Sprachmodelle für einen Buchstabier-Erkenner

Diplomarbeit

Martin Betz

Bearbeitungszeitraum:
15. Nov. 1993 – 15. Mai 1994

Institut für Logik, Komplexität und Deduktionssysteme

Betreuer: Prof. Dr. Alex Waibel
Dipl.-Inform. Hermann Hild

Hiermit erkläre ich, vorliegende Arbeit selbständig erstellt und keine anderen als die angegebenen Quellen verwendet zu haben.

Karlsruhe, 15. Mai 1994

M. Betz

Martin Betz

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Spracherkennung

Werbebroschüren für Spracherkennungsprodukte im PC-Bereich oder die Lektüre populärwissenschaftlicher Magazine erwecken mitunter den Anschein, das perfekte System zur Spracherkennung existiere bereits und ist für jedermann verfügbar, oder es kann nur noch kurze Zeit dauern, bis es so weit ist. Die Realität ist weit davon entfernt – beim Großteil der heute laufenden Spracherkennungssysteme handelt es sich um Forschungsprojekte von Universitäten oder großen Firmen, die für den Alltagseinsatz nicht geeignet sind. Das hängt damit zusammen, daß die Erkennungsraten in der Praxis weit hinter den Erwartungen zurückbleiben, besonders wenn kontinuierlich gesprochene Sprache erkannt werden soll und außerdem sehr schnelle und damit teure Hardware benötigt wird, um den enormen Rechenaufwand beherrschen zu können.

Eine kommerzielle Anwendung aus diesem Sektor, welche sich in jüngerer Zeit einer gewissen Beliebtheit mit steigender Tendenz erfreut, ist das sogenannte Telephone-Banking. Es werden hier manchmal Spracherkennungssysteme eingesetzt, die den Kundenverkehr abwickeln. Dabei handelt es sich allerdings um sehr primitive Spracherkenner, die üblicherweise nur die Ziffern von Null bis Zehn und einige weitere Kommandowörter unterscheiden können. Bei einem derartig beschränkten Vokabular kann tatsächlich eine fast fehlerfreie Erkennung erreicht werden.

Ein der Ziffernerkennung verwandtes Problem ist das Erkennen von Buchstaben, bzw. buchstabierten Wörtern. Es ist allerdings etwas anspruchsvoller, da hier einerseits ein etwas größeres Vokabular gegeben ist und andererseits, was die eigentliche Schwierigkeit ausmacht, die Worte des Vokabulars noch sehr leicht verwechselbar sind (z. B. '*b*', '*d*', '*g*' oder '*l*', '*m*', '*n*'). Einsatzbereiche von Systemen, die buchstabierte Worte erkennen können, liegen in Auskunftsystemen aller Art, insbesondere bei Telefonauskunftsdiensten, oder innerhalb komplexerer Spracherkennungssysteme als Korrekturmodul, welches erlaubt, neue oder falsch erkannte Wörter zu buchstabieren, um sie dem System bekannt zu machen.

1.2 Buchstabenerkennung

Die Arbeiten am Institut für Logik, Komplexität und Deduktionssysteme der Universität Karlsruhe in diesem Bereich beschäftigen sich mit der Erkennung buchstabierter Nachnamen. Es existiert ein System von Hermann Hild, welches nach Verfahren implementiert wurde, die ursprünglich nicht speziell auf das Problem der Erkennung buchstabierter Nachnamen ausgerichtet waren. Dieses System erkennt ungefähr jeden zweiten Namen falsch und ist damit z. B. für ein Telefonauskunftsystem nicht geeignet. Eine kleine Erweiterung, der Einbau eines Mechanismus, welcher den Erkennungsprozeß so beeinflußt, daß nur Namen aus einer vorgegebenen Liste erkannt werden, brachte eine drastische Steigerung der Erkennungsraten, so daß nur noch wenige Prozent falsch erkannt wurden. Das Problem bei diesem Verfahren war, daß es sehr viel Speicher verbrauchte und somit oftmals langsam war, da während der Erkennung nicht alle Daten im Hauptspeicher gehalten werden konnten und somit auf Platte ausgelagert werden mußten. Ein weiterer Nachteil war, daß durch den enormen Speicherplatzbedarf des Programms und die beschränkte Speichergröße üblicher Workstations die Größe der Namenslisten auf ungefähr 1000 Namen beschränkt war.

1.3 Ziel der Arbeit

Ziel der Arbeit war, verschiedene Methoden zu entwickeln und zu vergleichen, die es erlauben sollten, Namenslisten größeren oder beliebig großen Umfangs zu verwenden, um dadurch die Anzahl der Namen, die erkannt werden können zu erhöhen. Dabei sollten ähnlich gute Erkennungsleistungen wie bei der speicheraufwendigen, ersten Implementierung erzielt werden. Gleichzeitig sollte die Zeit, die zum Erkennen eines gesprochenen Namens benötigt wird, möglichst kurz ausfallen. Dem Institut stand hierzu eine 32000 verschiedene Nachnamen umfassende Liste in maschinenlesbarer Form zur Verfügung.

1.4 Danksagung

Besonders bedanken möchte ich mich bei meinem Betreuer Hermann Hild, den ich bei Unklarheiten jederzeit ansprechen konnte und der sich sehr viel Zeit für mich genommen hat. Weiterhin bedanke ich mich bei Monika für die Vorschläge zum n-best Algorithmus, bei Petra für ihre Unterstützung beim Perplexitäts-Kapitel und bei Andreas und nochmal bei Hermann fürs Korrekturlesen.

Kapitel 2

Grundlagen

In diesem Kapitel möchte ich die dem System, wie es jetzt implementiert ist, zugrundeliegenden Algorithmen erklären. Dazu werde ich zuerst vorstellen, wie man mit dem TDNN, das von A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, und K. Lang 1988 in [WHH+89] vorgestellt wurde, Phoneme erkennen kann. Anschließend, wie aufgrund dieser kontinuierlich gesprochene Sprache erkannt werden kann. Am Ende des Grundlagen-Kapitels befindet sich eine theoretische Betrachtung verschiedener mit Sprachmodellen zusammenhängender Aspekte.

2.1 TDNN

Ein neuronales Klassifikationsnetz besteht normalerweise aus mehreren Schichten, die 'übereinander' angeordnet sind und zwischen denen Verbindungen bestehen, über die gewisse Eingaben der unten liegenden Schicht gewichtet summiert an die darüberliegende Schicht weitergeleitet werden. In der Ausgabeschicht (der obersten Schicht) entspricht die Zahl der Zellen der Zahl der Klassen.

Ein Time Delay Neural Network (TDNN) funktioniert im Prinzip ähnlich, mit dem Unterschied, daß die Eingabe keine statische Momentaufnanme ist, sondern ein sich über die Zeit kontinuierlich veränderndes Signal. Im Fall der Spracherkennung liegt als Eingabedatum ein Vektor von aus den Sprachsignalen extrahierten Merkmalen vor, in unserem Fall sind es 16 Melscale-Koeffizienten*. Über die Eingabedaten gleitet ein 3 Frames† breites Fenster. Die $3 * 16$ Koeffizienten, die in diesem Fenster liegen, stellen die Eingabe an eine Spalte der ersten versteckten Schicht dar. Im nächsten Schritt wird das Fenster um einen Frame weitergeschoben und die nächsten $3 * 16$ Koeffizienten, die sich zu 2/3 mit den alten überlappen, werden an die nächste Spalte der ersten versteckten Schicht weitergeleitet. Zwischen der versteckten- und der Ausgabeschicht

*Melscale-Koeffizienten sind FFT-Koeffizienten, bei denen die Frequenzbänder im unteren Bereich eine höhere Auflösung besitzen als im oberen. Dadurch wird die Physiologie des menschlichen Hörens simuliert.

†Ein Frame bezeichnet den Eingabedatenvektor zu einem Zeitpunkt. Er besteht aus den alle 10 msec abgetasteten Melscale-Koeffizienten über einen jeweils 16 msec breiten Bereich.

wird genauso verfahren. Hier wurde eine Fensterbreite von 5 Frames gewählt (Abbildung 2.1). Die einzelnen Zeilen der Ausgabeschicht beinhalten die Bewertungen der verschiedenen Phonemhypothesen zu den jeweiligen Zeitpunkten.

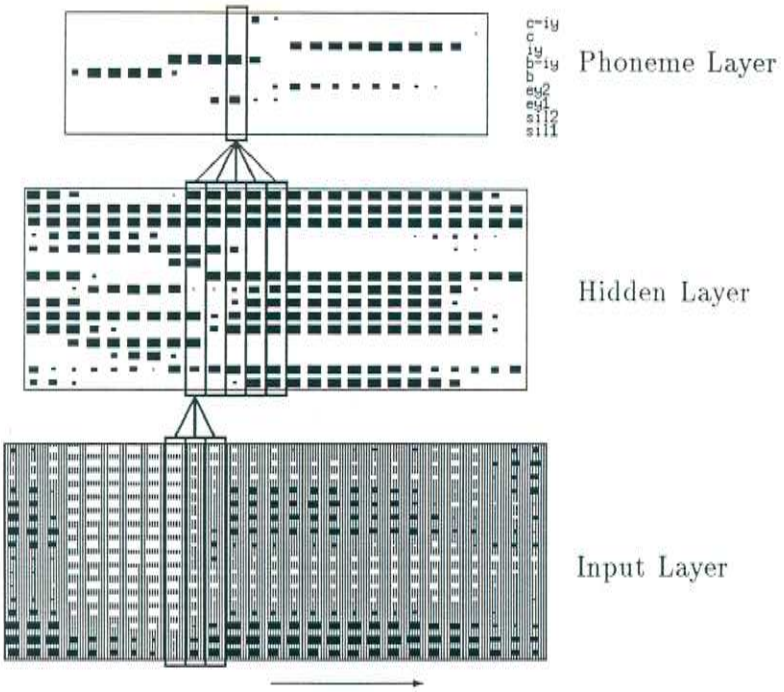

Abbildung 2.1: Ein Auschnitt aus dem TDNN zur Phonemerkennung (nach [HW93.2])

In [WHH+89] sind Tests beschrieben, die das Ziel hatten, die schwer voneinander zu unterscheidenden Phoneme *b*, *d* und *g* zu erkennen. Dabei wurden die Bewertungen für jedes Phonem über den gesamten Zeitraum addiert und dasjenige mit der höchsten Endsumme als Sieger definiert. Verschiedene sprecherabhängige Tests ergaben Erkennungsraten zwischen 97.5% und 99.1%

2.2 DTW

Mit Phonemerkennung alleine kann man nicht viel anfangen. Das Ziel soll sein, ganze Worte oder Sätze zu erkennen, in unserem Fall Sätze, in denen zwischen den einzelnen Wörtern keine bewußte Pause gesprochen werden muß. Zum Problem der kontinuierlichen Spracherkennung existieren verschiedene Lösungsansätze. Ein weit verbreiteter Algorithmus, um aus Phonemhypothesen kontinuierlich Worte zu erkennen, ist der *One-Stage Dynamic Programming* Algorithmus von Herman Ney ([Ney90]), der auch in unserem System verwendet wird.

Neys Algorithmus bietet den Vorteil, daß er nur einen einzigen, zeitsynchronen Durchlauf braucht, in dem gleichzeitig die Wortgrenzen gefunden werden und die Erkennung durchgeführt wird [Ney90]. Er ist zudem nicht sehr rechenzeitaufwendig, so

daß er selbst auf nichtparallelen Rechnern so implementiert werden kann, daß er fast ohne Zeitverzögerung nach dem Einsprechen eines Satzes das Ergebnis liefert.

Der Algorithmus benötigt als Eingabe eine Phonemhypothesenbewertung für den gesprochenen Satz. Er bekommt diese in unserem Fall von dem TDNN geliefert (Abbildung 2.2). Die Werte stellen in etwa Wahrscheinlichkeiten dafür dar, daß zu diesem Zeitpunkt die Akustik mit dem entsprechenden Phonem übereinstimmt.

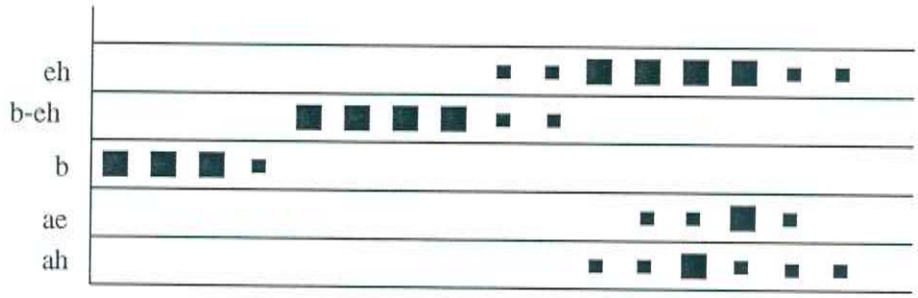

Abbildung 2.2: Phonemhypothesen aus dem TDNN.

Um Phoneme zur Worterkennung benutzen zu können, müssen diese zu Wortmodellen gruppiert werden. Wir benutzen Modelle, bei denen in jedem Zustand nur erlaubt ist, im selben zu verweilen oder in den Nachfolgezustand überzugehen (Abbildung 2.3).

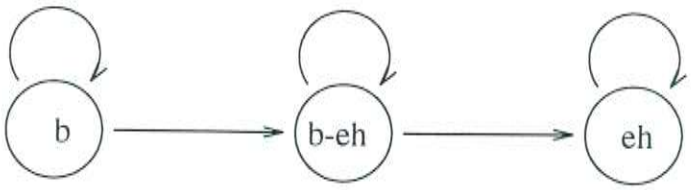

Abbildung 2.3: Wortmodell für B aus 3 Phonemen.

Für alle Worte des Vokabulars werden entsprechende Modelle erstellt und anhand der Phonemhypothesen wird versucht, den besten Pfad durch ein Wortmodell zu finden (Bild 2.4). Das Gütekriterium für einen Pfad ist die Summe der Phonembewertungen entlang des Pfades.

Aufgrund der Beschaffenheit der Wortmodelle dürfen Pfade von einem Zeitschritt zum nächsten entweder waagrecht (Verweilen im selben Zustand) oder im 45-Grad Winkel schräg nach oben gehen (Übergang in Nachfolgezustand). Bei Zuständen am Ende eines Wortmodells darf der Pfad in den Startzustand eines beliebigen Wortmodells übergehen (Abbildung 2.5).

Um die 'Steigung' der Pfade, d. h. die Länge, die in einem Wortmodell verblieben wird, zu beeinflussen, kann man Übergangsstrafen einführen. Das sind (kleine) Werte, die je nach Übergangsart (selber Zustand, Nachfolgezustand, Übergang in anderes Wortmodell) von dem Pfad abgezogen werden. Wenn Strafen für das Verweilen

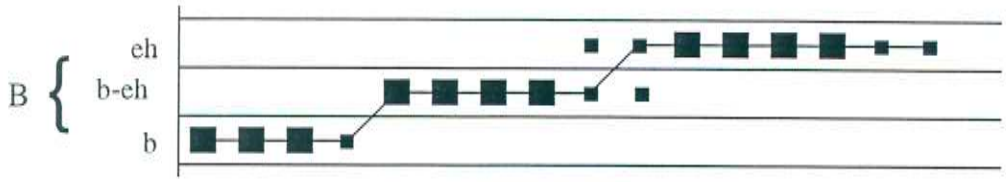

Abbildung 2.4: Suchen des optimalen Pfades durch ein Wortmodell.

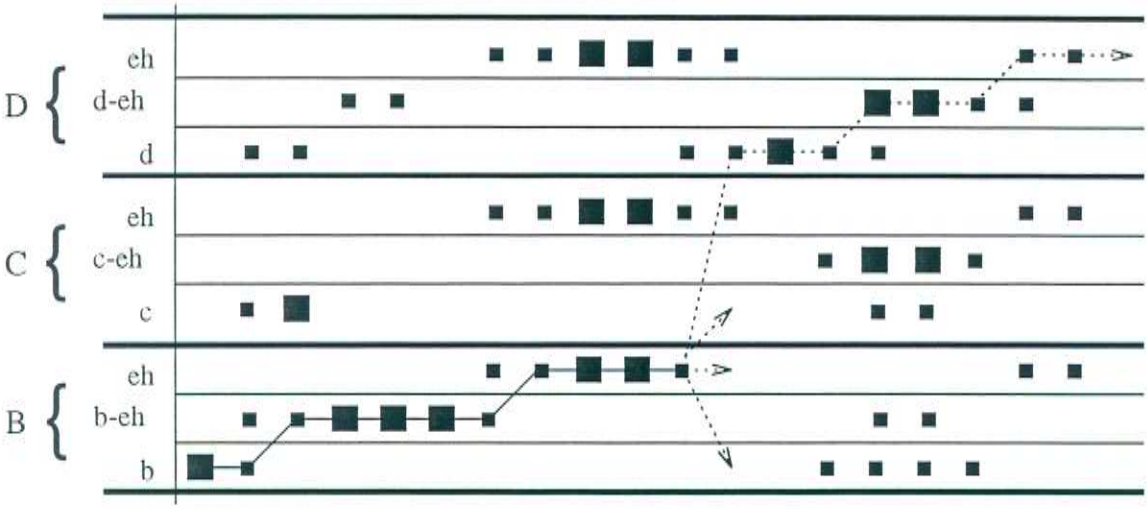

Abbildung 2.5: Wenn der Pfad im letzten Zustand von '*B*' angekommen ist, kann entweder hier verweilt werden oder in den Anfangszustand eines anderen (evtl. des selben) Wortmodells gesprungen werden.

im gleichen Zustand zu hoch angesetzt werden, wird der Algorithmus sehr schnell durch die Wortmodelle laufen und es werden vermutlich viele Einfügefehler entstehen. Der umgekehrte Fall entsteht, wenn die anderen Übergangsarten zu stark bestraft werden. Eine andere Methode, die Mindestdauer, die ein Pfad in einem Wortmodell verweilen muß zu steuern, besteht darin, die Zustände zu vervielfachen. Dieses Verfahren wird in Abschnitt 3.1.1 näher erläutert.

