
K-0	9.59	3.177	1.463	0.01163
JH-0	7.45	4.108	1.451	0.00831
CH-0	6.67	4.200	1.402	0.01008
SIL-0	7.50	5.591	1.382	0.00033
B-0	11.20	2.836	1.354	0.00996
TD-0	8.83	3.553	1.351	0.01563
P-0	11.48	2.706	1.342	0.00870
KD-0	10.68	2.989	1.333	0.00934
PD-0	9.02	3.137	1.310	0.01037
SH-0	8.91	3.822	1.207	0.00486
TS-0	7.35	4.152	1.153	0.00843

Abbildung 19: Tabelle aller verwendeten Codebücher eines streams (FFT-Koeffizienten), nach Fehlerrate der BVI-Suche sortiert

Als Nächstes wurde an verschiedenen K-d Bäumen zu einem einzigen Codebuch der Einfluß der Tiefe des Baumes auf die mittlere Anzahl durchsuchter Vektoren, den erreichten Speed-Up beim Quantisieren, sowie auf die Quantisierungsfehler und die Signal-to-Noise ratio untersucht. Unter den 49 zur Verfügung stehenden Codebüchern wurde dafür das Codebuch TS-0 ausgewählt, da mit den dazu erzeugten Bäumen ein vergleichsweise großer Speed-Up erreicht wird, was den Einfluß der Baumtiefe deutlicher macht. Zudem ist die Fehlerrate bei den Bäumen zu diesem Codebuch sehr gering, so daß man die Ergebnisse als eine Art untere Schranke für alle anderen Codebücher ansehen kann.
Auf den folgenden fünf Seiten werden die Resultate gezeigt. Dabei wurden die Diagramme folgendermaßen strukturiert:

- Auf jeder Seite sind 2 Diagramme dargestellt, jeweils eines für die beiden zum Optimieren der Bäume zur Verfügung stehenden Kriterien.
- In jedem Diagramm sind 4 Kurven dargestellt, wobei die zugehörigen Bäume jeweils mit folgender Anzahl an Trainingsvektoren erstellt wurden:
  - 25000 Vektoren
  - 50000 Vektoren
  - 100000 Vektoren
  - 200000 Vektoren
- Alle Meßwerte wurden durch Quantisieren von 100000 Testvektoren gewonnen.

Aus den Diagrammen wird folgendes deutlich :

- Gemäß den theoretischen Überlegungen lassen sich mit dem EOC Kriterium kleinere mittlere bucketsizes als mit dem GOC Kriterium erreichen.
- Dadurch ist der Speed-Up beim EOC Kriterium auch größer als beim GOC Kriterium (gleiche Baumtiefe vorausgesetzt).
- Man sieht aber auch daß man diesen Vorteil des EOC Kriteriums mit einer höheren Fehlerrate und einer schlechteren SNR bezahlt.
- Allerdings wird anhand der Histogramme der Fehlerverteilung deutlich, daß sich eine gleiche Fehlerrate beim GOC Kriterium stärker als beim EOC Kriterium auf die Fehlerdistanz auswirkt.