Die Übergänge zwischen zwei Wortmodellen sind der Ansatzpunkt, den der DTW-Algorithmus bietet, ein Sprachmodell einzubauen (siehe Abschnitt 3.2).

2.3 Sprachmodelle

Bei der Spracherkennung geht es darum, nach dem Vorliegen der akustischen Daten A eine Bewertung $P(S|A)$ für eine Satzhypothese S zu finden. Die Anwendung der

bekannten Bayes-Formel ergibt:

$$P(S|A) = \frac{P(S)P(A|S)}{P(A)} \tag{2.1}$$

Da $P(A)$ bei einer Testserie von n verschiedenen Sätzen immer $1/n$ und somit für alle A gleich ist, heißt das für den Erkenner, daß der Zähler des Bruches maximiert werden muß und die Hypothese $\hat{S}$ als die beste gewählt wird, für die gilt:

$$\hat{S} = argmax_S P(S)P(A|S) \tag{2.2}$$

Man sieht hieraus, daß $P(S)$ eine nicht unwesentliche Rolle in der Ermittlung der besten Hypothese $\hat{S}$ spielt. Es ist die Aufgabe eines Sprachmodells, geeignete Möglichkeiten zur Berechnung der a priori Wahrscheinlichkeiten P(S) bereitzustellen. *Das Sprachmodell* schlechthin existiert nicht, es muß vielmehr dem jeweiligen Aufgabengebiet angepaßt werden, richtet sich besonders nach der Größe des Vokabulars, und stellt im Allgemeinen einen Kompromiß zwischen Speicherplatz, Rechenzeit und Erkennungsleistung dar. Unter der Annahme, daß ein Satz ein n-Tupel von Worten ist,

$$S = \langle w_1, w_2, \ldots w_n \rangle \tag{2.3}$$

kann ein einfaches Sprachmodell, welches von der Unabhängigkeit der hintereinander gesprochenen Worte ausgeht, folgendermaßen aussehen:

$$P(S) = P(\langle w_1, w_2, \ldots w_n \rangle) = P(w_1)P(w_2)\ldots P(w_n) \tag{2.4}$$

Die Werte für die $P(w_i)$ in diesem sogenannten **Unigram-Modell** ergeben sich aus den aus einem Trainingsset statistisch ermittelten Worthäufigkeiten der einzelnen w_i. Die Annahme, daß zwischen den w_i keine Abhängigkeiten bestehen ist allerdings sehr schwach – deshalb werden normalerweise in Spracherkennern Modelle benutzt, die die Berechnung von $P(w_i)$ von vorherig erkannten Worten abhängig machen. Solche Modelle werden **n-gram-Modelle** genannt. Weit verbreitet sind **Trigramme**:

$$P(S) = \prod_{i=1}^{n} P(w_i|\langle w_{i-2}, w_{i-1} \rangle) \tag{2.5}$$

Wenn es bei der Aufgabe um die Erkennung gesprochener Texte geht, bei denen die Vokabulare meist recht groß sind, ist es kaum möglich, mehr als die letzten beiden erkannten Worte in die Berechnung von $P(w_i)$ einfließen zu lassen, da hier z. B. selbst schon bei einem relativ kleinen Vokabular von 100 Worten Wahrscheinlichkeiten für $100^3 = 1$ Million Kontexte gespeichert werden müßten, um alle Kombinationen abzudecken.

In unserem dedizierten Aufgabengebiet, dem Erkennen von buchstabierten Nachnamen, verhält es sich anders. Das Vokabular beschränkt sich auf die Buchstaben des Alphabets (und einige Spezialwörter). Zusätzlich ist eine Liste aller erlaubten Nachnamen in Form des Karlsruher Telefonbuchs vorhanden, so daß das Problem der unbekannten Wortsequenz beim Berechnen der a priori Wahrscheinlichkeiten nicht auftritt. Deshalb haben wir uns entschieden, ein Sprachmodell zu benutzen, bei dem die Wahrscheinlichkeiten von allen Vorgängern abhängen.

$$P(S) = \prod_{i=1}^{n} P(w_i|\langle w_1, w_2, \ldots w_{i-1} \rangle) \tag{2.6}$$

2.3.1 Darstellung des Sprachmodells

In dem System wurde das Sprachmodell als ein endlicher, deterministischer Automat abgebildet und als gerichteter Graph implementiert. Die Knoten des Graphen entsprechen den Zuständen und die Kanten den Übergängen, bei denen jeweils ein Symbol (hier ein Buchstabe des Alphabets) ausgegeben wird. Es existieren ein Startzustand (S) und ein oder mehrere Endzustände (F). (Abbildung 2.6)[‡].

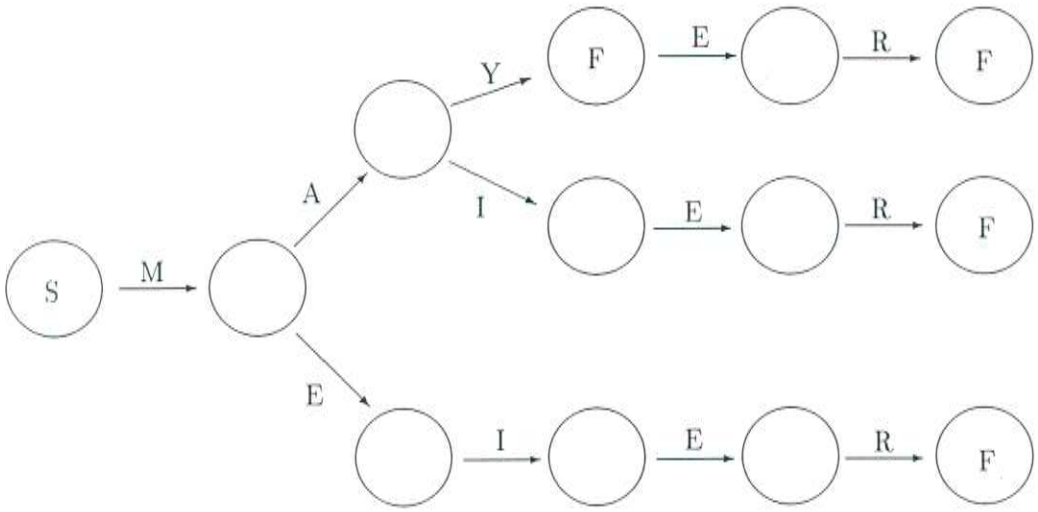

Abbildung 2.6: Modell für May, Mayer, Maier, Meier als Baum.

Um Speicherplatz zu sparen und um die Zahl der Kanten so klein wie möglich zu halten (wichtig für die erste Implementierung des Erkenners mit Grammatik), wird kein Baum, sondern ein minimaler Graph erzeugt, bei dem gleiche Wortenden wieder zusammengeführt werden (Abbildung 2.7).

Mit den Übergängen sind Wahrscheinlichkeiten verbunden, die die Verteilung der verschiedenen Nachfolger eines Zustandes beschreiben sollen. Sie werden gewonnen, indem beim Erzeugen des Graphen für jeden Übergang mitgezählt wird, wie oft er 'benutzt' wird, d. h. wenn die Sätze Mayer und Maier eingefügt werden, werden die Übergänge 'M', 'a', 'e' und 'r', je zweimal und 'y' und 'i' je einmal benutzt. Wenn der Graph komplett aufgebaut ist, werden die jeweiligen Zähler an den Übergängen durch die Summe aller Zähler der vom zugehörigen Zustand weggehenden Übergänge geteilt. Pfade entlang häufig vorkommender Namen können dadurch mit höheren Wahrscheinlichkeiten verbunden werden, daß man einzelne Namen mehrmals in den Graphen einfügt. Damit ändert sich die Struktur nicht, wohl aber die Zähler und somit die Wahrscheinlichkeiten. Wir bauen den Automaten aus dem Karlsruher Telefonbuch auf, in dem sich z. B. 252 Einträge für 'Maier' und nur einer für 'Peil' befinden. $P(S)$ berechnet sich als Produkt der auf dem Pfad für S auftretenden Übergangswahrscheinlichkeiten.

Bei Benutzung eines minimalen Graphen, kann es vorkommen, daß die Wahrschein-

[‡]Ein ähnliches Prinzip verwendet Knuth in seiner *trie-search*. Siehe hierzu [Knuth73, S. 481-486]).

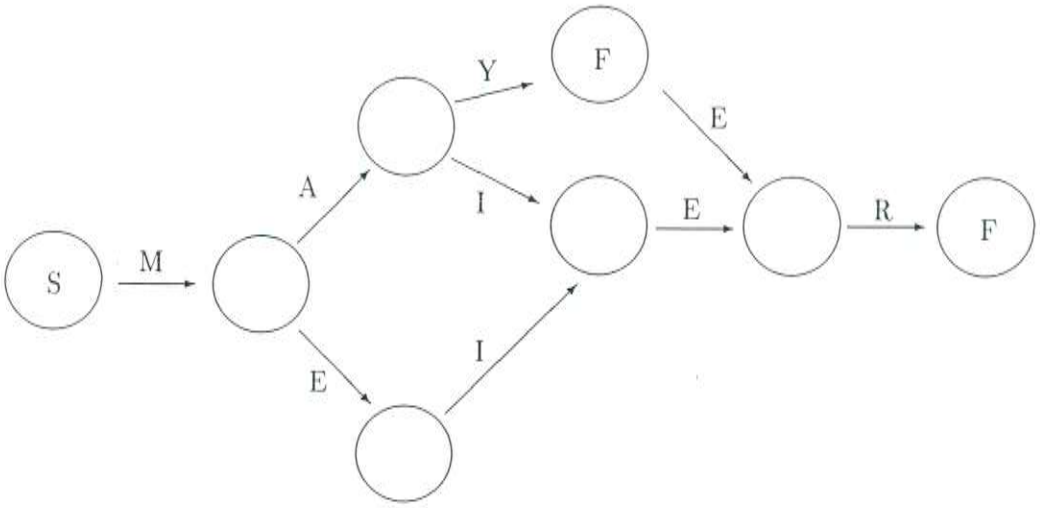

Abbildung 2.7: Modell für May, Mayer, Maier, Meier als minimaler Graph.

lichkeiten nicht mehr korrekt sind. In obigem Beispiel würden keine Fehler entstehen, die Pfade durch den minimalen Graphen haben die selben Wahrscheinlichkeiten wie im Baum. Bei einigen Konstellationen jedoch ergeben sich Schwierigkeiten. Wenn man beispielsweise die Sätze ABC, ABD, XYC, und zweimal XYD nimmt, entstehen zwei Teilgraphen, die, wenn man sie zusammenfügt, indem man die Zähler der von beiden Teilgraphen benutzten Übergänge zusammenzählt, einen fehlerhaften Gesamtgraphen erzeuegn. In Abbildung 2.8 sind im oberen Teil die beiden Teilgraphen und unten das Resultat dargestellt.

Das Problem entsteht in dem Zustand, von dem der C- und der D-Übergang ausgehen. Die dort abgelegten Wahrscheinlichkeiten stimmen weder, wenn man über den A-B Pfad, noch wenn man über X-Y kommt. Sie stellen einen Mittelwert zwischen den beiden Pfaden dar. Mit unseren Test- und Trainingsdaten sind die entstehenden Fehler jedoch so gering, daß statt der theoretisch korrekten Baumlösung die minimale Graphenlösung gewählt wurde. Von den 1316 Sätzen des Testsets ändert sich $P(S)$ in weniger als 5% der Fälle und dann nur gering. Die Erkennungsraten sind in beiden Fällen identisch. Die Perplexität (siehe nächster Abschnitt) ändert sich erst ab der dritten Nachkommastelle.

Silence Modellierung

Bei der Spracherkennung taucht immer das Problem auf, daß zwischen den einzelnen Wörtern eines Satzes eine mehr oder weniger lange Pause liegen kann. Wenn man diese nicht gezielt behandelt, kommt es mitunter zu beträchtlichen Fehlern bei der Erkennung. Es besteht die Möglichkeit, die Leerräume in einem Vorverarbeitungsschritt herauszuschneiden und dem Erkenner 'saubere' Daten zu liefern. Die Stilleperioden lassen sich relativ sicher erkennen, allerdings sind Fehler nicht auszuschließen. Probleme

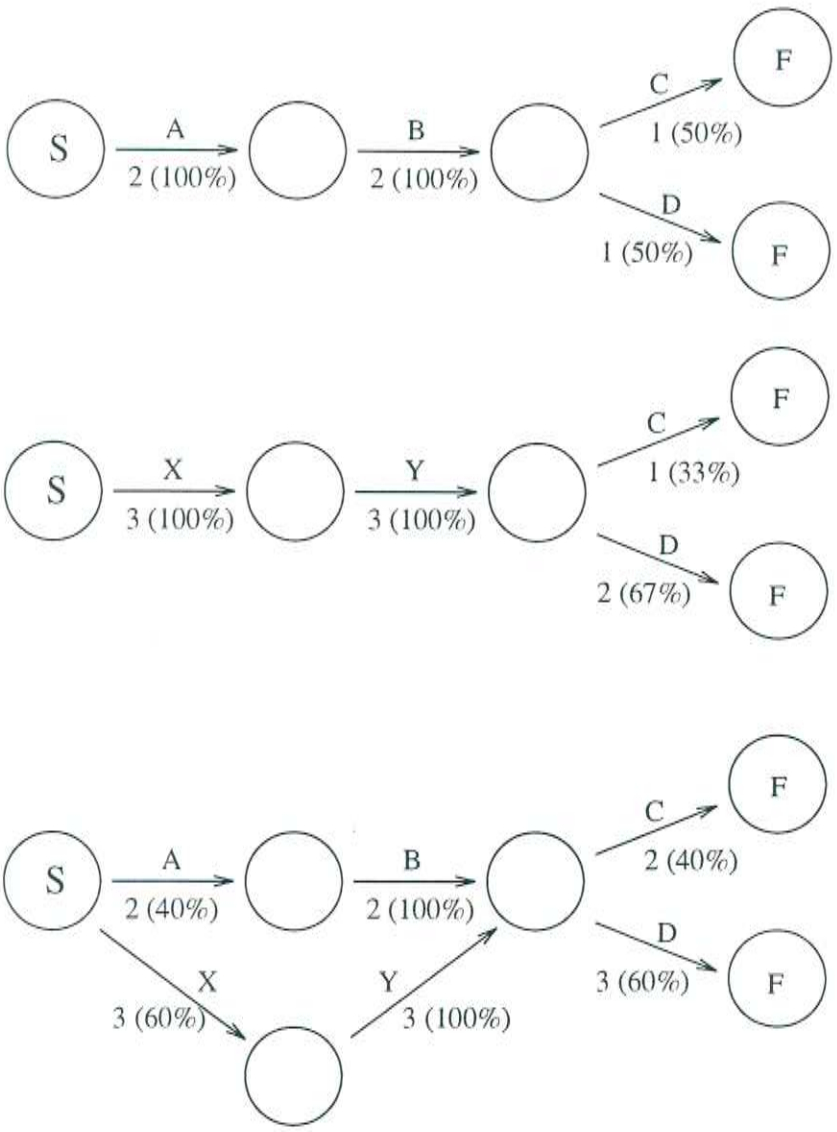

Abbildung 2.8: Fehlerhafte Wahrscheinlichkeiten beim Zusammenfügen der beiden Teilgraphen. An den Kanten stehen die Zähler und in Klammern die Wahrscheinlichkeiten.

entstehen z. B. beim Buchstabieren von '*h*', welches leicht mit einem '*a*' verwechselt werden kann, wenn vom Anfang zu viel Stille weggeschnitten wird. Eine elegantere Lösung besteht darin, die Erkennung der Pausen mit der Buchstabenerkennung verwoben laufen zu lassen. Dabei wird die Stille als normales, gültiges Wort zugelassen und trainiert und in das Sprachmodell mit eingebaut. Dieses wird so modifiziert, daß nach und vor jedem Buchstaben eine Pause liegen darf, aber nicht muß (Abbildung 2.9).

Ein Problem stellt die Zuweisung von Wahrscheinlichkeiten an die silence-Übergänge dar. Es bieten sich zwei Lösungen an: einen festen Wert für alle silence-Übergänge oder die Berechnung eines speziellen Wertes für jeden einzelnen silence-Übergang. Letzteres

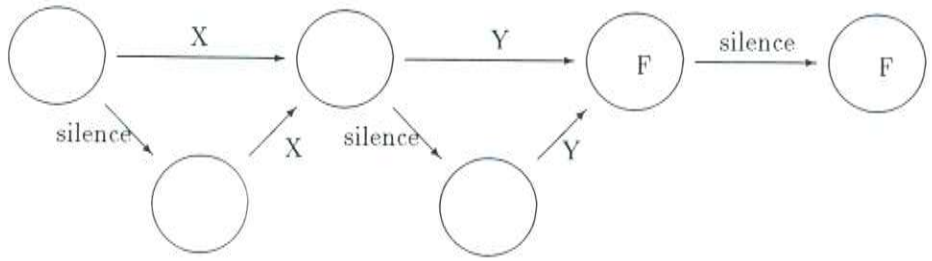

Abbildung 2.9: Optionale silences im Sprachmodell.

sollte von der Zahl oder Verteilung der Übergänge abhängen. Man könnte $1/n$ nehmen, wobei n die Zahl der von dem entsprechenden Zustand weggehenden Übergänge ist (inklusive des silence-Übergangs). Gute Ergebnisse ließen sich erzielen, indem die Wahrscheinlichkeiten etwas größer, nämlich $2/(n+1)$ gewählt wurden.

Aussprachevarianten

Wenn in einem Namen hintereinander 2 gleiche Buchstaben vorkommen, wird als Variante auch zugelassen, daß das Wort 'doppel' vorangestellt werden kann (Abbildung 2.10).