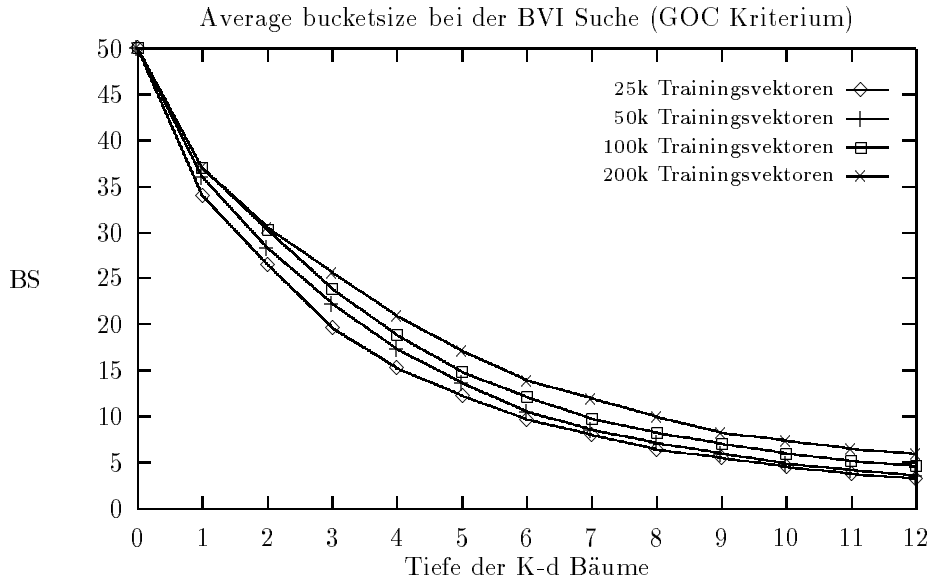

Abbildung 20: Average bucketsize beim GOC Kriterium

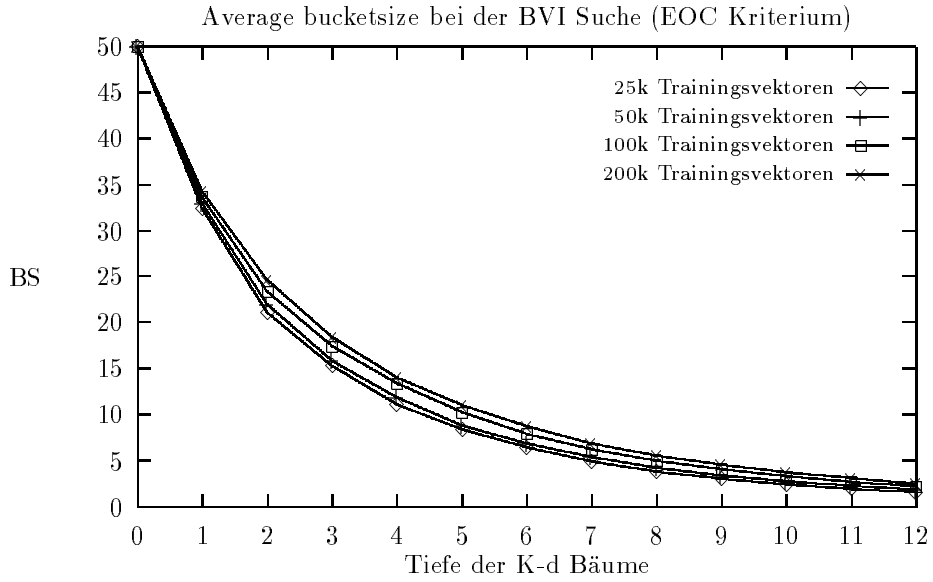

Abbildung 21: Average bucketsize beim EOC Kriterium

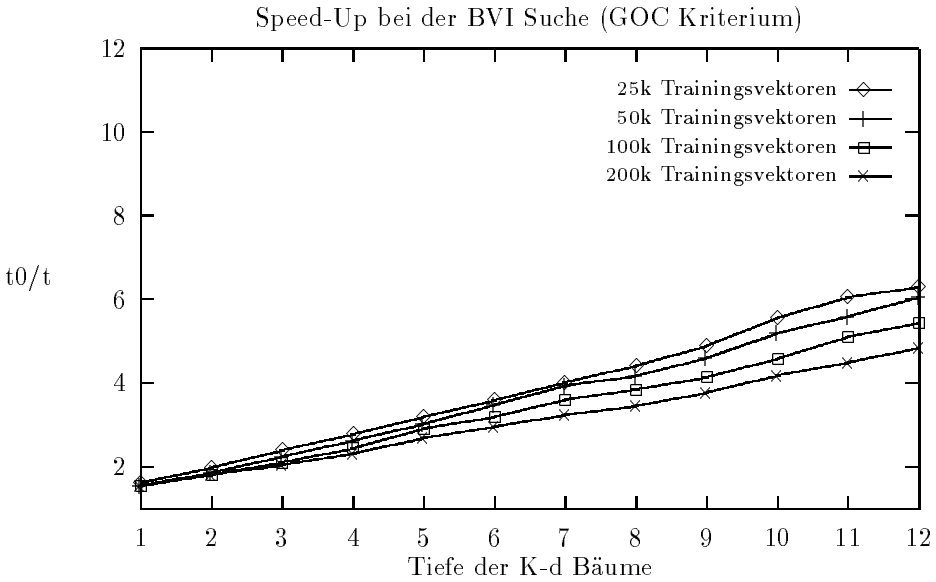

Abbildung 22: Speed-Up beim GOC Kriterium

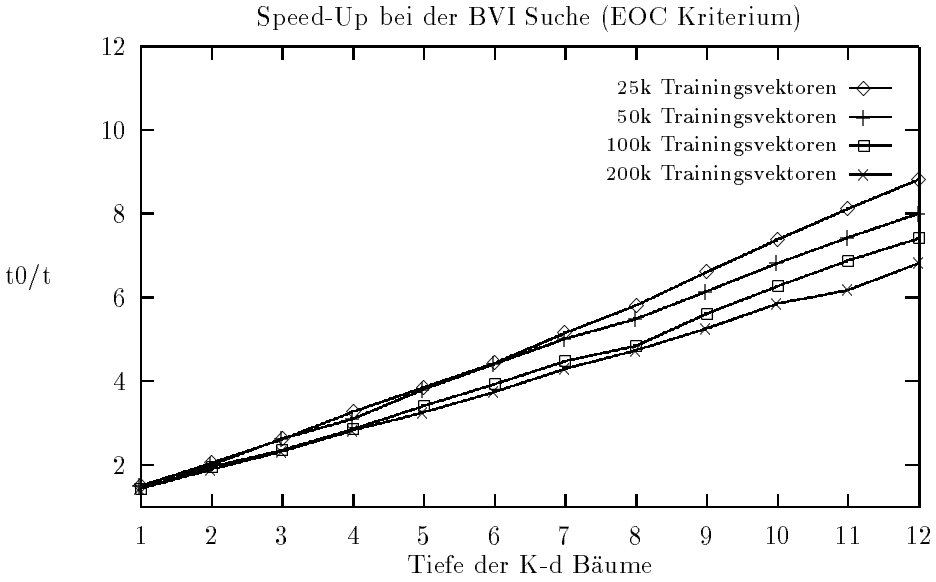

Abbildung 23: Speed-Up beim EOC Kriterium

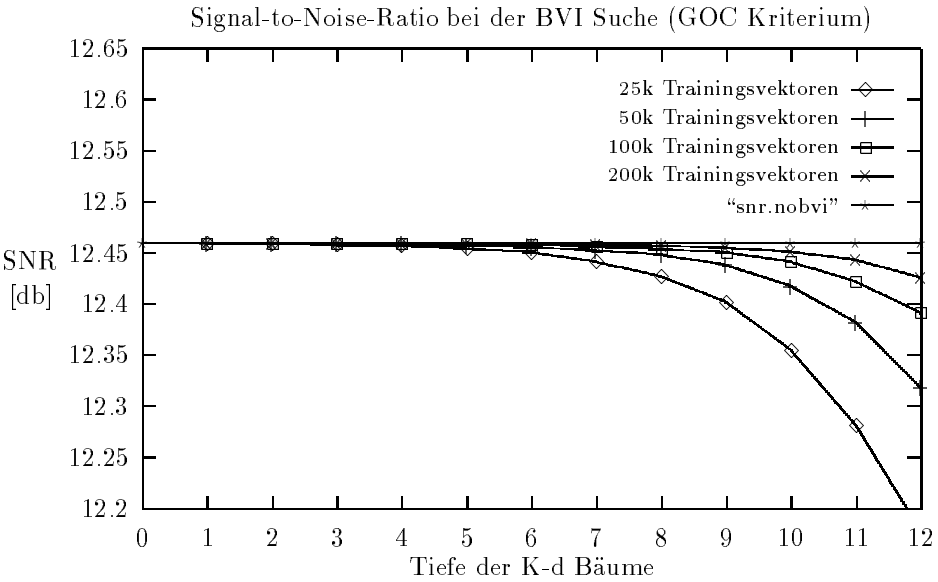

Abbildung 24: SNR beim GOC Kriterium

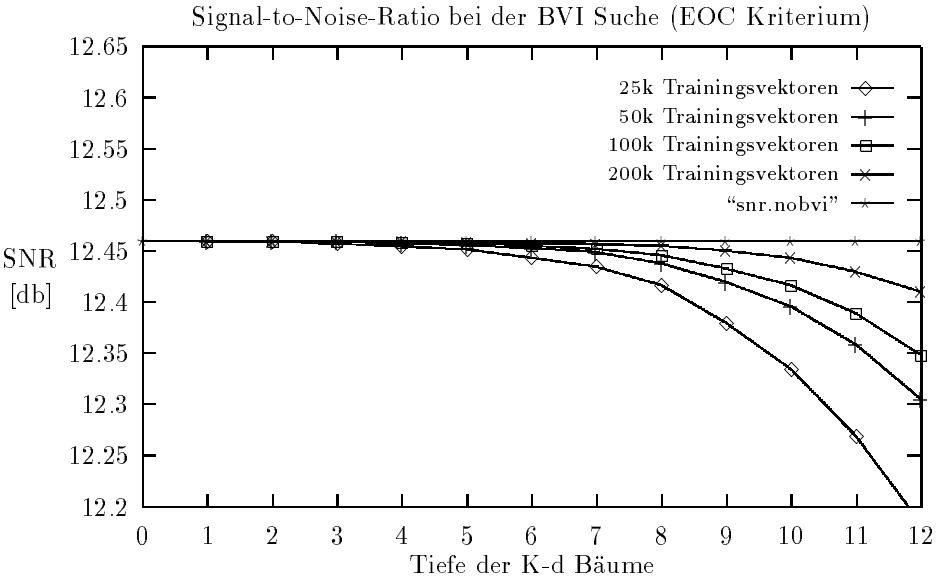