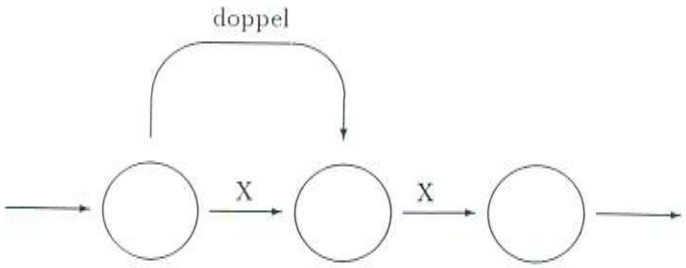

Abbildung 2.10: Aussprachevariante für doppelte Buchstaben.

Für einige 'Buchstaben' des deutschen Alphabets existieren verschiedene Aussprachevarianten. Es sind dies 'scharf-ß', 'eß-zett' und 'scharfes-ß' und die beiden Varianten 'Strich' oder 'Bindestrich', die manchmal in Doppelnamen vorkommen. Für jedes dieser Worte wird ganz normal trainiert und nachdem der Grammatik-Graph aufgebaut ist, wird jeder 'ß'-Übergang, wie in Abbildung 2.11 zu sehen, durch seine Varianten ersetzt. Mit 'Bindestrich' und 'Strich' wird äquivalent verfahren. Die Übergänge der Varianten werden mit Wahrscheinlichkeiten belegt, die der ursprünglichen Wahrscheinlichkeit geteilt durch die Zahl der Varianten entsprechen.

Mit den 26 Buchstaben des Alphabets, den 3 deutschen Umlauten, den 3 Aussprachevarianten von 'ß' und der Spezialworte 'Bindestrich', 'Strich', 'doppel' und silence

kommt man auf insgesamt 36 Worte, die das System erkennen muß und für die Wortmodelle erstellt werden müssen.

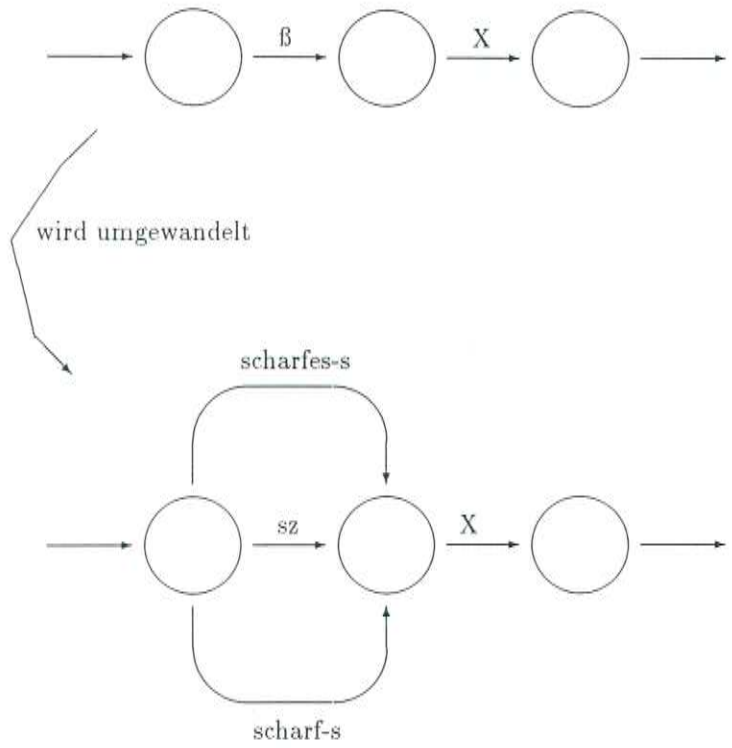

Abbildung 2.11: Ersetzen von ß in seine 3 Aussprachevarianten.

2.3.2 Perplexität

Es existiert ein Maß für die Qualität einer Grammatik. Es basiert auf der Entropie H aus der Informationstheorie, welche wie folgt definiert ist:

$$H := -\sum_{i=1}^{L} P(w_i) log P(w_i) \tag{2.7}$$

Die Entropie ist ein Maß für den Informationsgehalt einer Quelle, die unabhängig Worte w_i aus einem Alphabet der Größe L ausgibt. Unter zusätzlicher Gleichverteilungsannahme der w_i folgt:

$$
\begin{aligned}
P(w_i) = \frac{1}{L} \quad \Rightarrow \quad H &= -\sum_{i=1}^{L} \frac{1}{L} log \frac{1}{L} \\
&= -L \frac{1}{L} log \frac{1}{L} \\
&= -log \frac{1}{L} \\
&= log L
\end{aligned} \tag{2.8}
$$

Der Wert H stellt in diesem Fall den Informationsgehalt einer Quelle dar, welche gleichverteilt L verschiedene Worte ausgibt und ist gleichzeitig, wenn der Logarithmus zur Basis 2 genommen wird, die Zahl der Bits, die für die Codierung der ausgegebenen Daten benötigt werden. Dies ist unmittelbar einsichtig (z. B. $L = 256 \Rightarrow H = 8$). Das Codierungs-Theorem der Informationstheorie besagt, daß auch bei Nichtgleichverteilungsannahme H die Zahl der zur Codierung der aus der Quelle kommenden Worte benötigten Bits darstellt.

Durch eine Umformung kommt man auf eine andere Darstellung der Entropie für unabhängige, gleichverteilte Quellen:

$$
\begin{aligned}
H &= \frac{1}{n} \sum_{i=1}^{n} H && (Tautologie) \\
&= \frac{1}{n} \sum_{i=1}^{n} -log \frac{1}{L} && (mit\ 2.8) \\
&= -\frac{1}{n} \sum_{i=1}^{n} log P(w_i)
\end{aligned} \tag{2.9}
$$

Für ergodische Quellen, d. h. wenn die Symbole 'schön gemischt' ausgegeben werden und für große n, kann Formel 2.9 nach [JMR] auch für nichtgleichverteilte und nichtunabhängige Quellen verwenden:

$$
H = -\frac{1}{n} \sum_{i=1}^{n} log P(w_i | \langle w_1, w_2, \ldots w_{i-1} \rangle) \tag{2.10}
$$

Oftmals findet man in der Literatur auch eine andere Darstellung für H, die sich aus 2.10 durch einige weitere Umformungen erreichen läßt:

$$
\begin{aligned}
H &= -\frac{1}{n} \sum_{i=1}^{n} log P(w_i | \langle w_1, w_2, \ldots w_{i-1} \rangle) \\
&= -\frac{1}{n} log \prod_{i=1}^{n} P(w_i | \langle w_1, w_2, \ldots w_{i-1} \rangle)
\end{aligned}
$$

$$= -\frac{1}{n} log P(\langle w_1, w_2, \ldots, w_n \rangle) \tag{2.11}$$

Die Entropie wird als Basis zur Berechnung der Perplexität eines Sprachmodells benutzt. Dazu wird die *logarithmische Perplexität* eines Sprachmodells definiert durch:

$$LP := H \tag{2.12}$$

Der einzige Unterschied besteht darin, daß die Werte $P(w_i)$ in diesem Fall die Wahrscheinlichkeiten, die das dem Erkenner zugrundeliegende Sprachmodell liefert, darstellen. Die *Perplexität* letztendlich ist definiert als

$$PP := 2^{LP} \tag{2.13}$$

Normalerweise findet man die Perplexität in einer anderen Darstellung:

$$\begin{aligned} PP &= 2^{LP} \\ &= 2^{-\frac{1}{n} log P(\langle w_1, w_2, \ldots, w_n \rangle)} \qquad (mit\ 2.11) \\ &= \left[2^{log P(\langle w_1, w_2, \ldots, w_n \rangle)}\right]^{-\frac{1}{n}} \\ &= [P(\langle w_1, w_2, \ldots, w_n \rangle)]^{-\frac{1}{n}} \\ &= \left[\prod_{i=1}^{n} P(w_i | \langle w_1, w_2, \ldots w_{i-1} \rangle)\right]^{-\frac{1}{n}} \end{aligned} \tag{2.14}$$

Die Perplexität hängt also von dem Sprachmodell, von den Wahrscheinlichkeiten und von den Testdaten w_i, mit denen die Formel ausgewertet wird, ab.

Die Perplexität gibt, wenn man sich das Sprachmodell als einen Graphen vorstellt, den mittleren Verzweigungsgrad pro Knoten an. Wenn man z. B. als 'Sprachmodell' wählt, daß zu jedem Zeitpunkt jedes Wort aus einem Alphabet der Größe L möglich und gleichwahrscheinlich ist, ergibt sich mit 2.8 das offensichtlich sinnvolle Ergebnis:

$$PP = 2^{LP} = 2^{log_2 L} = L \tag{2.15}$$

Wenn für das in unserem Projekt verwendete Sprachmodell die Perplexität ausgerechnet werden soll, so ist Formel 2.14 etwas unhandlich, da es sich jetzt um eine Quelle handelt, die Sätze ausgibt, und nicht einen Strom gleichwertiger Worte. Zwischen den Worten der verschiedenen Sätze bestehen keine Abhängigkeiten mehr. Formel 2.12 bzw. 2.11 zur Berechnung der logarithmischen Perplexität ändert sich damit zu:

$$LP = -\frac{1}{n} \sum_{i=1}^{\#s} log P(S_i) \tag{2.16}$$

Hierbei ist n nach wie vor die Zahl der Wörter aus dem Testset und $\#s$ die Zahl der Sätze. Nachdem LP berechnet wurde, wendet man 2.13 an, um daraus die Perplexität zu erhalten. $P(S_i)$ berechnet sich nach 2.6 von Seite 13.

Pseudocode zur Berechnung der Perplexität eines Sprachmodells.

$SUM = 0$
$n = 0$
for $i = 1$ to $test\_set\_size$
    Sei S_i der i-te Satz des Testsets
    berechne $P(S_i)$ mit Formel 2.6
    $SUM = SUM + logP(S_i)$
    $n = n + sentence\_length(S_i)$
end_for
$LP = -\frac{1}{n}SUM$
$PP = 2^{LP}$

Die Perplexität an sich ist schon ein recht gutes Maß für die Schwierigkeit, die das Spracherkennungssystem zu bewältigen hat. In unserem Fall, der Erkennung buchstabierter Namen, sind die Perplexitäten relativ klein (≈ 5, siehe Tabelle 3.4 auf Seite 39). Die Aufgabe ist allerdings doch relativ schwierig, da die Buchstaben des Alphabets sehr leicht verwechselbar sind, z. B. *b*, *d*, *e*, *g*, oder *l*, *r*, *m*, *n*. Unser Buchstabenerkenner erkennt im Schnitt jeden zehnten Buchstaben falsch.

Kapitel 3

Der Buchstabiererkenner

Ausgehend von der ersten Version des Erkenners ohne Grammatik werde ich in diesem Kapitel die verschiedenen vorgenommenen Erweiterungen darlegen. Den Ergebnissen der Testläufe, die sich jeweils an den Enden der einzelnen Abschnitte befinden, lag immer das gleiche neuronale Netz und das gleiche Testset zugrunde – sie sind somit gut vergleichbar. Bei den Tests mit Grammatik wurden zwei verschiedene Grammatiken benutzt. Sie sind mit '*kleine Grammatik*' und '*große Grammatik*' bezeichnet, wobei erstere ungefähr 1000 und letztere 32000 Namen aus dem Karlsruher Telefonbuch enthält. Genaue Beschreibungen des Testsets und der Grammatiken befinden sich im Anhang.

In den Tabellen, die die Testergebnisse beinhalten, befinden sich immer zwei Einträge für die Erkennungsrate. Der erste Eintrag ist mit '*Worte*' bezeichnet und gibt die Worterkennungsrate an. Die Zahl berechnet sich, indem die Einfüge-, Auslaß- und Ersetzfehler aufsummiert und dann von 100% abgezogen werden. Die Satzerkennungsrate, in den Tabellen mit '*Sätze*' bezeichnet, gibt an, wieviele Sätze (hier Nachnamen) vollkommen richtig erkannt wurden.

3.1 Die alte Version ohne Grammatik

Die erste Version des dieser Arbeit zugrundeliegenden kontinuierlichen Buchstaben-Erkenners wurde von Hermann Hild implementiert und in [HW92] erläutert. Es handelt sich um ein hybrides MS-TDNN-System, das aus einem TDNN und einer DTW-Schicht besteht. Es verbindet die Vorteile des TDNNs, das sich als ausgezeichneter Phonem-Erkenner erwiesen hat [WHH+89], und des DTWs, mit dem ein nichtlineares Zeitalignment durchgeführt wird, um Worte, die als Sequenz von Phonemen dargestellt sind, zu erkennen [HW93].

3.1.1 Arbeitsweise

In Bild 3.1 ist das Prinzip dargestellt, nach dem das MS-TDNN funktioniert. Die unteren 3 Schichten stellen den TDNN-Teil des Sytems dar. Zwischen ihnen befinden

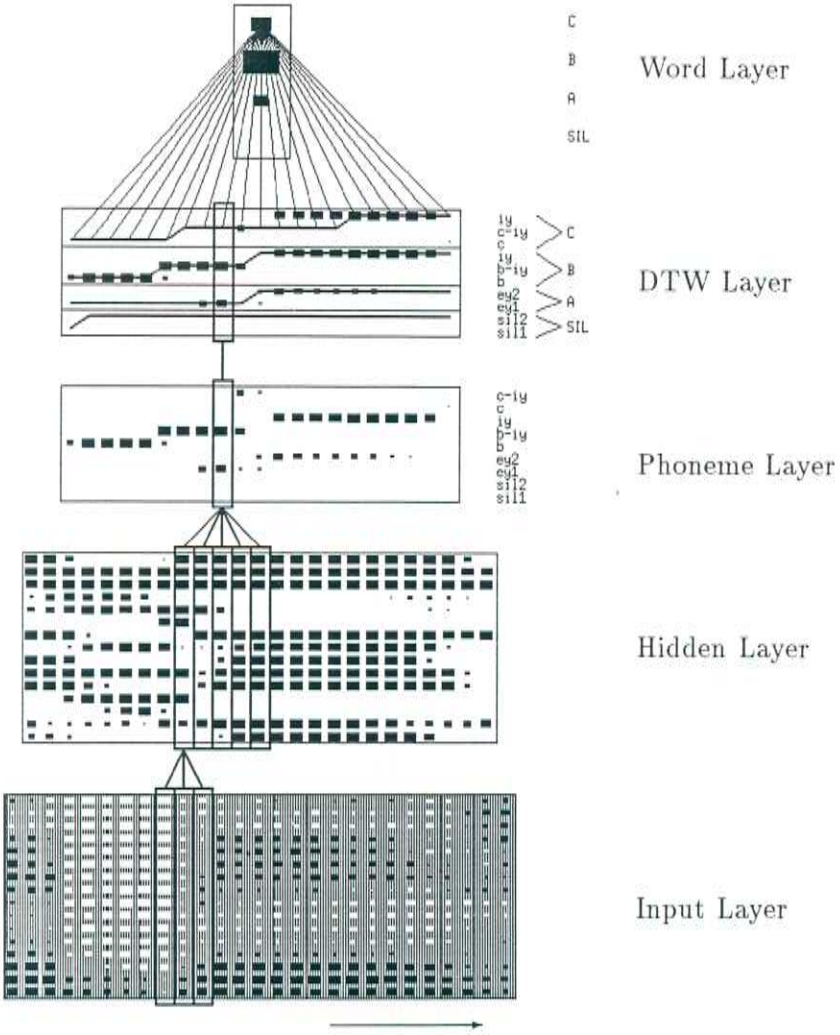

Abbildung 3.1: Ein Auschnitt aus dem MS-TDNN zur Einzelworterkennung von *SIL*ence, *A*, *B* und *C* (nach [HW93.2]).

sich jeweils die gewichteten Verbindungen von einem Zeitfenster der unteren Schicht zu einer Spalte der darüberliegenden Schicht. Die Fenster werden jeden Zeitschritt (hier 10 msec), um je einen Frame vorwärtsverschoben. Das TDNN wird so trainiert, daß in der dritten Schicht (Phoneme Layer) Bewertungen für Phonem-Hypothesen zu den jeweiligen Zeitpunkten ausgegeben werden. Im Bild stellen größere Rechtecke höhere Bewertungen dar.

Im DTW-Layer sind aus den Phonemen Wortmodelle gebildet, z. B. das Wort *B* aus den Phonemen *b*, *b-ih* und *ih* (englische Aussprache). Die Aktivierungen aus dem Phoneme-Layer sind für die jeweiligen Phoneme in den DTW-Layer kopiert und werden als Basis für das 'dynamic time warping' benutzt, welches den am höchsten bewerteten Pfad sucht und das zugehörige Wortmodell (in diesem Fall das *B*) als Ergebnis ausgibt.

Nach diesem Verfahren können Einzelworte erkannt werden. Wenn kontinuierliche Sprache erkannt werden soll, muß, wie in [Ney90] beschrieben, zusätzlich hierzu noch

erlaubt werden, daß Pfade vom Ende des einen Wortmodells zum Anfang eines anderen gehen dürfen (Abbildung 3.2).

Pseudocode für Erkennung ohne Grammatik

```
for t = 0 to end
    for all Wortmodelle w
        Berechne DTW innerhalb von w
        Erlaube, daß bester Pfad bis t − 1 in Anfangszustand von w übergehen darf.
    end
end
```

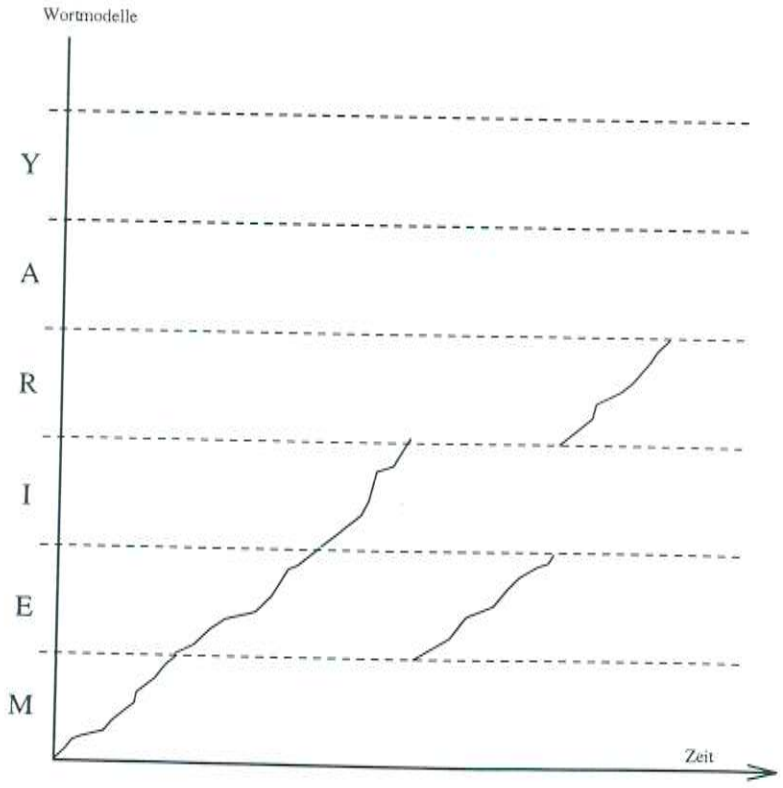

Abbildung 3.2: DTW über *Meier*. Der Pfad geht zweimal durch das *E*.

Dabei entsteht gerade bei der Buchstabenerkennung ein Problem, da viele Kombinationen existieren, wo Wortenden und Anfänge sehr ähnlich klingen, und nicht gut zu erkennen ist, ob eine Wortgrenze vorlag oder nicht, was zu relativ hohen Fehlerraten führen kann. So ist die Sequenzen '*B E*' leicht mit '*B*' oder '*A A*' mit '*A*' verwechselbar. Eine Abhilfe besteht darin, ein geeignetes Modell für die Wortlänge zu finden. Eine minimale Wortlänge kann relativ einfach durch eine Vervielfachung der Phoneme in den

Wortmodellen erreicht werden. Man kann das *B* statt durch *b*, *b-ih*, *ih*, wie oben beschrieben, durch die Sequenz *b*, *b*, *b*, *b-ih*, *b-ih*, *b-ih*, *ih*, *ih* und *ih* modellieren und setzt dadurch minimale Wortlänge auf 90 msec, da in jedem Phonem-Zustand mindestens 10 msec verweilt werden muß.

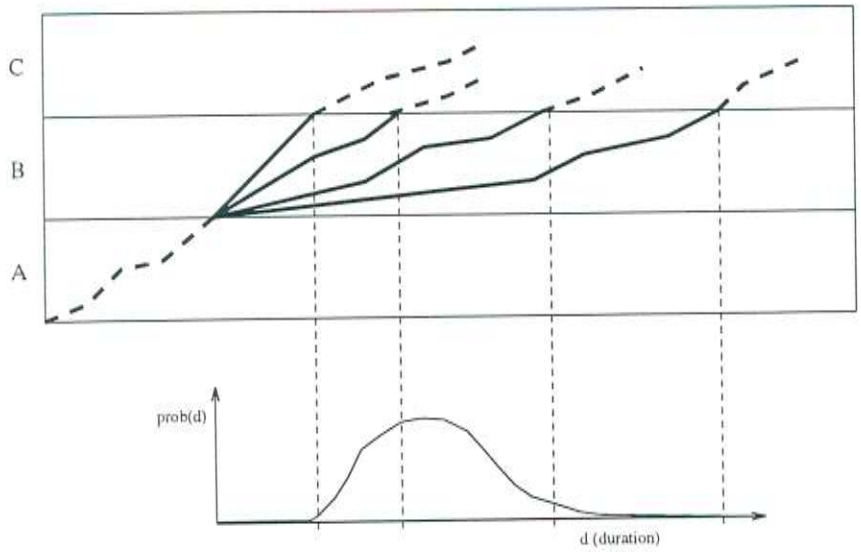

Abbildung 3.3: Zeitabhängige Strafen (nach [HW93.2])

Weiterhin kann man zu kurze oder zu lange Pfade durch ein Wortmodell durch eine zeitabhängige Strafe zu unterdrücken versuchen (Bild 3.3). Dazu wird ein wort- und zeitabhängiges Strafmaß nach der Formel $Pen_w(d) = log(k + prob_w(d))$ eingeführt [HW93]. $prob_w(d)$ ist die aus den Trainingsdaten ermittelte Wahrscheinlichkeit, für eine Dauer d für daß Wort w. Eine kleine Konstante k dient dazu, Null-Wahrscheinlichkeiten auszuschließen. Dieses Strafmaß wird zu dem Pfad aus dem DTW addiert, wenn ein Wortübergang stattfinden soll: $score = score + \lambda \cdot Pen_w(d)$. Die Konstande λ gibt an, wie stark der Einfluß dieses Wortlängenmodells sein soll. Es besteht kein einfacher mathematischer Weg, um den Einfluß von diesem λ auf die Fehlerraten zu berechnen [HW93]. Wahlweise kann man auch λ auf Null setzen und nur die minimale Wortlänge, die sich implizit aus der Zahl der Zustände in den DTW-Modellen ergibt und die maximale Wortlänge, einer Zahl die aus den Trainingsdaten gewonnen wird, indem für jedes Wort die längste Aussprechzeit gemerkt wird, verwenden. Nur Pfade innerhalb dieses Zeitintervalls werden dann als gültig erklärt (siehe Bild 3.4).

3.1.2 Ergebnisse

Das implementierte Verfahren liefert auf dem deutschen Alphabet eine Wortfehlerrate von 9.9%. Dies ist an sich kein sehr schlechter Wert. Wenn man allerdings in Betracht zieht, daß das Programm dazu dienen soll, buchstabierte Nachnamen zu erkennen, ist das nicht hinreichend, da ganz offensichtlich bei einer durchschnittlichen Namenslänge

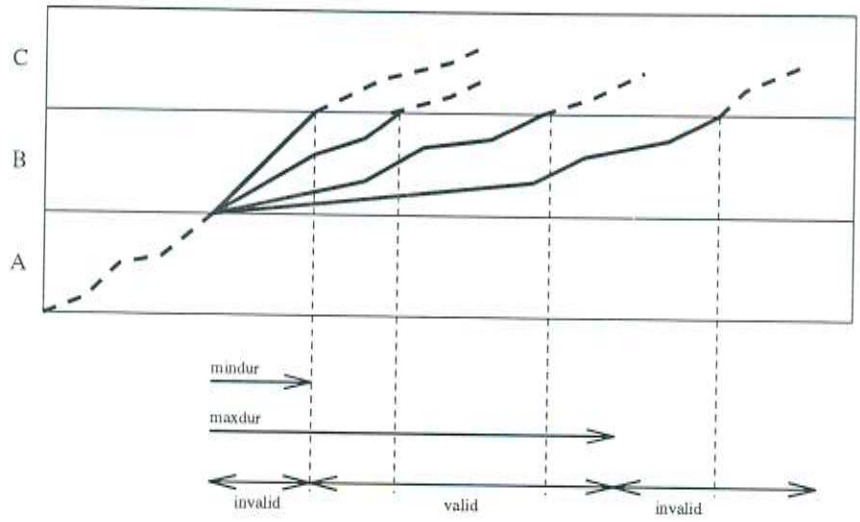

Abbildung 3.4: Gültiges Wort-Zeitintervall

von 5 oder 6 Buchstaben und einem Erkenungsfehler bei jedem zehnten Buchstaben, etwa jeder zweite Name falsch erkannt wird. Tatsächlich wird auch eine sehr schlechte Satzfehlerrate von 43.2% erreicht. Die Erkennungsgeschwindigkeit ist allerdings sehr hoch: Auf einer HP735 im Schnitt 0.4 Sekunden für einen 3 Sekunden langen Satz. Für eine Zusammenfassung der Ergebnisse siehe Seite 37.

3.2 Die erste Version mit Grammatik

Die erste Erweiterung des soeben vorgestellten Systems um ein Sprachmodell wurde von Hermann Hild vorgenommen und von mir noch in einigen Details modifiziert.

3.2.1 Arbeitsweise

Bei diesem Ansatz liegt das Sprachmodell in Form eines Automaten, wie z. B. auf Seite 15 gezeigt, vor. Das bisherige Verfahren ohne Grammatik wird im großen und ganzen beibehalten, mit dem Unterschied, daß es nun nicht mehr nur 36 Wortmodelle (die Buchstaben des Alphabets, Umlaute und einige Spezialzeichen) gibt, sondern so viele, wie in dem Automaten Übergänge existieren. Diese Worte, die mit den Übergängen verbunden sind, werden, obwohl es tatsächlich nur maximal 36 verschiedene geben kann, als vollkommen unabhängig betrachtet und bei den zwischen-Wortmodell-Übergängen innerhalb der DTW Berechnung werden nur Übergänge entsprechend des Sprachmodells erlaubt.

Graphisch veranschaulichen kann man sich diesen Unterschied durch einen Vergleich der Erkennung einer Buchstabensensequenz ohne (Abbildung 3.2, Seite 25) und mit einer Grammatik (Sprachmodell in Abbildung 3.5, DTW in Abbildung 3.6). Die

Grammatik erlaubt, nachdem ein Pfad durch das M gelaufen ist, als Nachfolger nur A oder E_2. Nachdem sich für das E entschieden wurde, ist die weitere Wortfolge schon festgelegt. Es müssen I_2, danach E_2 und schließlich R folgen. Man sieht, daß durch das Verwenden der Grammatik der Suchraum erheblich eingeschränkt wird. Bei der Version ohne Grammatik müßte bei jedem Wortende jeder Buchstabe als möglicher Nachfolger in Betracht gezogen werden. Man vergleiche auch folgenden Pseudocode mit dem der Version ohne Grammatik von Seite 25.

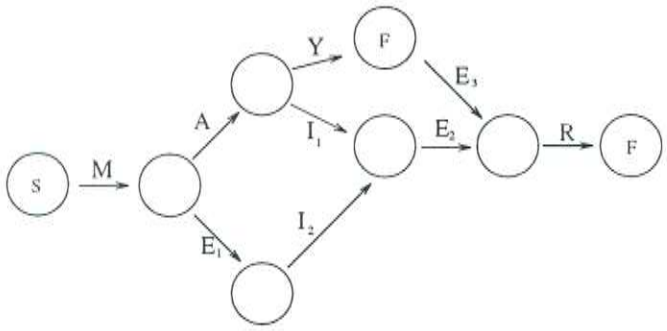

Abbildung 3.5: Sprachmodell für verschiedene '*Meiers*'.

Pseudocode für alten Grammatikmodus

```
for t = 0 to end
    for all Übergänge w aus Grammatik
        Berechne DTW für w
    end
    for all Übergänge w aus Grammatik
        for all Nachfolger von w, die Grammatik erlaubt
            falls Pfad von w zur Zeit t − 1 nach Nachfolger besser als
                aktueller Pfad, dann diesen nehmen.
        end
    end
end
```

Ein klarer Vorteil dieser Lösung ist die Einfachheit der Implementierung. Es muß der Algorithmus ohne Grammatik nur leicht modifiziert werden: Die Initialisierung mit den vermehrten Wortmodellen und die Berücksichtigung der Grammatikübergänge an den Enden der Wortmodelle während des DTW-Laufs. Zusätzlich muß für jeden Pfad, der in den Anfangszustand eines Wortmodells übergeht, gemerkt werden, wo er herkam. Im Modus ohne Grammatik reichte es aus, zu jedem Zeitpunkt den einen besten Pfad, der an einem Wortende die höchste Bewertung hat, zu speichern. Dadurch kennt man implizit den Vorgänger, wenn zu diesem Zeitpunkt tatsächlich ein Wortübergang stattgefunden haben soll.

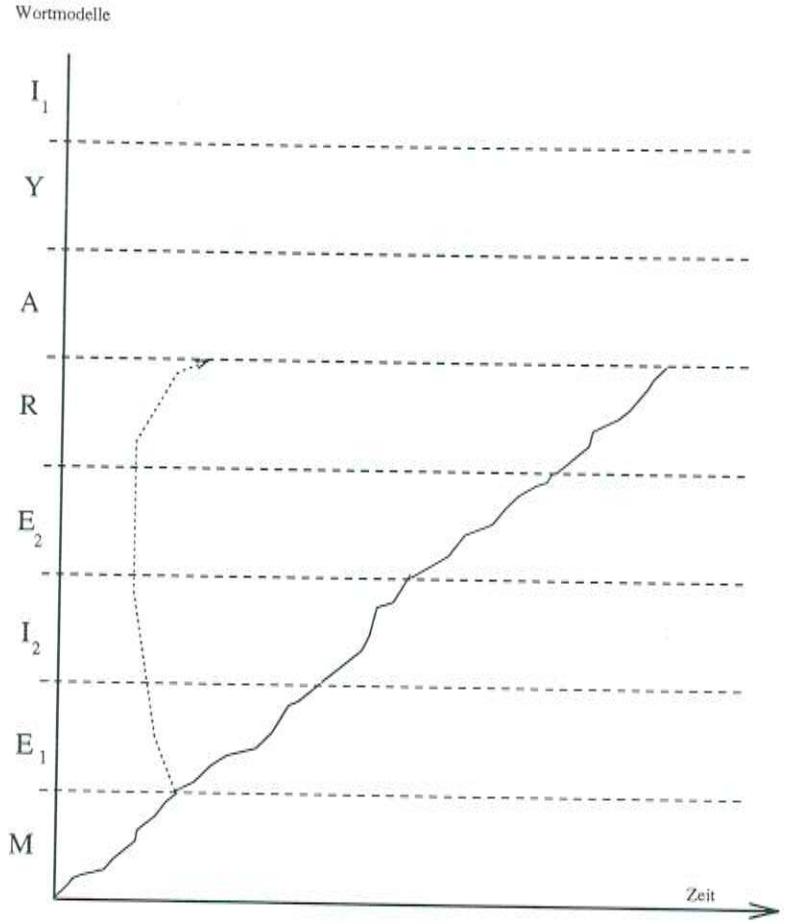

Abbildung 3.6: Einschränken des Suchraums durch Verwenden einer Grammatik. Erste Implementierung mit Vervielfachung von Wortmodellen (hier E und I).

Die Laufzeit und der Speicherplatzverbrauch verhalten sich proportional zur Zahl der Wortmodelle. Bei der kleinen Version der Grammatik, welche 1300 Nachnamen enthält, besteht die Grammatik aus ungefähr 8000 Übergängen* und man verbraucht dann insgesamt fast 50 MB für die anfallenden Datenstrukturen. Damit ist oftmals der verfügbare Speicher erschöpft oder der Rechner muß sehr häufig Daten auf Platte auslagern, was mitunter zu langen Laufzeiten führt.

Um Laufzeit zu sparen, habe ich einen Aktivierungs/Deaktivierungs-Mechanismus eingebaut. Es werden Wortmodelle, bei denen zu einem Zeitpunkt alle Pfade um einen gewissen Betrag schlechter sind, als der beste, komplett deaktiviert und erst wieder aktiviert, wenn ein Übergang von einem anderem Wortmodell in dieses Wortmodell erfolgt. Wenn zu Beginn nur der Startzustand aktiv ist und dann sukzessive die möglichen Nachfolger aktiviert werden (ohne jemals wieder deaktiviert zu werden) wird die Laufzeit um 33% reduziert. Mit zusätzlicher Deaktivierung konnte eine Laufzeitreduk-

*inklusive der optionalen *silence*-Übergänge

tion von 59% erreicht werden. Typische Verläufe aktivierter Wortmodelle finden sich in Diagram 3.7.

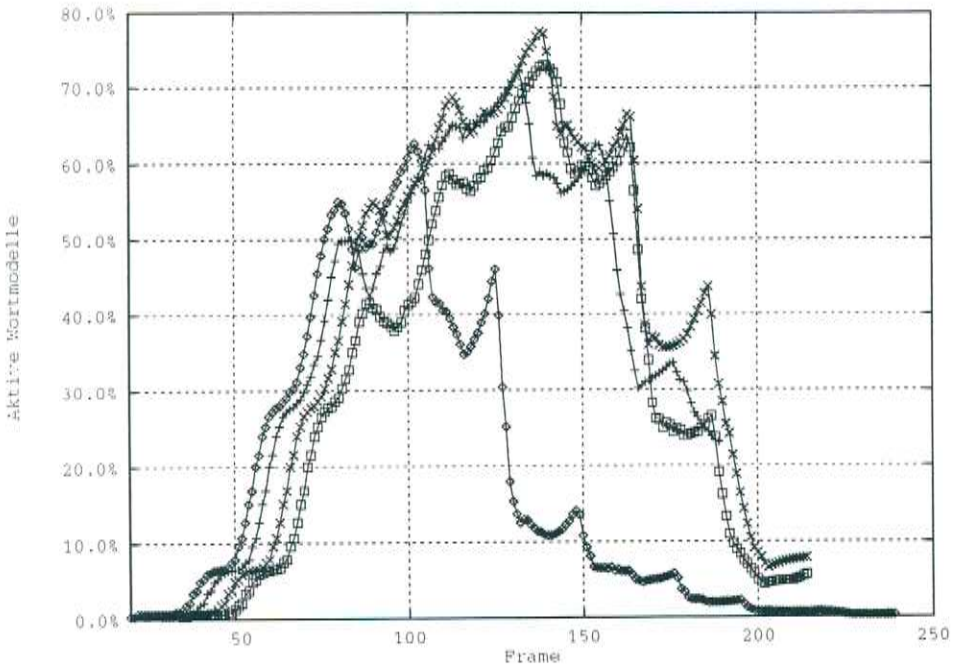

Abbildung 3.7: Prozentsatz der aktiven Wortmodelle über der Zeit. Kurven von vier verschiedenen Beispielsätzen.