Abbildung 25: SNR beim EOC Kriterium

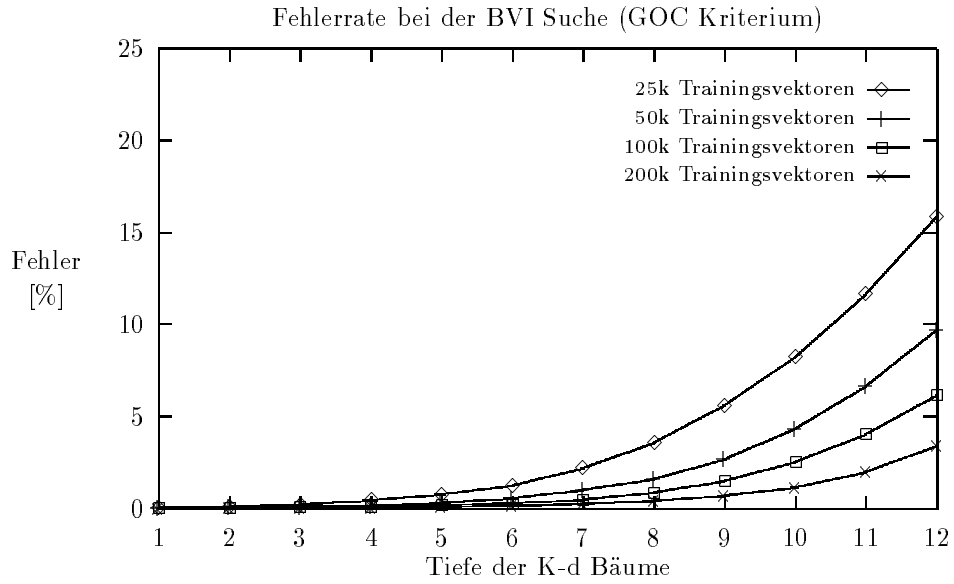

Abbildung 26: Fehlerrate beim GOC Kriterium

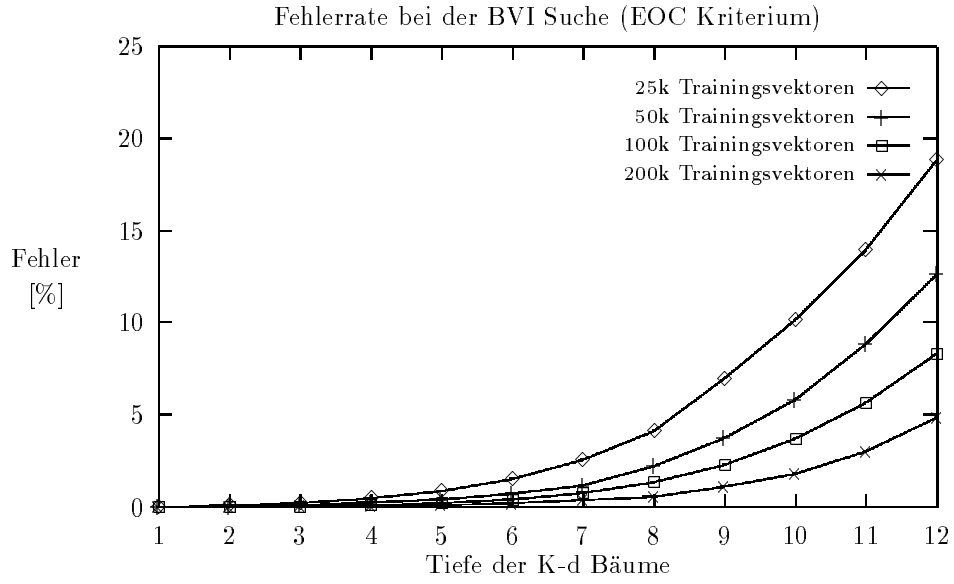

Abbildung 27: Fehlerrate beim EOC Kriterium

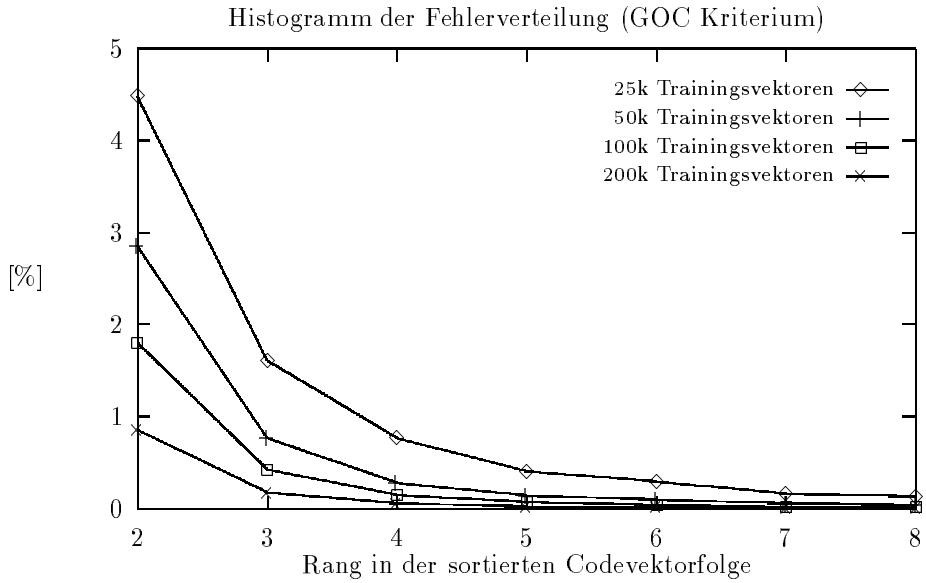

Abbildung 28: Fehlerverteilung beim GOC Kriterium

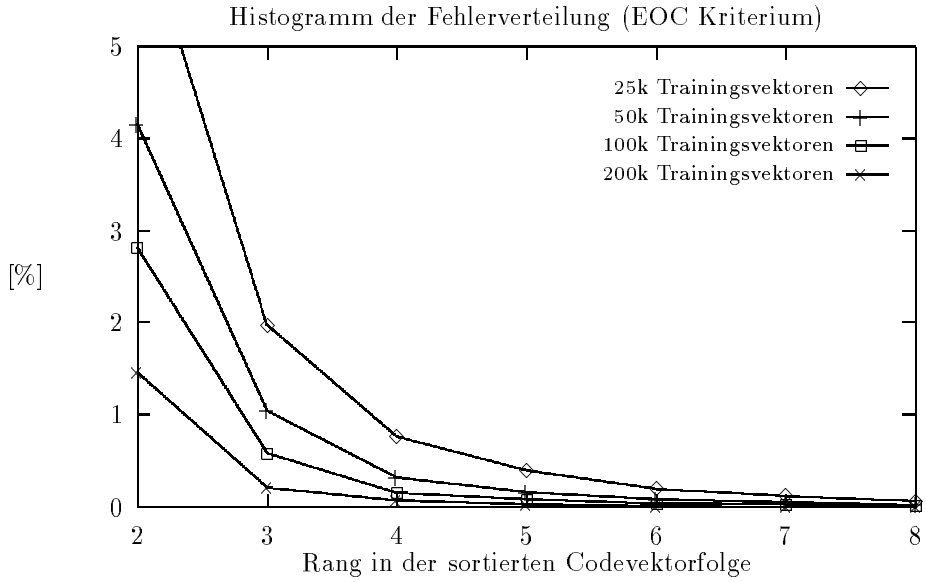

Abbildung 29: Fehlerverteilung beim EOC Kriterium

7.3 Approximative Optimierung

Die approximative Optimierung ist als schnelle Alternative zur korrekten Optimierung gedacht. Dabei läßt sich die Nächster-Nachbar Suche mit dieser Implementierung nur etwa um den Faktor 2 beschleunigen. Die approximative Optimierung mit vorberechneten Voronoi-Projektionen bietet aber noch einen Vorteil: Die Fehlerrate ist im Vergleich zu korrekt optimierten Bäumen etwa um den Faktor 10 kleiner.
Abbildung 30 zeigt den Verlauf der mittleren bucketsize mit zunehmender Tiefe der Bäume.

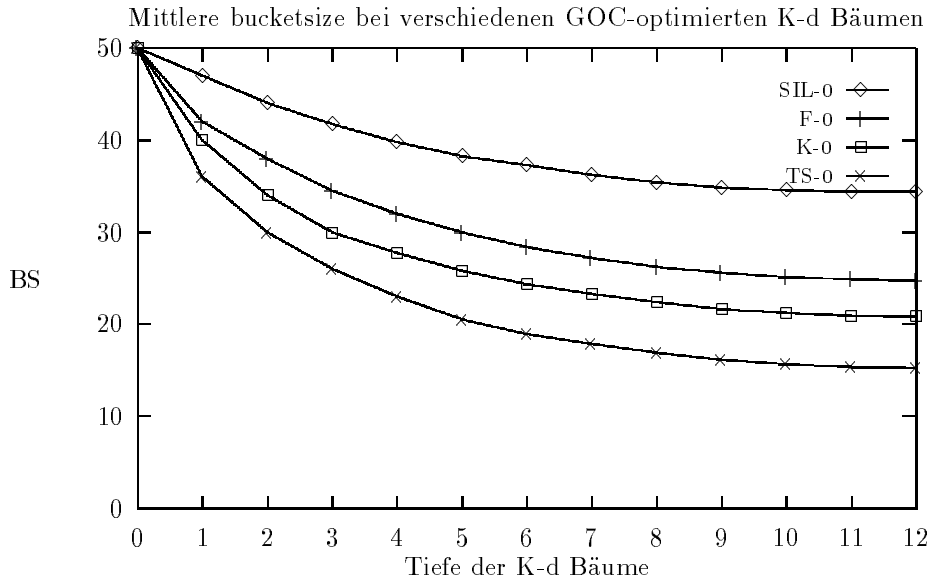

Abbildung 30: Mittlere bucketsize bei der approximativen Optimierung

Die Kurven verlaufen zwar genau wie bei der korrekten Optimierung streng monoton fallend, aber es wird auch deutlich, daß sich schon ab einer Tiefe von etwa 10 keine signifikante Dezimierung der bucketsizes mehr erzielen läßt, die den exponentiell steigenden Speicherbedarf der Bäume mit einer weiteren Steigerung der Tiefe noch rechtfertigen würde.
Abbildung 31 zeigt den Verlauf der Fehlerrate bei der approximativen Optimierung, abhängig von der Tiefe der Suchbäume. Das Diagramm zeigt, daß die Fehlerrate nicht exponentiell mit der Tiefe steigt, wie das bei der korrekten Optimierung der Fall ist. Beim EOC Kriterium scheint sie linear zu verlaufen, während sie beim GOC Kriterium mit zunehmender Tiefe der Bäume immer weniger ansteigt.
Falls die Fehlerrate bei korrekt optimierten Suchbäumen unakzeptabel hoch wird, hat man mit der approximativen Optimierung die Möglichkeit auf praktisch fast fehlerfreie Bäume zurückzugreifen, wobei man allerdings Einbußen im Speed-Up hinnehmen muß.
Da die approximative Implementierung sehr schnell ist, eröffnen sich weitere An-

wendungsfelder in Bereichen, wo man mit ständig neuen Codebüchern quantisieren muß, und daher auch ständig neue K-d Suchbäume berechnen muß. Dies ist zum Beispiel beim trainieren der Codebücher von Spracherkennern der Fall.

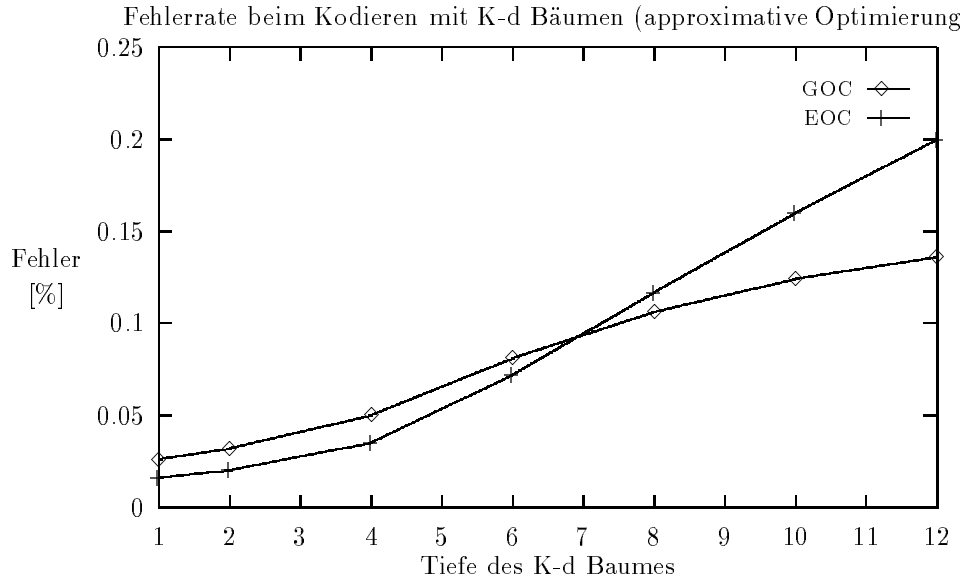

Abbildung 31: Fehlerrate bei der approximativen Optimierung

7.4 Vergleich der beiden Implementierungen

Abbildung 32 vergleicht die beiden vorgestellten Implementierungen. Dazu wurde zu allen 49 zur Verfügung stehenden Codebüchern jeweils ein K-d Suchbaum der Tiefe 12 mit dem Generalized Optimization Criterion (GOC) erstellt und die insgesamt dafür benötigte Zeit gemessen. Zur Bestimmung der Voronoi-Gebiete wurden 100000 Trainingsvektoren verwendet. Unter anderem ist auch der Speicherbedarf aller 49 Suchbäume angegeben. Die Zeiten wurden auf einer DEC alpha Workstation gemessen.

	korrekte Optimierung	approximative Optimierung
Optimierungsdauer	16 h	9 min
Speicherbedarf der Bäume	1900 KBytes	5200 KBytes
Mittlerer Speed-Up	4.1	1.6
Mittlere Fehlerrate	5.4 %	0.2 %

Abbildung 32: Vergleich der beiden Implementierungen

Die Optimierungsdauer steigt bei beiden Implementierungen exponentiell mit

der Tiefe der zu optimierenden Bäume an. In den Versuchen zu Abbildung 32 wurden bewußt relativ große Bäume berechnet, um den hohen Aufwand der korrekten Optimierung zu verdeutlichen. In der Praxis wird man wohl eher Bäume der Tiefe 8 oder 10 benutzen, die wesentlich schneller berechnet werden können. Da der BVI Algorithmus zur Beschleunigung der Nächster-Nachbar Suche entwickelt wurde, wird man sich bei einer Anwendung immer einen möglichst großen Speed-Up bei kleiner Fehlerrate erhoffen. Man wird daher in den allermeisten Fällen der korrekten Optimierung den Vorzug geben und den sehr großen Aufwand zur einmaligen Berechnung der Suchbäume im Angesicht der erzielbaren Speed-Ups tolerieren.