3.2.2 Ergebnisse

Ein Test mit der großen Grammatik, die alle Karlsruher Nachnamen enthält, war nicht möglich, da der Speicher hier bei weitem nicht ausgereicht hätte (fast 1 GB wäre benötigt worden!). Mit der kleinen Grammatik werden sehr gute Ergebnisse erzielt: 99.5% Worterkennungsrate und 98.4% Satzerkennungsrate, wobei vor allem letzteres eine drastische Steigerung gegenüber der Version ohne Grammatik darstellt. Pro Satz werden auf einer HP im Schnitt 5 Sekunden benötigt, was noch eine vertretbare Zeit ist. Der Speicherplatzbedarf ist allerdings enorm und bei größeren Grammatiken hat man Probleme, das ganze überhaupt auf einem Rechner zum Laufen zu bringen. Zusammenfassung der Ergebnisse siehe Seite 37.

3.3 Die neue Version mit Grammatik

Weil es mit der ersten Lösung nicht möglich war, größere Grammatiken zu benutzen, entstand der Wunsch nach einem neuen Ansatz. Die naheliegende Idee ist, wie bei der Version ohne Grammatik, nur eine DTW über die tatsächlich vorhandenen (hier

36) Wortmodelle zu berechnen und etwas mehr Arbeit in die Vorgehensweise bei den Wortübergängen zu stecken.

3.3.1 Wie es nicht funktioniert

Da bei dem DTW zu jedem Zeitpunkt für jeden Pfad genau bekannt ist, wo er herkommt, könnte man bei den Wortenden den bisherigen Pfad als Präfix des zu erkennenden Satzes ansehen. Damit kann man in der Grammatik nachsehen, welche zulässigen Nachfolger möglich sind und dies bei der Berechnung der weiteren Pfade berücksichtigen. Diese Lösung wäre sehr schnell – sie würde nicht wesentlich mehr Zeit brauchen, als die Suche ohne Grammatik. Leider funktioniert sie nicht. Das Problem ist, daß bei Wortübergängen, wie auch zu jedem anderem Zeitpunkt, nur jeweils ein Pfad gemerkt wird. Deshalb kann es vorkommen, daß bei einer einmal begangenen Fehlerkennung keine spätere Korrektur möglich ist, die den richtigen Pfad wieder herstellen könnte. In Bild 3.8 ist dieser Fall dargestellt. Hierbei laufen 2 Pfade von A und B nach C. Der Pfad von A sei etwas schlechter als der andere. Damit wird er vollkommen eliminiert und am Ende von C kann, wenn die Grammatik nach der Folge B – C beispielsweise keinen Nachfolger zuläßt, nicht auf den Pfad A – C zurückgegriffen werden.

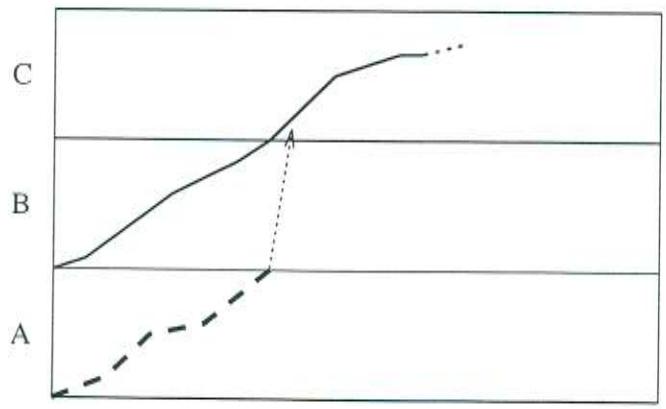

Abbildung 3.8: Der von A kommende (schlechtere) Pfad wird eliminiert, wenn er in den besten Pfad, der von B kommt, einmündet.

3.3.2 Generieren von Hypothesen

Man muß deshalb einen anderen Weg gehen und sich zu jedem Zeitpunkt mehrere Alternativen merken. Die Lösung, die ich hier vorstelle und implementiert habe, basiert darauf, daß zu jedem Zeitpunkt eine Liste mit den n besten Hypothesen aufgebaut wird. Eine Hypothese in diesem Sinn ist ein Pfad inklusive einer Bewertung. Zu jedem Zeitpunkt t wird diese Hypothesenliste 'vorwärtspropagiert'. Dies bedeutet, daß für alle Einträge geprüft wird, welche Nachfolgewörter möglich sind. Dann wird für die Nachfolger N eine komplette DTW für den Zeitraum t bis $t + maximale\_dauer(N)$

berechnet, was quasi eine Einzelworterkennung für dieses Wort für diesen Zeitraum bedeutet. Anschließend wird zur Bewertung für den bisherigen Pfad die DTW-Bewertung des Nachfolgewortes addiert (jeweils für die Zeitpunkte t bis $t + maximale\_dauer(N)$), was die Bewertung für den Pfad plus dem Nachfolger ergibt. Dieser Wert wird dann in die Hypothesenliste des entsprechenden Zeitpunktes einsortiert. Es ist klar, daß die Größe, die man für die Hypothesenlisten wählt, einen entscheidenden Einfluß auf die Fehlerraten (und auch auf die Laufzeit) hat. Zum besseren Verständnis des Verfahrens mögen hier der Pseudocode und Abbildung 3.9 dienen.

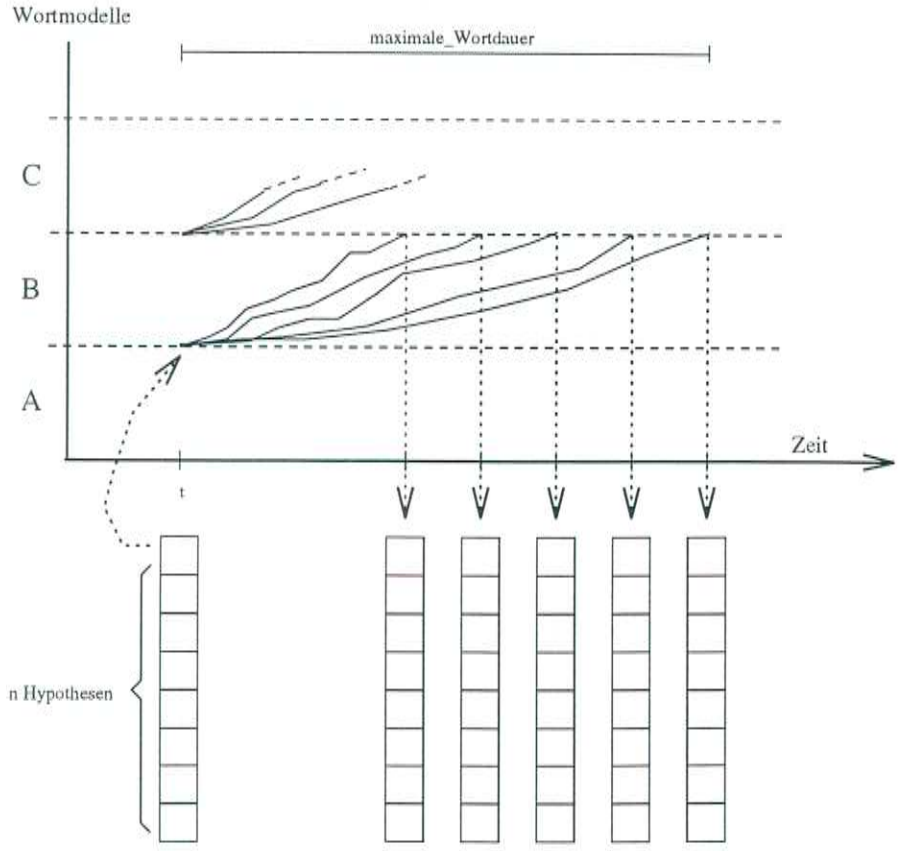

Abbildung 3.9: Die Hypothesen zur Zeit t werden nach Berechnen der DTW der Nachfolgewörter in die Hypothesenlisten von $t + minimale\_wortdauer$ bis $t + maximale\_wortdauer$ vorwärtspropagiert.

Pseudocode für neuen Grammatikmodus

schreibe $START$ in $Hypothesenliste[0]$
for $t = 0$ to end
  for all $h \in Hypothesenliste[t]$
    for all $Nachfolger$ von h, die Grammatik erlaubt
      for $tt = t$ to $t + max\_dauer(Nachfolger)$
        berechne $DTW(Nachfolger, t, tt)$
      end
      for $tt = t + min\_dauer(Nachfolger)$ to $t + max\_dauer(Nachfolger)$
        $neuer\_score = alter\_score(h) + DTW(Nachfolger, t, tt)$
        $neue\_hypothese = h$ mit $Nachfolger$
        sortiere $(neue\_hypothese, neuer\_score)$ in $Hypothesenliste[tt]$ ein
      end
    end
  end
end

Bisher wurde nur ganz allgemein von Pfaden bzw. Pfadhypothesen gesprochen, die in die Hypothesenlisten eingetragen werden. Ein Eintrag in eine solche Liste besteht aus einem 4er-Tupel, welches den aktuellen Übergang aus der Grammatik, den Vorgängerübergang, den Index t der Vorgängerhypothesenliste und die akkumulierte Bewertung des Pfades enthält. So kann am Ende der Generierung der Hypothesenlisten der beste Pfad restauriert werden, indem der beste Eintrag der letzten Liste genommen wird und dann nach folgendem Verfahren der Pfad aus den Einträgen der alten Hypothesenlisten zusammengebaut wird:

Pseudocode für das Finden des besten Pfades aus den Hypothesenlisten

$pfad$ = leerer pfad
$eintrag$ = bester Eintrag aus $Hypothesenliste[letzer\_frame]$
schleife
  $aktuell$ = aktueller Übergang aus $eintrag$
  $vorgänger$ = Vorgänger-Übergang aus $eintrag$
  $vorgänger\_frame$ = t-backpointer aus $eintrag$
  $pfad = pfad + aktuell$
  $eintrag$ = der Eintrag aus $Hypothesenliste[vorgänger\_frame]$
    mit aktueller Übergang == $vorgänger$
bis $vorgänger\_frame == 0$
gib $pfad$ rückwärts aus

Wenn beim Einsortieren festgestellt wird, daß für die gleiche Hypothese, d. h. den gleichen Übergang aus der Grammatik schon ein Eintrag existiert, so bleibt nur der bessere von beiden erhalten. Wenn eine Liste voll ist, fallen die schlechtesten Einträge heraus.

Aus der Art und Weise, wie Pfade abgelegt werden, folgt, daß auch zwei unterschiedliche Pfade, die zu irgendeinem Zeitpunkt auf den selben Übergang aus der Grammatik treffen, zusammengeworfen werden und nur der bessere von beiden abgelegt wird. Dies ist aber kein Problem und sogar vorteilhaft, da der schlechtere Pfad nie mehr die Chance haben wird, den anderen zu überholen, da ab diesem Zeitpunkt für beide Pfade der gleiche Weg bis zum Ende eingeschlagen wird und somit auch für beide Pfade die gleichen Bewertungen zu deren aktuellen Bewertungen hinzukommen, so daß am Ende der momentan bessere Pfad um den gleichen Betrag besser sein wird als zu dem Zeitpunkt, an dem sich die beiden Pfade treffen. Durch dieses Zusammenwerfen von Pfaden wird also auch noch unnötige Rechenzeit vermieden. Abbildung 3.10 versucht dies zu verdeutlichen.

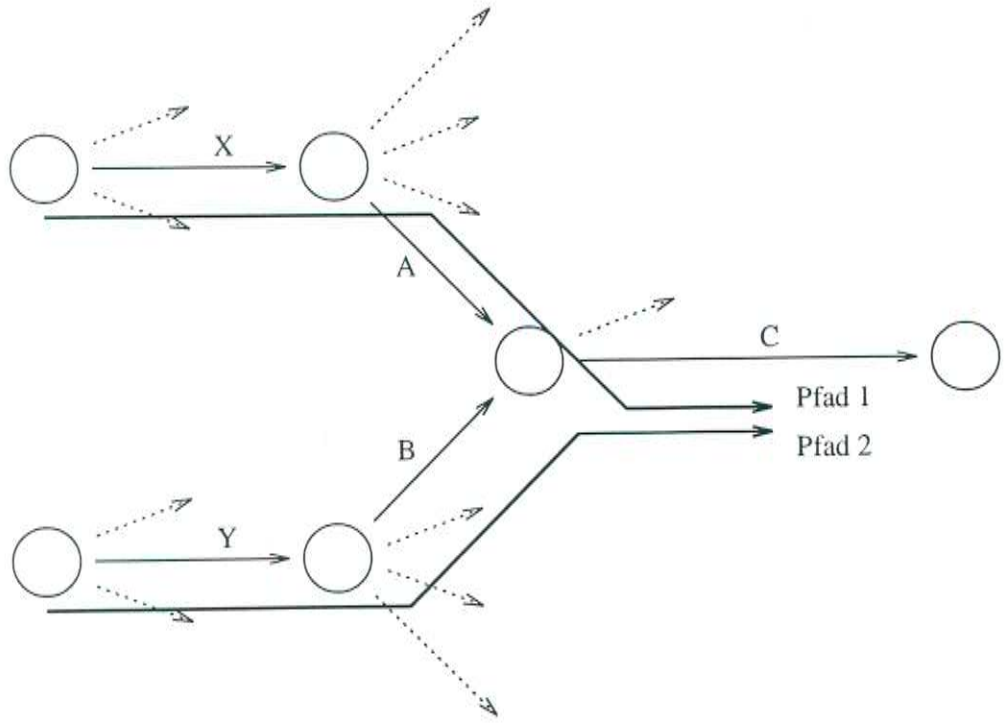

Abbildung 3.10: Nach dem Zusammentreffen der beiden Pfade nach dem A bzw. B wird es für beide Pfade die selbe Fortsetzung geben. Deshalb wird nur der bessere von beiden weiter berücksichtigt.

3.3.3 Rechenaufwand

Nachdem die Hypothesenlisten erstellt sind, wird für das Finden des besten Pfades kaum Rechenzeit benötigt. Jedoch legt ein erster Blick auf den Pseudcode von Seite 33 für die Erstellung der Hypothesenlisten die Vermutung nahe, daß der Algorithmus in dieser Form einen extrem hohen Rechenaufwand haben wird und man gezwungen ist, sich geeignete Maßnahmen auszudenken, diesen zu reduzieren.

Eine erste Verbesserung ergibt sich daraus, daß die doppelte Berechnung der DTWs entfernt wird. Bei vielen Hypothesen zu einem Zeitpunkt t wird der selbe Nachfolger erlaubt sein. Die DTW würde dann zwei- oder mehrfach berechnet. Dies kann man verhindern, indem man die DTW-Ergebnisse für den die komplette Abarbeitung eines Zeitschrittes (also einer t-Schleife) zwischenspeichert. Dazu muß obiger Pseudocode wie folgt modifiziert werden.

Vermeiden der doppelten Berechnung der DTW

```
...
for all Nachfolger von h, die die Grammatik erlaubt
    falls DTW(Nachfolger, t, t bis t+max_dauer(Nachfolger)) noch nicht berechnet
        for tt = t to t + max_dauer(Nachfolger)
            berechne DTW(Nachfolger, t, tt)
        end
    end
    ...
```

In Tabelle 3.1 sind für die 3 Versionen (neuer Grammatikmodus, alter Grammatikmodus mit Pseudocode auf Seite 28 und ohne Grammatik, mit Pseudocode auf Seite 25) die Arbeitsschritte gegenübergestellt. Man sieht, daß bei kleinen Grammatiken die alte Lösung wesentlich besser abschneidet, diese aber mit anwachsender Grammatikgröße deutlich langsamer wird. Dieses Verhalten zeigt sich bei dem neuen Grammatikmodus nicht, da hier die Grammatikgröße nur in Form der Perplexität einfließt, welche sich nicht in großem Rahmen ändert. Allerdings muß berücksichtigt werden, daß dennoch eine implizite Verlangsamung eintritt, da man bei größeren Grammatiken auch längere Hypothesenlisten wählen muß, um bessere Erkennungsraten zu erzielen.

In der Tabelle basieren die Laufzeiten für die neue Grammatikversion auf der oben beschriebenen Verbesserung. Für die einmalige Berechnung der DTW braucht man $max\_dur * J$ Schritte und muß dies für alle Nachfolger einer Hypothese, höchstens jedoch w mal ausführen. Dazu kommt für jeden Nachfolger einer Hypothese das Vorwärtspropagieren, welches jeweils $max\_dur * n$ Schritte benötigt (n für das Einsortieren in die Listen). Unter der Annahme, daß die Perplexität ein gutes Maß für den mittleren Verzweigungsgrad im Grammatikgraphen und somit für die Zahl der möglichen Nachfolger einer Hypothese ist, kommt man auf einen Aufwand von $max\_dur * J * w + max\_dur * n * pp$ für jede Hypothese zu einem Zeitpunkt. Durch

Grammatikmodus	Arbeitsschritte		
ohne	$W * J$	=	540
alt	$T * (J + pp)$	=	159 200 bzw. 3 290 000
neu	$n * max\_dur * (W * J + pp * n)$	=	2 944 000 bzw. 3 520 000

mit folgenden Bezeichnungen:

W	=	36	=	Zahl der Wortmodelle
J	=	15	=	Durchschnittliche Zahl der States pro Wortmodell
T	=	8000	=	Zahl der Übergange in der kleinen Grammatik
	bzw.	140 000	=	Zahl der Übergänge in der großen Grammatik
pp	=	4.9	=	Perplexität der kleinen Grammatik
	bzw.	8.5	=	Perplexität der großen Grammatik
n	=	40	=	Länge der Hypothesenliste
max_dur	=	100	=	Durschnittliche Maximaldauer der Wortmodelle

Tabelle3.1: Vergleich der Arbeitsschritte für einen Zeitschritt in den 3 Versionen.

Multiplikation der Formel mit n und Ausklammern von max_dur kommt man auf die in der Tabelle gewählte Darstellungform.

Neben dem Vermeiden von doppelten Berechnungen der DTW liegt der zweite Ansatzpunkt für Geschwindigkeitssteigerungen in dem innersten Schleifenschritt, dem Einsortieren. Es werden hier rein rechnerisch $O(n)$ Schritte benötigt, da man entweder eine verzeigerte Liste nehmen kann, bei der das Suchen der Position $O(n)$ Schritte braucht, oder einen Array, in dem man für das Verschieben $O(n)$ Schritte braucht. Ich benutze einen Array, in dem die Einträge sortiert gehalten werden. Beim Einsortieren können folgende Fälle auftreten:

- Ein alter Eintrag für die gleiche Hypothese existiert schon:
  - Wenn der alte Eintrag eine besserer Bewertung besitzt, muß nichts getan werden. Für jede Hypothesenliste existiert eine Hashtabelle, mit der mit dem Aufwand $O(1)$ geprüft werden kann, ob ein Eintrag für einen bestimmten Pfad existiert.
  - Hat die alte Hypothese eine schlechterer Bewertung, so muß der Eintrag ausgetauscht werden und die Liste neu sortiert werden. Im Normalfall weicht allerdings die neue Bewertung nicht drastisch von der alten ab, so daß nur sehr wenige Einträge verschoben werden müssen, manchmal auch gar keine.
- Es existiert noch kein Eintrag für die neue Hypothese:
  - Die Bewertung für die neue Hypothese ist schlechter als der schlechteste Eintrag. Wenn die Liste schon voll ist (n Einträge), erfolgt kein Eintrag. Ansonsten wird der neue Eintrag hinten angehängt. In beiden Fällen ist der Aufwand $O(1)$.
  - Wenn die neue Hypothese besser ist, als der schlechteste Eintrag, muß sie an der richtigen Stelle einsortiert werden. Im worst case hat man tatsächlich n Arbeitsschritte.

Speziellen Verfahren zur weiteren Geschwindigkeitssteigerung ist Abschnitt 3.8 gewidmet.

3.3.4 Ergebnisse

In der Praxis hat sich gezeigt, daß die neue Methode, wenn man die Hypothesenlistenlänge n auf 40 setzt, fast genauso schnell ist, wie die alte Grammatiklösung. Interessanterweise steigt die Zeit ungefähr proportional mit n (siehe Abbildung 3.11) und nicht etwa quadratisch, wie bei der theoretischen Betrachtung der Schleifendurchläufe in Tabelle 3.1.

Die Ergebnisse, die dieses Verfahren liefert, sind leicht schlechter als die der alten Version mit Grammatik. Das kommt durch die feste Beschränkung der Hypothesenlistenlänge, wodurch mögliche richtige Lösungen, die zu einem frühen Zeitpunkt eine sehr schlechte Bewertung haben, getilgt werden. Wenn man n entsprechend hoch setzt, kommt man bis auf wenige Prozentpunkte auf die Ergebnisse der exakten Version. Allerdings sind die Laufzeiten dann auch deutlich schlechter. Ein guter Kompromiß findet sich, wenn man n auf 40 setzt (siehe Tabelle 3.2). Man sieht dann auch in Abbildung 3.11, daß danach die Erkennungsraten nur noch langsam steigen, die Zeit allerdings nach wie vor deutlich zunimmt.

Grammatik	Hypothesen	Worte	Sätze	Zeit/Satz	Speicher
ohne	-	90.1%	56.8%	0.4 sec	3.5 MB
alt	∞	99.5%	98.4%	5.0 sec	49.0 MB
neu	100	99.4%	98.3%	12.0 sec	8.0 MB
neu	60	99.2%	98.2%	8.4 sec	6.5 MB
neu	40	99.1%	98.0%	6.3 sec	5.5 MB
neu	20	98.4%	97.1%	3.9 sec	4.5 MB
neu	10	97.6%	96.0%	2.5 sec	4.0 MB
neu	6	96.4%	94.7%	2.0 sec	4.0 MB
neu	4	94.8%	93.1%	1.5 sec	4.0 MB
neu	2	87.2%	84.8%	1.2 sec	4.0 MB

Tabelle3.2: Gegenüberstellung Erkennungsraten – Laufzeiten – Speicherplatzverbrauch bei Verwendung der kleinen Grammatik

Vorteilig zur alten Version ist ganz klar der drastisch verminderte Speicherverbrauch und somit die Möglichkeit, auch größere Grammatiken zu verwenden. Testläufe mit der Grammatik, die die gesamten Namen des Karlsruher Telefonbuchs enthält, ergaben wegen der jetzt höheren Perplexität, wie erwartet, schlechtere Ergebnisse und längere Laufzeiten (Tabelle 3.3, Abbildung 3.12, Abbildung 3.13). Beim Verwenden der großen Grammatik werden im Vergleich zur kleinen ungefähr 3 MB Speicher mehr benötigt. Dieser zusätzliche Speicher wird alleine für die Darstellung des Grammatikgraphen gebraucht.

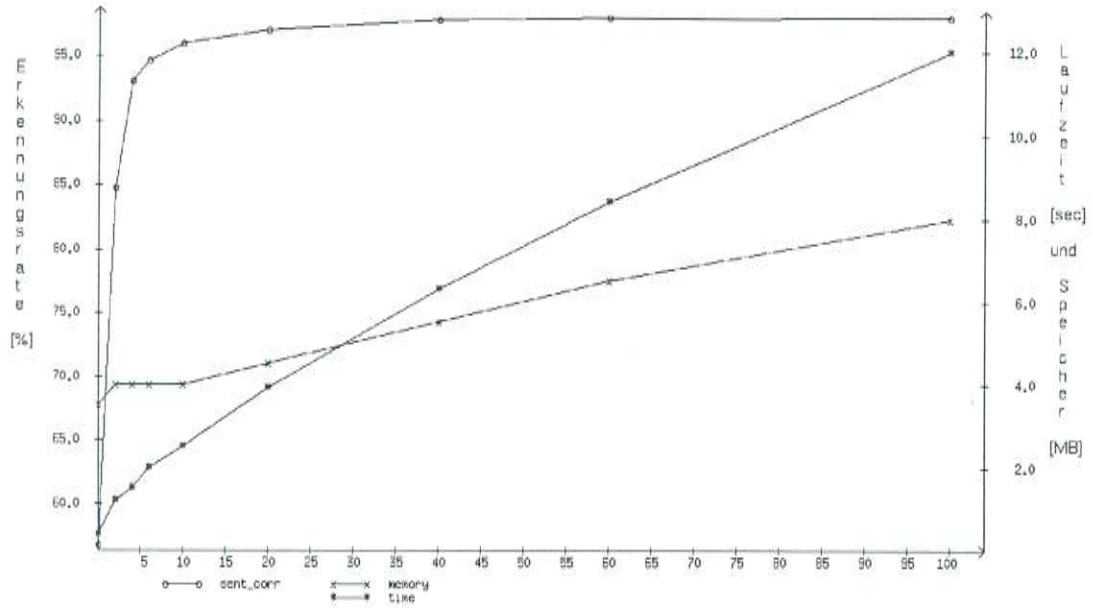

Abbildung 3.11: Graphische Gegenüberstellung von Erkennungsraten, Zeit- und Speicherplatzverbrauch, abhängig von der Hypothesenlistenlänge (Werte aus Tabelle 3.2).

Hypothesen	Worte	Sätze	Zeit/Satz
100	98.2%	92.8%	29.5 sec.
60	98.1%	92.6%	18.8 sec.
40	97.9%	92.2%	13.1 sec.
20	97.2%	91.6%	8.3 sec.
10	95.9%	90.3%	4.8 sec.
6	94.0%	87.8%	3.6 sec.
4	91.9%	85.6%	2.6 sec.
2	84.2%	76.4%	1.8 sec.

Tabelle3.3: Gegenüberstellung Erkennungsraten – Laufzeiten – Speicherplatzverbrauch bei Verwendung der großen Grammatik

3.4 Einbau von Wahrscheinlichkeiten

Die bisherigen Ergebnisse wurden mit Grammatiken erzeugt, in denen Übergänge zwischen den Zuständen entweder erlaubt oder verboten, nicht aber mit Wahrscheinlichkeiten versehen waren. Man könnte es auch so auffassen, daß nur Wahrscheinlich-

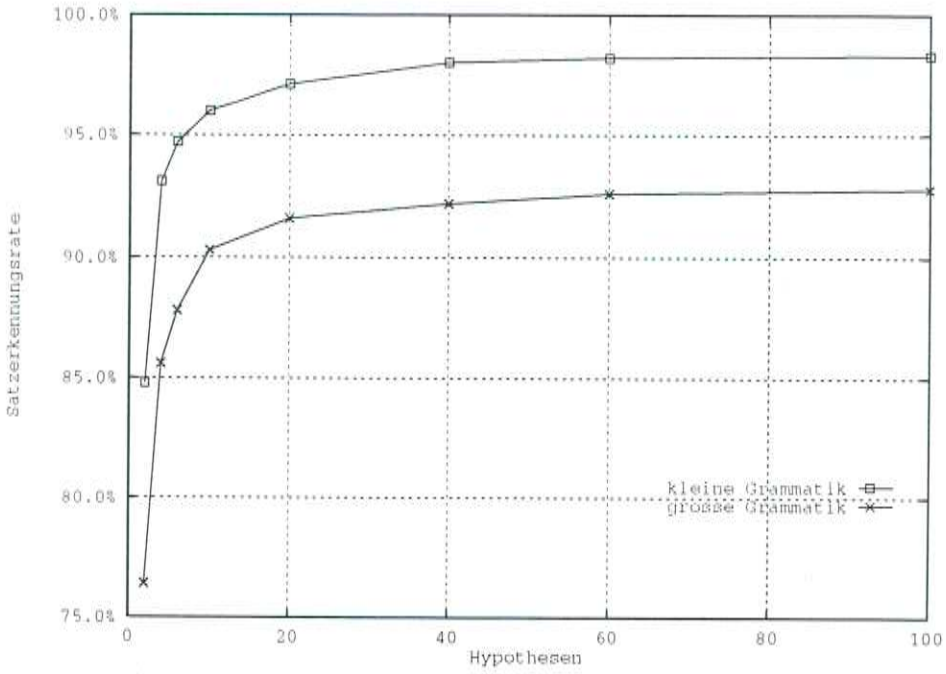

Abbildung 3.12: Gegenüberstellung der Erkennungsraten mit kleiner und großer Grammatik, in Abhängigkeit von der Anzahl der Hypothesen.

keitswerte von 0 oder 1 vorkamen. Wenn man die Übergänge nun tatsächlich mit aus einem Trainingsset ermittelten Wahrscheinlichkeiten belegt, ist eine weitere Verbesserung der Erkenungswahrscheinlichkeiten zu erwarten, da mehr Wissen einbaut wird und die Perplexität der Aufgabe sinkt. In Tabelle 3.4 sind die Testset-Perplexitäten beider Grammatiken, jeweils mit und ohne Wahrscheinlichkeiten gegenübergestellt. Sie wurden nach dem Verfahren aus Kapitel 2.3.2 berechnet.

Grammatik	ohne	klein	groß
ohne Wahrscheinlichkeiten	36.0	4.90	8.46
mit Wahrscheinlichkeiten	-	4.28	7.61

Tabelle3.4: Perplexitätsreduktion durch Verwenden von Wahrscheinlichkeiten

Um nun die Wahrscheinlichkeiten in den Erkennungsprozeß einfließen zu lassen, muß man sich eine geeignete Methode ausdenken, wie man diese mit den Bewertungen der DTW-Pfade kombiniert.

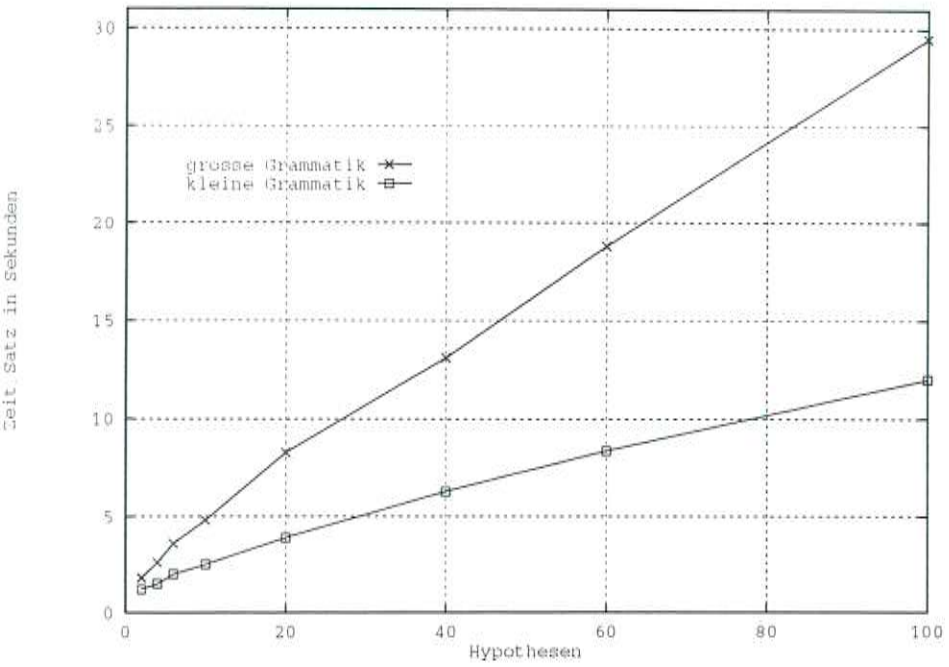

Abbildung 3.13: Gegenüberstellung der Laufzeit mit kleiner und großer Grammatik, in Abhängigkeit von der Anzahl der Hypothesen.

3.4.1 Arbeitsweise

In vielen Spracherkennungssystemen, die nach dem DTW Prinzip arbeiten, liegen in den einzelnen Zellen, über die das DTW ausgeführt wird, Wahrscheinlichkeiten, die eine Abschätzung dafür sind, wie gut das entsprechende Phonem mit der Akustik übereinstimmt. Die Pfade werden dann berechnet, indem die negativen Logarithmen der Wahrscheinlichkeiten summiert und minimiert werden. Dann kann man auch am Wortende die Negationen der logarithmierten Wahrscheinlichkeiten eines entsprechenden Wortübergangs auf die Pfadbewertung addieren. In unserem Fall werden in der DTW die Phonemhypothesenbewertungen summiert und maximiert, was eigentlich kein mathematisch korrektes Vorgehen ist. Unter der Annahme, daß es sich bei diesen Werten um logarithmierten Phonemwahrscheinlichkeiten ähnliche Werte handelt, addiere ich hierzu bei Wortübergängen die logarithmierten Wahrscheinlichkeiten aus der Grammatik. Allerdings kommt man damit noch zu keinen guten Ergebnissen, da in diesem Fall vom Pfad immer nur Werte abgezogen werden und Wortübergänge somit immer bestraft werden. Das Resultat ist, daß Wortübergänge soweit möglich vermieden werden und somit sehr viele Auslaß-Fehler entstehen. Es ist also nötig, das ganze zu normalisieren. Ziel sollte sein, ein ausgewogenes Verhältnis zwischen der Summation positiver und negativer Werten entstehen zu lassen, so daß am Ende eines Pfades die Summe aller Bewertungen, die von den Wahrscheinlichkeiten kamen, vom Betrag her nicht sehr groß ist.

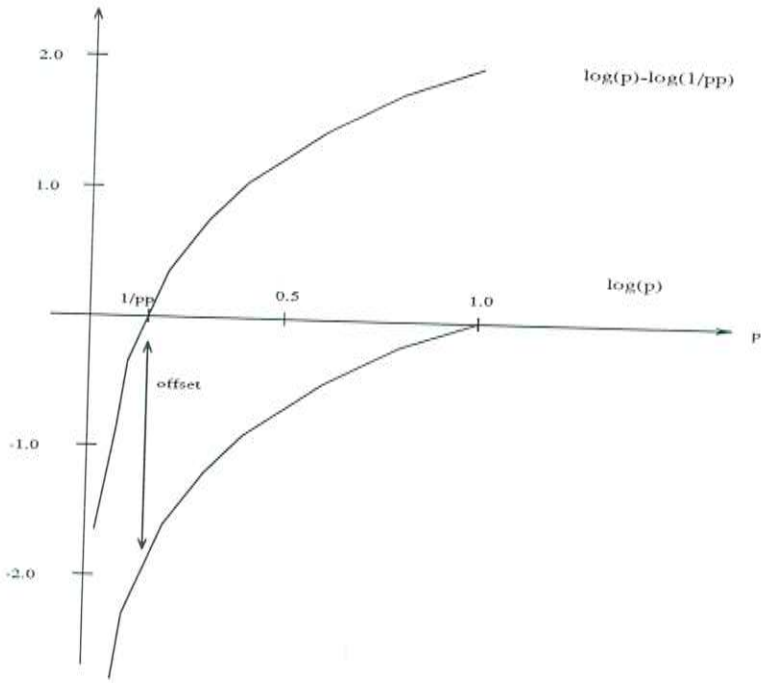

Abbildung 3.14: Zu den logarithmierten Wahrscheinlichkeiten wird $-log(1/pp)$ addiert.

Da die Perplexität pp den durchschnittlichen Verzweigungsgrad angibt, liegt die durchschnittliche Wahrscheinlichkeit für einen Übergang bei $1/pp$. Da in diesem Normalfall der Wert, der zu der Pfadbewertung addiert werden soll, bei 0 liegen soll, bietet es sich an, bei einem Wortübergang der Wahrscheinlichkeit p zu der bisherigen Pfadbewertung den Wert $log(p) - log(1/pp)$ zu addieren (Abbildung 3.14). Ein Testlauf mit der großen Grammatik ergab so eine Steigerung der Satzerkennungsrate um 0.7 Prozentpunkte. Üblicherweise werden die Ergebnisse der obigen Formel vor der Summation noch mit einem Faktor gewichtet, um den Einfluß, den die Grammatik auf den Erkennungsprozeß hat, steuern zu können. So kommt man für den allgemeinen Fall bei Wortübergängen auf die Formel

$$pfad := pfad + factor \cdot (log(p) + offset) \tag{3.1}$$

3.4.2 Ergebnisse

Ergebnisse für die große Grammatik mit Wahrscheinlichkeiten finden sich in Tabelle 3.5. Bei einer Perplexität von 7.6 für diesen Test ergibt sich für den Offset ein Wert von ungefähr 2.0. In der Tabelle wurden für diesen Offset verschiedene Gewichtungsfaktoren getestet, und man stellt fest, daß bei einer zu hohen Gewichtung der Wahrscheinlichkeiten die Ergebnisse wieder schlechter werden. Die besten Ergebnisse ließen sich mit Faktoren im Bereich zwischen 0.3 und 0.7 erzielen. Die Satzerkennungsrate konnte dann

um einen Prozentpunkt gegenüber der Erkennung ohne Wahrscheinlichkeiten gesteigert werden.