8 Integration der BVI-Suche in JANUS-2

8.1 Übersicht

JANUS-2 ist ein Sprach-zu-Sprach Übersetzungssystem, das die Teilsysteme Erkennung, Übersetzung und Generierung von spontan gesprochener Sprache integriert. Es wird in einem gemeinsamen Projekt der Universität Karlsruhe und der Carnegie Mellon University in Pittsburgh als ein sprecherunabhängiges System entwickelt, das gesprochene Sprache aus einer der Quellsprachen Deutsch oder Englisch in eine der Zielsprachen Deutsch, Englisch oder Japanisch übersetzen kann. Für eine detailliertere Beschreibung des Systems sei auf [13] verwiesen.
Im Rahmen dieser Studienarbeit wurde untersucht, in welchem Maße sich mit dem vorgestellten BVI Algorithmus die Berechnung von HMM Emissionswahrscheinlichkeiten *(score computation)* im akustischen Modell des Spracherkenners von JANUS-2 beschleunigen läßt. Die Emissionswahrscheinlichkeiten können entweder durch diskrete Verteilungen oder durch parametrische Wahrscheinlichkeitsdichtefunktionen nachgebildet werden. Als Dichtefunktionen verwendet man meist multivariate Normalverteilungen (Gauß-Dichten), oft auch gewichtete Überlagerungen mehrerer Normalverteilungen *(mixture densities)*. Man umgeht oft auch die (teure) Berechnung der Exponentialfunktion bei Normalverteilungen, indem man nur den Exponenten derselben berücksichtigt und die darin enthaltene Kovarianzmatrix durch eine Diagonalmatrix ersetzt (Mahalanobis-Abstand). Im Falle diskreter Verteilungen ist die BVI Suche anwendbar, da hier die Merkmalsvektoren durch Nächster-Nachbar Vektorquantisierung auf Prototypen abgebildet werden. Ein kleiner Abstand entspricht hier einer hohen Wahrscheinlichkeit.
Obwohl die Berechnung der Emissionswahrscheinlichkeiten nur einen Teil der Gesamtlaufzeit ausmacht (bei Verwendung sehr großer Vokabulare sogar nur einen kleinen Teil), lohnt sich die Beschleunigung dieser Berechnung dennoch, da sie die Antwortzeit des Erkenners merklich reduziert. Beim Training benötigt der Erkenner sogar fast 50 % der Gesamtlaufzeit zur Nächster-Nachbar Suche, sodaß sich hier die Verwendung des BVI Algorithmus am stärksten auswirken dürfte.

Bei der Integration des BVI Algorithmus in JANUS-2 wurden insgesamt drei Kommandos zur Verwendung des Verfahrens implementiert:

- **mkd (make K-d trees)** zur Berechnung von K-d Suchbäumen zu einer Menge von Codebüchern eines streams von Merkmalsvektoren mittels GOC- oder EOC-Optimierungskriterium. Die berechneten Suchbäume werden direkt mit den Codebüchern verknüpft, so daß sie nach Beendigung von mkd sofort benutzt werden können. Zusätzlich werden die berechneten Suchbäume noch in ein File geschrieben.
- **lkd (load K-d trees)** zum Laden von K-d Suchbäumen, die bereits von mkd erzeugt wurden.
- **fkd (free K-d trees)** zum freigeben des von K-d Suchbäumen belegten Speichers.

Des weiteren wurde die Routine zur Berechnung des Nächsten-Nachbarn eines Merkmalsvektors derart modifiziert, daß sie bei Verfügbarkeit eines Suchbaumes eine BVI Suche anstatt der bisher verwendeten Partial Distance Suche verwendet. Es wurden eine Reihe von Experimenten mit dem derart erweiterten JANUS-2 durchgeführt, um

- den Einfluß der Anzahl der bei der Suchbaumberechnung benötigten Trainingsvektoren,
- den Einfluß der Tiefe der Suchbäume,
- den Einfluß des verwendeten Optimierungskriteriums

auf die Beschleunigung und die Erkennungsrate des Systems zu bestimmen. Insbesondere die Auswirkungen der bei der BVI Suche auftretenden Quantisierungsfehler auf die Erkennungsrate *(word accuracy)* waren dabei von besonderem Interesse. Grundlage für die Versuche waren Sprachdaten der GSST-task (German Spontaneous Scheduling Task) mit vorberechneten Codebüchern mit jeweils 50 Vektoren, bestehend aus 16-dimensionalen LDA-Koeffizienten, die durch lineare Diskriminanzanalyse aus FFT-Koeffizienten hervorgegangen sind.

8.2 Analyse der Beschleunigung

Abbildung 33 zeigt den Verlauf des gemessenen Speed-Ups der Nächster-Nachbar Suche abhängig von der Tiefe der Suchbäume und der Anzahl verwendeter Trainingsvektoren. Man kann erkennen, daß die Verwendung des BVI Verfahrens die score-Berechnung enorm beschleunigt. Es zeigte sich aber auch, daß sich mit

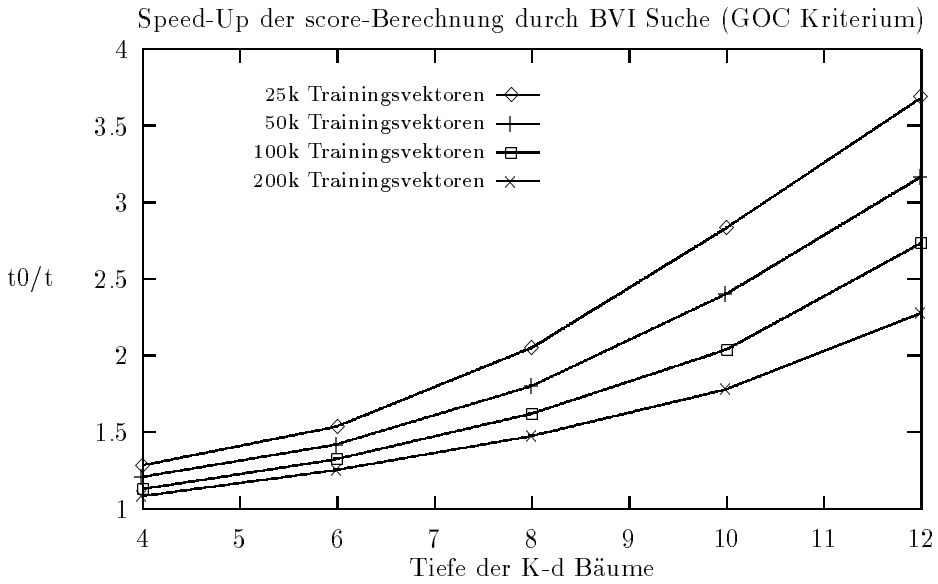

Abbildung 33: Gemessener Speed-Up der score-Berechnungen durch BVI-Suche mit verschiedenen K-d Suchbäumen