Der Einbau der Berücksichtigung von Wahrscheinlichkeiten ändert am Laufzeitverhalten des Programms nichts wesentliches. In einem Vorverarbeitungsschritt werden alle Wahrscheinlichkeiten mit der oben beschriebenen Formel in Strafen umgewandelt, die dann während der Erkennung nur noch auf die Pfadbewertung addiert werden müssen. Die Zeit, die zur Erkennung eines Satzes benötigt wird steigt im Schnitt um 0.5 Sekunden.

factor	Worte	Sätze
0.0	97.9%	92.2%
0.1	98.0%	92.6%
0.2	97.9%	93.1%
0.3	98.1%	93.2%
0.4	98.2%	93.2%
0.5	98.1%	93.2%
0.6	98.1%	93.2%
0.7	98.0%	93.2%
0.8	98.0%	93.1%
0.9	97.9%	93.1%
1.0	97.8%	92.9%

Tabelle3.5: Erkennungsraten mit Wahrscheinlichkeiten. Der Offset liegt jeweils bei $-log(1/pp) \approx 2.0$, die Hypothesenlistenlänge wurde auf 40 gesetzt.

Ein Test mit verschiedenen anderen Offsets (Tabelle 3.6) hat gezeigt, daß der theoretisch ermittelte Wert fast das Optimum darstellt. Mit einem etwas kleineren Offset ließ sich allerdings das Ergebnis nochmals um 0.1 Prozentpunkte (Satzerkennungsrate) steigern.

factor	offset	Worte	Sätze
0	-	97.9%	92.2%
1.0	0.5	98.0%	93.1%
1.0	1.0	98.1%	93.3%
1.0	1.8	97.9%	93.1%
1.0	2.0	97.8%	92.9%

Tabelle3.6: Test mit verschiedenen Offsets und gleichen Faktoren; 40 Hypothesen.

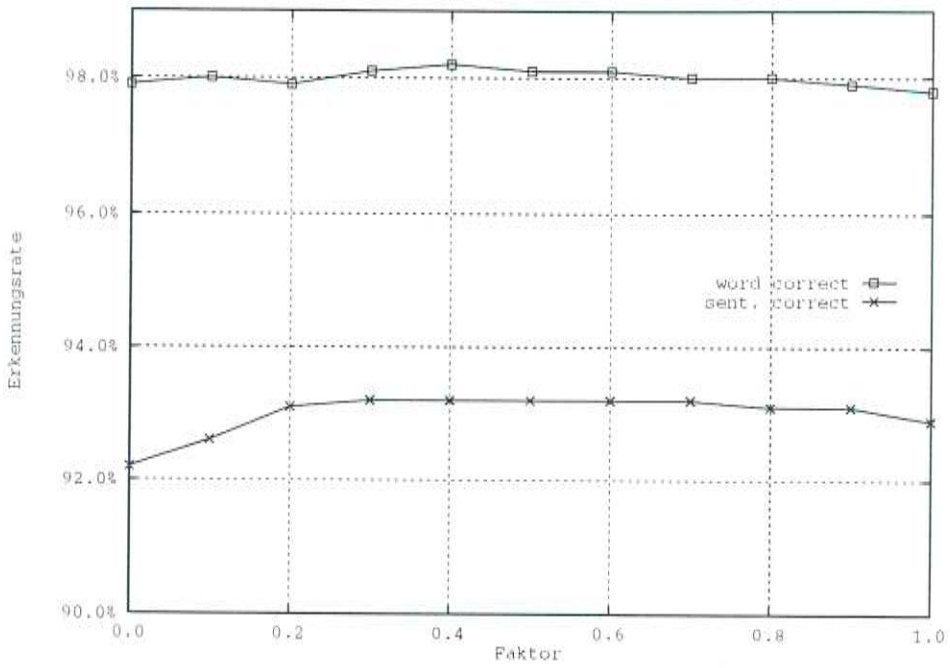

Abbildung 3.15: Satz- und Worterkennungsrate mit Wahrscheinlichkeiten in Abhängigkeit des Gewichtungsfaktors.

3.5 Einbau von Uni- und Bigrammen

Die bisherige Grammatikimplementierung basiert darauf, daß die Übergangswahrscheinlichkeiten für jeden Buchstaben von allen vorherigen Buchstaben abhängig sind. In vielen Spracherkennungssystemen werden dagegen die in Kapitel 2.3 beschriebenen Uni-, Bi- oder Trigramme benutzt, die Abhängigkeiten nur von einer fest vorgegebenen Zahl von Vorgängern zulassen. Da bei uns so viele Vorgänger wie vorhanden berücksichtigt werden, könnte man auch von ∞-grammen oder einer Grammatik mit vollem Linkskontext sprechen.

Um eine Vergleichsmöglichkeit zu haben, wie gut unsere ∞-gramme im Vergleich zu den klassischen Uni-, Bi- und Trigrammen abschneiden, entstand der Wunsch, in das System allgemeine n-gramme einzubauen. Solche n-gramme sind normalerweise als Tabellen abgelegt, aus denen man die Wahrscheinlichkeiten der Übergänge ablesen kann (Tabelle 3.7).

Die naheliegende Idee ist, das Programm an sich nicht zu modifizieren und stattdessen als Grammatik einen Graphen zu benutzen, der das jeweilige n-gram simuliert. Für Uni- und Bigramme sind die Lösung relativ einfach und in Abbildung 3.16 und 3.17 dargestellt. Bei Trigrammen wird der Graph etwas unhandlich, und das Programm, welches einen Trigram-Graphen erzeugt, konnte aus Zeitgründen noch nicht implementiert werden.

	a	b	c	d	e	...
$	0.09	0.04	0.02	0.13	0.11	
a	0.02	0.15	0.09	0.12	0.02	
b	0.21	0.01	0.02	0.09		
c	0.14	0.04	0.00		$p(e\|c)$	
d	0.11	0.03		$p(d\|d)$	$p(e\|d)$	
...						

Tabelle3.7: Bigramm-Wahrscheinlichkeiten als Tabelle ($ bezeichnet den Satzanfang).

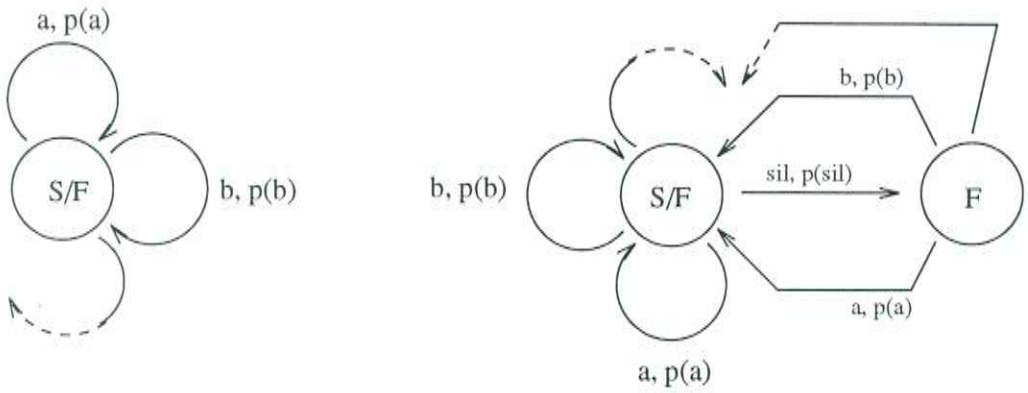

Abbildung 3.16: Das Unigram als Graph. Links ohne und rechts mit *silence*

3.5.1 Ergebnisse

Die Perplexitäten der n-gramme sind in Tabelle 3.8 gegenübergestellt. Sie wurden mit dem in Abschnitt 2.3.2 beschriebenen Verfahren berechnet. Man erwartet, daß sich die Perplexitätsunterschiede der verschiedenen n-gramme in den Erkennungsraten niederschlagen.

n	Perplexität
0	36.0
1	22.3
2	13.3
∞	7.6

Tabelle3.8: Perplexitäten von n-grammen

Entgegen der Intuition konnte durch das Verwenden von Unigrammen keine merkliche Steigerung der Erkennungsraten gegenüber der Version ohne Grammatik erzielt werden (Tabelle 3.9). Bei Bigrammen konnten Satzerkennungsraten bis 62.1% erzielt

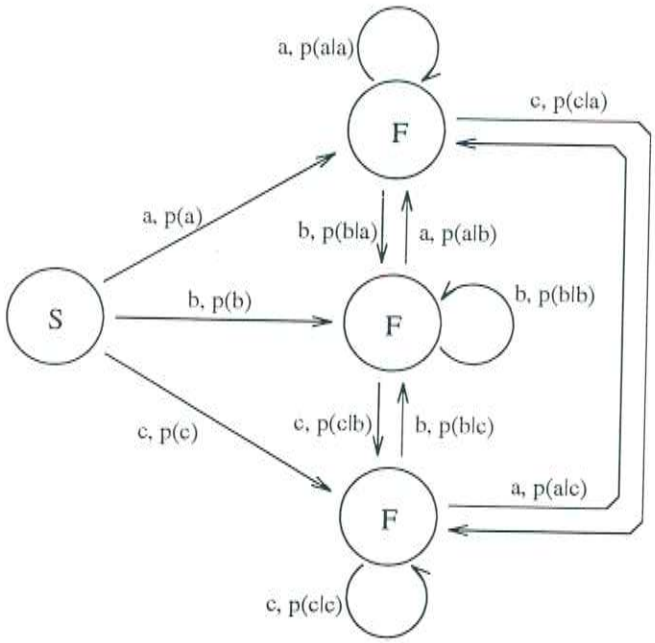

Abbildung 3.17: Darstellung eines Bigrams für 3 Buchstaben a, b und c als Graph

werden, was schon eine deutliche Verbesserung gegenüber den Unigrammen ist (Tabelle 3.10). Allerdings waren die Laufzeiten extrem hoch (23 Sekunden), was von einer hohen Zahl von Grammatik-Übergängen ($36^2 \approx 1300$) und der hohen Perplexität kommt.

factor	Worte	Sätze
0.0	90.1%	56.8%
0.1	90.1%	56.8%
0.2	90.1%	56.9%
0.3	90.0%	56.6%
0.4	90.0%	56.2%
0.5	90.0%	55.5%
0.6	89.8%	55.1%
0.7	89.6%	54.7%
0.8	89.5%	54.4%
0.9	89.4%	54.1%
1.0	89.3%	54.0%

Tabelle3.9: Erkennungsraten von Unigrammen mit verschiedenen Faktoren. Der Offset liegt jeweils bei $-log(1/22.3) \approx 3.1$, die Hypothesenlistenlänge wurde auf 40 gesetzt.

factor	Worte	Sätze
0.0	90.2%	57.1%
0.1	90.5%	58.1%
0.4	91.2%	59.9%
0.7	91.8%	61.2%
0.8	91.9%	61.8%
0.9	92.0%	62.1%
1.0	92.0%	61.9%
1.1	91.9%	61.6%
1.2	91.8%	61.2%

Tabelle3.10: Erkennungsraten von Bigrammen mit verschiedenen Faktoren. Der Offset liegt jeweils bei $-log(1/13.3) \approx 2.6$, die Hypothesenlistenlänge wurde auf 40 gesetzt.

grammatik	Worte	Sätze	Zeit/Satz
ohne	90.1%	56.8%	0.4 sec
Unigram	90.1%	56.9%	1.1 sec
Bigram	92.0%	62.1%	23.1 sec
∞-gram	98.2%	93.2%	13.1 sec

Tabelle3.11: Erkennungsraten mit verschiedenen n-grammen. Die Ergebnisse der Uni- und Bigramme wurden mit dem alten Grammatikmodus erzeugt.

3.6 Einbau einer n-best Suche

Dadurch daß in dem neuen Grammatikmodus mit Hypothesenlisten gearbeitet wird, kommt man auf die Vermutung, daß es möglich sein müßte, ohne große Änderungen im Programm eine n-best Suche zu implementieren. Die Idee solcher Ansätze ist, den Erkenner in einem ersten Durchlauf möglichst schnell möglichst viele Hypothesen generieren zu lassen und sich in einem zweiten Schritt daraus diejenige auszuwählen, die gewissen Kriterien entspricht. In unserem Fall bedeutet das, zuerst ohne Grammatik zu arbeiten und dabei aufgrund eines gegenüber der neuen Grammatiklösung leicht modifizierten Verfahrens Hypothesen generieren zu lassen, um anschließend daraus die beste zu wählen, die nach der Grammatik erlaubt ist.

Es existieren zahlreiche Veröffentlichungen mit Vorschlägen zu n-best Algorithmen. Ich habe mich entschieden, das von Richard Schwartz und Steve Austin in [SA91] vorgeschlagene Word-Dependent n-best Verfahren zu implementieren, da dieses erstens gute Erkennungsraten verspricht und sich zweitens ohne große Änderungen in das jetzige System einbauen läßt.

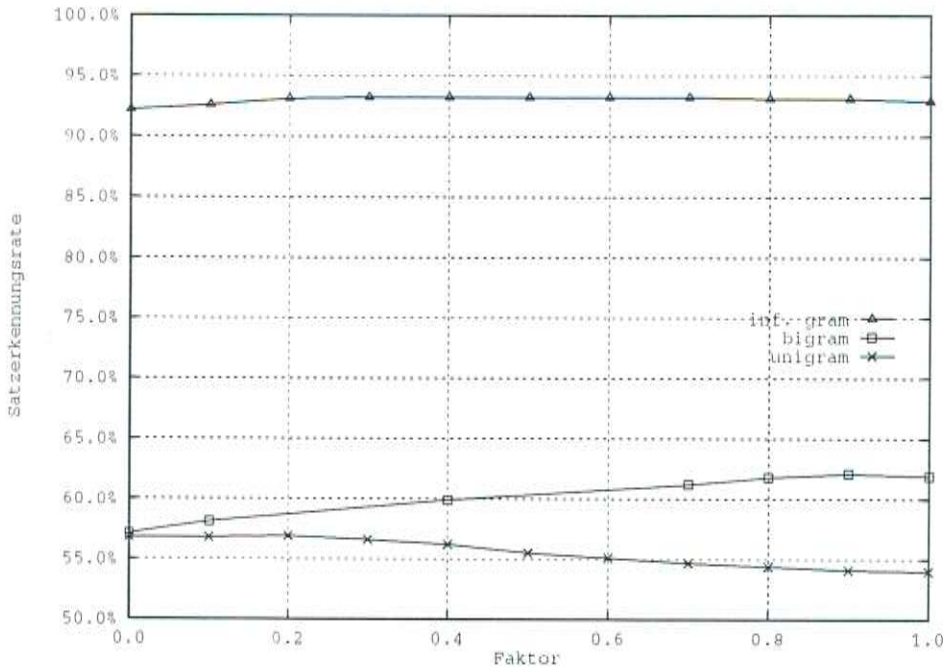

Abbildung 3.18: Satzerkennungsrate verschiedener n-Gramme in Abhängigkeit der Gewichtungsfaktoren.

3.6.1 Arbeitsweise

Schwartz und Austin gehen von einer normalen DTW ohne Grammatik aus, in der bei jedem Wortübergang nicht nur der beste Pfad gemerkt wird, sondern die Bewertungen aller besten Vorgänger (das wären bei uns bis zu 36). Zum Weiterrechnen in der DTW wird allerdings nur der beste Pfad genommen. Vergleichbares passiert in dem von mir implementierten neuen Grammatikmodus. Ich simuliere das Berechnen der DTW ohne Grammatik mit mehreren Hypothesen pro Zeitpunkt durch das Laufenlassen des Algorithmus für den neuen Grammatikmodus, wobei ich keine Grammatik benutze, sondern zu jedem Zeitpunkt für jeden Pfad erlaube, in jedem beliebigen Wortmodell weiterzulaufen.

Nachdem dieser Vorwärtsdurchgang abgeschlossen ist, beginnt die n-best Suche. Dabei wird zuerst der beste Pfad und anschließend solange rekursiv der nächstbeste gesucht, bis einer gefunden wird, der in der Grammatik liegt. Dieses Finden des $n+1$-besten Pfades aus dem n-besten passiert, indem für jeden Wortübergang zum Zeitpunkt t eine Alternative gebildet wird, und zwar indem aus der Hypothesenliste von t der nächstschlechtere Eintrag genommen wird. Für die Worte von t bis zum Satzende wird die alte Sequenz beibehalten und ab t bis zum Anfang wird wird die Sequenz genommen, die sich durch Verfolgen der Backpointer bei dem alternativen Eintrag ergeben. Die so gewonnenen neuen Pfade werden auf einen Stack gelegt und der beste Pfad, der auf dem Stack liegt, wird jetzt zur n-besten Lösung.