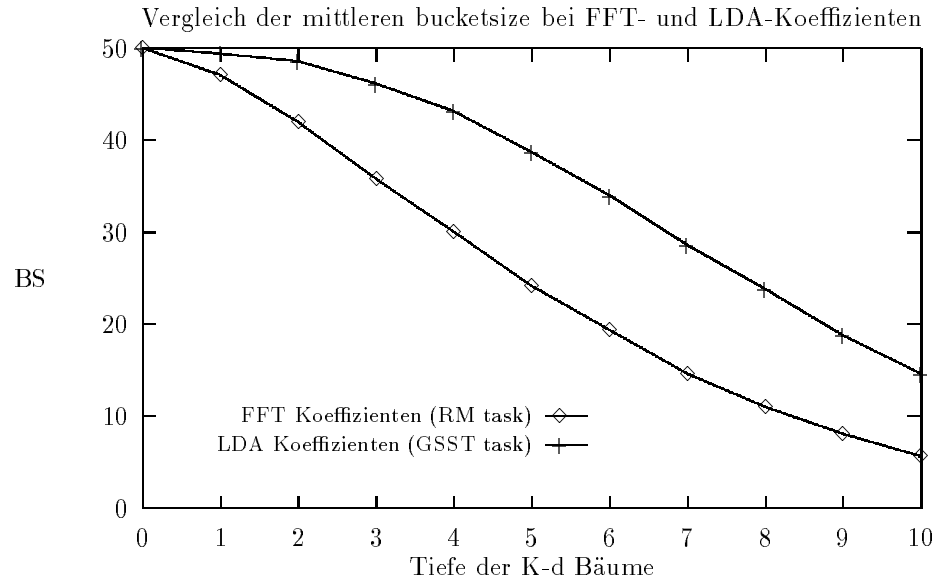

Abbildung 34: Vergleich der mittleren bucketsizes der Suchbäume zu LDA- und FFT-Daten

den verwendeten LDA-Koeffizienten nicht ganz so hohe Speed-Ups wie mit FFT-Koeffizienten erzielen ließen. Abbildung 34 veranschaulicht dies anhand der mittleren bucketsizes, die bei K-d Bäumen der jeweiligen Merkmalsvektorenart beobachtet wurden. Der Grund für das schlechtere Abschneiden der LDA-Koeffizienten wird in der größeren Varianz der Daten der GSST-task (LDA-Koeffizienten) ge-

genüber der RM-task (FFT Koeffizienten) vermutet.
Prinzipiell läßt sich der Speed-Up in weiten Bereichen durch Baumtiefe und Größe der Trainingsvektormenge skalieren, wobei die Beschleunigung nur durch den exponentiell steigenden Speicherplatz- und Rechenzeitbedarf der Suchbäume beschränkt wird.

8.3 Auswirkungen auf die Erkennungsrate

Abbildung 35 zeigt die Auswirkungen der Quantisierungsfehler, die bei der BVI Suche auftreten, auf die Worterkennungsrate des Spracherkenners. Da der Anteil falsch klassifizierter Merkmalsvektoren exponentiell mit der Tiefe der verwendeten Bäume zunimmt, liegt es nahe, einen ähnlichen Verlauf bei der Erkennungsrate zu vermuten. Dies wurde auch, wie man sieht, in den Versuchen bestätigt. Man kann diesem Phänomen aber durch die Verwendung größerer Mengen an Trainingsvektoren begegnen, was aber einen erhöhten Aufwand bei der Berechnung der Suchbäume zur Folge hat.

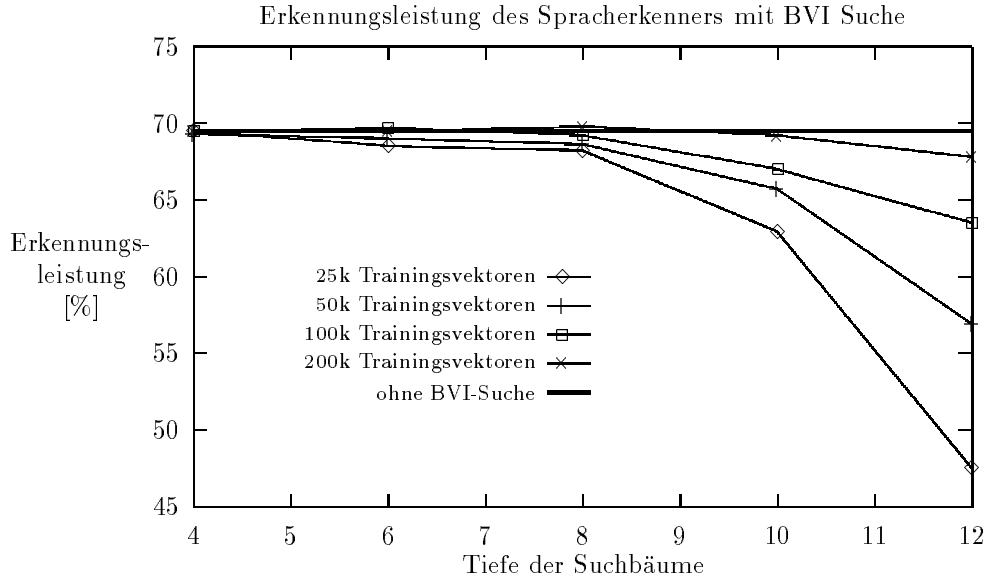

Abbildung 35: Auswirkungen der Anzahl der verwendeten Trainingsvektoren und der Tiefe der Suchbäume auf die Erkennungsleistung des Spracherkenners

Formal lassen sich Speed-Up S und Fehlerrate E als Funktionen der beiden Parameter Baumtiefe d und Anzahl verwendeter Trainingsvektoren n darstellen: $S = S(d, n)$ und $E = E(d, n)$. Maximierung des Speed-Ups $S(d, n)$ bei gleichzeitiger Minimierung der Fehlerrate erzwingen aber sehr große Trainingsvektorenmengen, so daß man in der Praxis durch eine beschränkte Anzahl an Trainingsvektoren stets einen Kompromiß zwischen den Faktoren

- Hoher Speed-Up

- Tolerierbare Fehlerrate
- Verfügbare Rechner-Resourcen (Suchbaum-Erzeugung)

finden muß.

9 Schlußbemerkungen

Mit dem Bucket Voronoi Intersection Algorithmus konnte ein ursprünglich für allgemeine Vektorquantisierungsaufgaben entworfenes schnelles Nächster-Nachbar Suchverfahren erfolgreich zur Beschleunigung der score-Berechnungen eines HMM-Spracherkenners eingesetzt werden. Dies war nicht von vornherein zu erwarten, da in einem vergleichsweise hochdimensionalen Merkmalsraum mit wenigen Codebuchvektoren gearbeitet wurde (vgl. [1]). Es sollen nun noch einige Erweiterungen und Verbesserungen des Verfahrens vorgeschlagen werden, die eventuell zu weiterführenden Arbeiten anregen könnten.

	#Codebuchvektoren N	Dimension d	Bitrate r	Suchaufwand bei Tiefe 10	max. Speed-Up
Ram. und Pal.	1024	2	5.000	3.0	85
Ram. und Pal.	1024	4	2.500	6.9	60
Ram. und Pal.	1024	6	1.750	8.0	40
Ram. und Pal.	1024	8	1.250	7.9	35
Ram. und Pal.	1024	10	1.000	8.3	35
Fritsch	50	16	0.352	1.2	7

Abbildung 36: Abhängigkeit der Effizienz der BVI-Suche von der Bitrate

- Vergleiche zwischen den Arbeiten in [1] und dieser Studienarbeit lassen eine Korrelation zwischen erreichbarem Speed-Up und Bitrate r vermuten (siehe 36), so daß die Verwendung größerer Codebücher oder kleinerer Dimensionen sicherlich eine weitere Beschleunigung der NN-Suche erlauben würde.
- Die approximative Berechnung der Voronoi-Gebiete macht den Großteil des Gesamtaufwandes des BVI-Algorithmus zur Erzeugung von K-d Suchbäumen aus. Die ansatzweise Parallelisierung dieses Teilproblems auf einem datenparallelen SIMD-Rechner (MasPar MP-1) versprach eine sehr gute Effizienz, konnte aber im Rahmen dieser Studienarbeit nicht weiter untersucht werden. Es ließe sich aber auch der gesamte Optimierungsprozeß bei der Baumerzeugung parallelisieren, beispielsweise in dem jeder Prozessor einen Knoten des Suchbaumes optimiert.
- Das BVI-Verfahren arbeitet grundsätzlich mit vollständigen Binärbäumen, obwohl es Vorteile bringen würde, wenn man zu einem Knoten des Baumes keine weiteren Nachfolger mehr berechnet, falls das zugehörige bucket leer ist, da dann auch alle Nachfolger leere buckets haben.

- Schließlich könnte man statt mit einfachen Trennhyperebenen der Form $x_j < h$ auch mit allgemeineren Trennebenen arbeiten. Dadurch könnte die Form der Voronoi-Gebiete besser angenähert werden, allerdings wäre der Vorteil der einfachen Berechnung des Nachfolgeknotens dahin. Allgemeinere Trennhyperebenen würden auch ein neues Optimierungskriterium verlangen, welches die Lage der Ebenen im Merkmalsraum bestimmt. Unter Umständen würden sich dafür auch die diskriminativen Eigenschaften Neuronaler Netze eignen.

Literatur

[1] Ramasubramanian, V.; Paliwal, K. K.: *„Fast K-Dimensional Tree Algorithms for Nearest Neighbor Search with Application to Vector Quantization Encoding“* IEEE Transactions on Signal Processing, Vol. 40, No. 3, March 1992

[2] Gersho, A.; Gray, R. M.: *„Vector Quantization and Signal Compression“* Kluwer Academic Publishers, Dordrecht, Netherlands, 1992

[3] Gray, R. M.: *„Vector Quantization“* IEEE ASSP Magazine, volume 1, pp.4-29, April 1984

[4] Gersho, A.; Cuperman, V.: *„Vector Quantization: A pattern matching technique for speech coding“* IEEE Commun. Mag., volume 21, pp.15-21, December 1983

[5] Bentley, J. L.; *„Multidimensional binary search trees used for associative searching“* Commun. Ass. Comput. Mach., volume 18, no.9, pp.509-517, September 1975

[6] Friedman, J. H.; Bentley, J. L.; Finkel, R. A.: *„An algorithm for finding best matches in logarithmic expected time“* ACM Trans. Math. Software, volume 3, no. 3, pp.209-226, September 1977

[7] Cheng, D. Y.; Gersho, A.: *„A fast codebook search algorithm for nearest neighbor pattern matching“* in Proc. IEEE ICASSP 1986, pp.265-268

[8] Ramasubramanian, V.; Paliwal, K. K.: *„A generalized optimization of the K-d tree for fast nearest neighbor search“* in Proc. 4th IEEE Region 10 Int. Conf. TENCON 1989, pp.565-568

[9] Equitz, W. H.: *„Fast algorithms for vector quantization encoding of pictures“* in Proc. IEEE ICASSP 1987, pp.725-728

[10] Cheng, D. Y.; Gersho, A.; Ramamurthi, B., Shoham, Y.: *„Fast search algorithms for vector quantization and pattern matching“* in Proc. IEEE ICASSP 1984, volume 1, March 1984, pp.9.11.1-9.11.4

[11] Linde, Y.; Buzo A.; Gray, R. M.: *„An algorithm for vector quantization design“* IEEE Trans. Commun., volume COM-28, pp 84-95, January 1980

[12] Paliwal, K. K.; Ramasubramanian, V.: *„Effect of ordering the codebook on the efficiency of the partial distance search algorithm for vector quantization“* IEEE Trans. Commun., volume. COM-37, pp.538-540, May 1989

[13] Waibel, A. et. al.: *„JANUS 93: Towards spontaneous speech translation“* in Proc. IEEE ICASSP 1994