Pseudocode für n-best Suche

```
...
Hypothesengenerierung mit Grammatik, die alle Nachfolger erlaubt
...
lösche n-best-stack
finde beste Lösung und lege sie auf den n-best-stack
solange beste Lösung aus n-best-stack nicht in Grammatik (höchstens n mal)
   aktuelle_Lösung ist beste Lösung im stack
   bilde Alternativen bei jedem Wortübergang der aktuellen_Lösung
   bewerte die alternativen Pfade mit der Bewertung der
            aktuellen_Lösung - Betrag, den die Alternative schlechter ist
   vervollständige die alternativen Pfade und lege sie auf den n-best-stack
   lösche die aktuelle_Lösung vom n-best-stack
end
gebe die beste Lösung des n-best-stacks aus
```

Ein Vorteil dieses Ansatzes ist, daß die Lösungen Schritt für Schritt gefunden werden und man nicht am Anfang n fest vorgeben muß. Man kann aber ein maximales n angeben, um eine Beschränkung der Laufzeit zu erreichen. Der Rechenaufwand, um eine weitere Lösung zu erhalten, ist nicht sehr hoch. Das Problem hierbei ist allerdings, daß viele Lösungen unterschlagen werden, da im Vorwärtsdurchgang schon viele Pfade verlorengehen. Man betrachte nochmals Abbildung 3.10 von Seite 34. Da die beiden Pfade verschmolzen werden, geht mitunter die zweitbeste Lösung verloren. Eine Abhilfe dagegen verschaffe ich mir dadurch, daß Pfade, die zusammenlaufen, noch ein Wort lang parallel gehalten werden. Dadurch geht kein Vorgänger verloren. Schwartz und Austin nennen dieses Verfahren '*Word-Dependent N-best*'.

3.6.2 Ergebnisse

Die Ergebnisse des implementierten Verfahrens waren sehr enttäuschend (Tabelle 3.12). Der Grund liegt in der Verwendung einer nicht speziell an das Problem angepaßten Lösungsstrategie. So geht eigentlich die Grundidee eines N-best Verfahrens verloren, nämlich daß der Vorwärtsdurchgang sehr schnell geht. In meiner Implementierung braucht er durch die jetzt viel höhere Perplexität (36 statt 4.9 oder 8.5, vgl. Tabelle 3.4 auf Seite 39) deutlich mehr Zeit im Vorwärtsdurchgang. Dazu kommt dann noch die Zeit, um die eigentliche Lösung zu finden (obwohl das dann nicht mehr viel ausmacht). Zudem sind die Erkennungsraten spürbar schlechter als mit der normalen Grammatiksuche.

Dies könnte daher kommen, daß in meiner Implementierung im Gegensatz zu der in [SA91] vorgeschlagenen Strategie, nicht garantiert ist, daß pro Wortmodell die k besten Pfade gemerkt werden, die zu jedem Zeitpunkt in dieses hineinführen. Durch die Organisation der Hypothesenlisten werden stattdessen pro Zeitpunkt die besten Pfade

über *alle* Wortmodelle gesammelt. So kann es passieren, daß, wenn ein Buchstabe undeutlich ausgesprochen und somit schlecht erkannt wird, für ihn überhaupt keine Hypothese generiert wird, selbst wenn man sehr lange Hypothesenlisten wählt. Folglich kann dann auch nie die richtige Lösung gefunden werden. Schwartz und Austin schlagen vor, pro Zeitpunkt 3 bis 6 Hypothesen pro Wortmodell zu merken. Umgerechnet auf 36 Wortmodelle wären das in unserem Fall $3 * 36$ bis $6 * 36$, d. h. in etwa 100 bis 200. Ein Vergleichstest mit 100 Hypothesen ergab jedoch keine Steigerung der Erkennungsleistung gegenüber der Version mit 20 Hypothesen.

max n	Hypothesen	Worte	Sätze	Zeit/Satz
500	100	94.9%	82.8%	148.0 sec
500	40	94.9%	82.8%	41.8 sec
500	20	94.9%	82.8%	16.5 sec
500	10	94.6%	81.4%	9.9 sec

Tabelle3.12: Schlechte Ergebnisse mit der *N*-best Suche

3.7 Eine auf Stringabständen basierende Suche

Eine sich von den bisherigen Suchmethoden deutlich unterscheidende, wieder auf dem Modus ohne Grammatik basierende Vorgehensweise, wurde aus Vergleichsgründen implementiert. Da die Ergebnisse, die der Erkenner ohne Grammatik liefert, meist nur in ein oder zwei Stellen fehlerhaft sind, kommt man auf die Idee, die Ergebnisse in dieser Form zu benutzen und dann aus einer Liste erlaubter Nachnamen (oder im Allgemeinen: Sätze) denjenigen herauszusuchen, der der erkannten Buchstabensequenz am ähnlichsten ist.

3.7.1 Arbeitsweise

Das Hauptproblem bei dieser Methode ist, eine Funktion zu definieren, die für zwei Buchstabensequenzen ein geeignetes Abstandsmaß liefert. Hat man diese Funktion, muß man nur aus einer Liste erlaubter Sequenzen diejenige mit minimalen Abstand heraussuchen. Ein gängiges Abstandsmaß für zwei Buchstabensequenzen ist die Zahl der benötigten Ersetzungs-, Einfüge und Löschschritte, um von der einen Sequenz auf die andere zu kommen. Es handelt sich hierbei um einen relativ gängigen Algorithmus, der mit Methoden des dynamischen Programmierens angegangen wird. Die von mir erstellte Implementierung läßt es auch zu, daß für alle Buchstabenpaare ein Abstand angegeben werden kann. Dieser besagt, wie schwer ein Ersetzungsfehler für das entsprechende Paar zu bewerten ist, so daß man z. B. einen Ersetzungsfehler von 'b' und 'd' weniger schwer bewerten kann, als eine Verwechslung von 'b' mit 'x'.

Pseudocode zur Suche mit Stringabständen.

```
Berechne Lösung L ohne Grammatik.
if L in Grammatik
  L ausgeben
else
  for all erlaubte Nachnamen
      suche minimalen Abstand zu L
  gib den Nachnamen mit minimalem Abstand aus
end
```

3.7.2 Ergebnisse

Es ist klar, daß nun die Laufzeit stark variiert, je nachdem, ob die Suche ohne Grammatik eine Lösung anbietet, die in der Grammatik liegt, oder nicht. Im ersten Fall wird die Lösung nach ungefähr 0.4 Sekunden ausgegeben, im zweiten kommt noch die Zeit dazu, die für die Stringvergleiche benötigt wird, welche von der Länge der gefundenen Lösung (in Worten bzw. Buchstaben) und der Zahl der Vergleichsmuster abhängt. Im Schnitt sind es etwas mehr als zwei Sekunden die hierfür zusätzlich benötigt werden. Ein Problem dieses Ansatzes ist, daß er nicht gut parallelisierbar ist, da die Lösung erst komplett vorliegen muß, damit sie vernünftig mit der Liste der erlaubten Lösungen verglichen werden kann (eine Möglichkeit wäre, jeden einzelnen Vergleich parallel laufen zu lassen, das wären in unserem Fall 32000 parallele Prozesse).

Mit dieser Methode konnten die Erkennungsraten im Vergleich zur Version ohne Grammatik (90.1% Worte, 56.8% Sätze) deutlich gesteigert werden, reichen aber lange nicht an die Ergebnisse der Versionen mit Grammatik heran. Tabelle 3.13 enthält in der ersten Zeile die Ergebnisse, wenn für die Stringvergleiche nur die diskreten Werte null und eins für Fehler zugelassen werden und die zweite Zeile die Ergebnisse, wenn Wortverwechslungsfehler kontinuierlich bewertet werden. Diese Fehlerbewertungen wurden von mir zwar nur geschätzt – trotzdem ließen sich die Erkennungsraten verbessern. Mit aus tatsächlich begangenen Ersetzungsfehlern des Erkenners ermittelten Werten lassen sich die Erkennungsraten vermutlich noch weiter steigern.

Wortabstände	Worte	Sätze	Zeit/Satz
0 oder 1	92.5%	76.8%	1.6 sec
kontinuierlich	94.4%	82.8%	1.6 sec

Tabelle3.13: Tests ohne Grammatik mit anschließendem Suchen des Nachnamens mit minimalem Abstand zur erkannten Buchstabensequenz. Abstandssuche mit DTW, einmal mit diskreten, einmal mit (geschätzten) kontinuierlichen Wortpaarabständen.

3.8 Geschwindigkeitssteigerung

Neben guten Erkennungsergebnissen ist die Minimierung der Zeit, die das System zum Erkennen eines Satzes braucht, immer ein zweites, leider oft mit dem ersten konkurrierendes Ziel. Die 13 Sekunden, die das Programm mit der großen Grammatik und bei einer Hypothesenlistenlänge von 40 benötigt (siehe Tabelle 3.3 auf Seite 38) klingen zwar wenig, aber wenn man sich vorstellt, daß man mit einem Telefonvermittlungssystem spricht, welches nach dem Buchstabieren eines Namens durch den Benutzer 13 Sekunden braucht, bis eine Reaktion erfolgen kann, ist das zu viel.

3.8.1 Beam-search

Ein Verfahren, welches in fast jedem Spracherkennungssystem verwendet wird, um auf einfache Weise die Geschwindigkeit zu erhöhen, ist das sogenannte Beam-search, oder Strahlensuchverfahren. Man setzt dabei einen meist empirisch aus Testläufen ermittelten Wert, den sogenannten *beam*, der angibt, um wieviel ein Pfad maximal schlechter sein darf als der momentan beste, damit er noch weiterverfolgt wird (Abbildung 3.19). Dieses Verfahren wird in dem DTW-Lauf benutzt. In dem Modus ohne Grammatik werden die momentan berechneten Pfade zur Zeit t mit dem Maximum des Zeitpunktes $t-1$ verglichen. Pfade, die schlechter als $maximum - beam$ sind, werden mit $-\infty$ bewertet und fallen somit bei der weiteren Berechnung weg. In dem alten Grammatikmodus wird der *beam* noch benötigt, um die Deaktivierung von Wortmodellen zu steuern. Falls alle Pfade in einem Wortmodell zu einem Zeitpunkt (es gibt pro Wortmodell pro Zeitpunkt genausoviel Pfade wie Zustände existieren) schlechter sind als $maximum - beam$, wird das Wortmodell komplett deaktiviert und erst wieder aktiviert, falls wieder ein Pfad von einem aktiven Wortmodell in jenes führt. Im neuen Grammatikmodus wird der *beam* noch benutzt, um das Vorwärtspropagieren von Hypothesen einzuschränken: Eine Hypothese wird nur dann in eine Liste einsortiert, falls sie nicht um *beam* schlechter ist, als der beste Eintrag dieser Liste.

Es liegt auf der Hand, daß die Größe, die man für den *beam* wählt, sehr von der Implementierung, insbesondere von der Art der Eingabedaten in die DTW abhängt. Bei unserem System haben die Phonemhypothesen Bewertungen zwischen 0 und 1. Der *beam* war in den bisherigen Tests immer 40. Weitere Tests haben gezeigt, daß man den *beam* noch deutlich enger, bis auf 20 setzen kann, ohne die Erkennungsraten zu verschlechtern (Tabelle 3.14). Die Rechenzeit nimmt dabei etwa um 1/4 ab. Wenn man den *beam* weiter auf 10 zurücknimmt, sinkt die Satzerkennungsrate zwar um 1.2 Prozentpunkte, aber gegenüber der Version mit $beam = 40$ reduziert sich die Rechenzeit auf etwa 40%. Man sieht hier wieder deutlich das Dilemma in dem man steckt, wenn es darum geht, die Parameter, mit denen das Programm läuft, zu wählen.

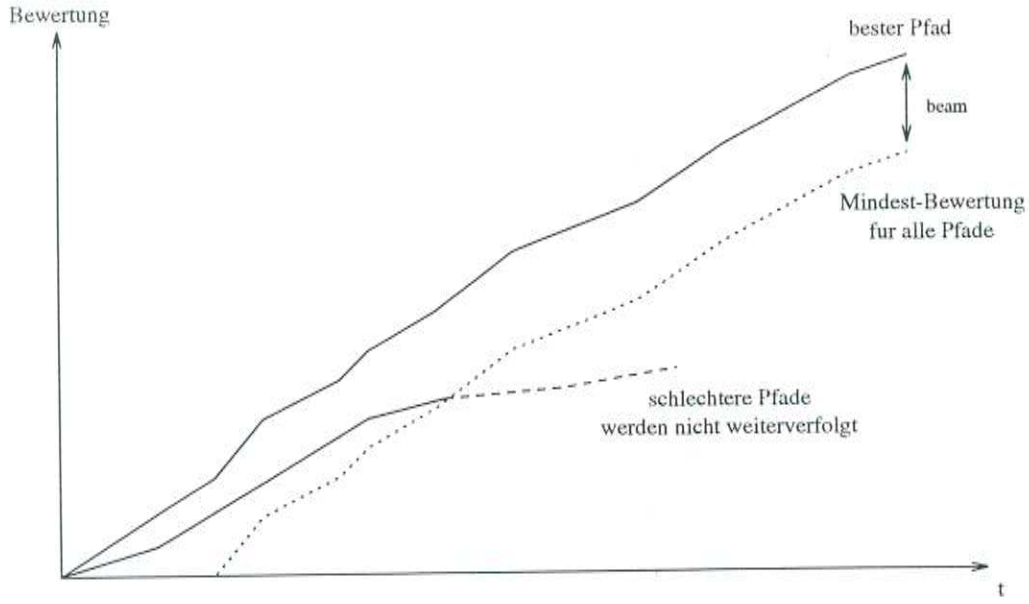

Abbildung 3.19: Alle Pfade, die um *beam* schlechter als der lokal beste sind, werden nicht mehr weiterverfolgt.

beam	Worte	Sätze	Zeit/Satz
40	97.9%	92.2%	13.1 sec
30	97.9%	92.2%	12.2 sec
20	97.8%	92.2%	9.6 sec
10	96.4%	91.0%	5.1 sec

Tabelle3.14: Ergebnisse mit verschiedenen *beams* mit der großen Grammatik ohne Wahrscheinlichkeiten, 40 Hypothesen

3.8.2 Überspringen von Frames

Bei Analysen der Hypothesenlisten stellte sich heraus, daß in zeitlich aufeinanderfolgenden Listen relativ ähnliche Einträge zu finden waren. Dies legte die Idee nahe, beim Vorwärtspropagieren der Hypothesen etwas ungenauer vorzugehen und die Hypothesen nicht mehr in alle Listen weiterzureichen, sondern nur in jede zweite oder dritte. Der Code ändert sich dabei nur minimal (umrahmter Teil):

Pseudocode für neuen Grammatikmodus **mit Überspringen von Frames**

schreibe $START$ in $Hypothesenliste[0]$
for $t = 0$ to end
  for all $h \in Hypothesenliste[t]$
    for all $Nachfolger$ von h, die Grammatik erlaubt
      for $tt = t$ to $t + max\_dauer(Nachfolger)$
        berechne $DTW(Nachfolger, t, tt)$
      end
      for $tt = t$ to $t + max\_dauer(Nachfolger)$ **step k**
        $neuer\_score = alter\_score(h) + DTW(Nachfolger, t, tt)$
        $neue\_hypothese = h$ mit $Nachfolger$
        sortiere $(neue\_hypothese, neuer\_score)$ in $Hypothesenliste[tt]$ ein
      end
    end
  end
end

Testläufe haben sehr gute Ergebnisse gezeigt (Tabelle 3.15). Wenn nur in jeden k-ten Frame eine Vorwärtspropagierung stattfindet, ergibt sich mit den Bezeichnungen aus Tabelle 3.1 von Seite 36 eine theoretische Zahl der Schleifendurchläufe von $n * max\_dur * (w * J + pp * \frac{n}{k})$. Zeitmessungen jedoch haben ergeben, daß es sich um eine zu k fast lineare Zeitreduktion handelt. Dabei nehmen die Erkennungsraten nicht sehr schnell ab – erstaunlicherweise nimmt die Satzerkennungsrate sogar noch um 0.3 Prozentpunkte zu, wenn k auf 2 gesetzt wird. Dies ist aber wohl eher Zufall als eine generell anzuratende Vorgehensweise, um die Erkennungsleistung zu erhöhen.

3.8.3 Der Forward-Backward-Algorithmus

Steve Austin, Richard Schwartz und Paul Placeway haben in [ASP91] einen allgemeinen Algorithmus vorgestellt, mit dem der Suchvorgang in Spracherkennungssystemen beschleunigt werden kann. Die Grundidee ist, in einem schnellen ersten Lauf ein grobes Ergebnis zu erzielen und dann in einem zweiten, detailierten Lauf in entgegengesetzter

k	Worte	Sätze	Zeit/Satz
1	97.9%	92.2%	13.1 sec
2	97.8%	92.5%	6.9 sec
3	97.6%	91.8%	5.2 sec
4	97.4%	90.7%	4.1 sec
5	96.7%	89.1%	3.5 sec
6	96.2%	87.6%	3.2 sec
8	94.0%	82.4%	2.6 sec

Tabelle3.15: Hypothesen werden nur in jedem $k$ten Frame eingetragen. Testläufe mit der großen Grammatik ohne Wahrscheinlichkeiten, *beam* = 20, Hypothesenlistenlänge = 40.

Richtung dieses Ergebnis zu benutzen, um die Suche zu beschleunigen. Aufgrund dieser zweistufigen Arbeitsweise mit entgegengesetzten Suchrichtungen wurde der Algorithmus *Forward-Backward Suche* genannt.

Ich benutze den Algorithmus, indem ich im ersten Lauf eine leicht modifizierte, aber sonst wie in Abschnitt 3.1 erklärte normale DTW ohne Grammatik laufen lasse. Die dabei gesammelten Ergebnisse benutze ich im zweiten Schritt, um den neuen Grammatikalgorithmus zu beschleunigen. Ich lasse den ersten Schritt rückwärts laufen, weil eine Umstellung des Grammatikmodus auf eine Abarbeitung in entgegengesetzter Richtung mehr Implementierungsaufwand bedeutet hätte. Prinzipiell könnte man die Reihenfolge auch umkehren. Das hätte zum Vorteil, daß man auf einer parallelen Maschine den Vorwärtsschritt parallel zur Spracheingabe durchführen könnte und am Ende des gesprochenen Satzes das Ergebnis des DTW-Laufs ohne weitere Zeitverzögerung schon vorliegen hätte. Da allerdings die DTW-Berechnung ohne Grammatik sehr schnell ist (etwa eine halbe Sekunde im Gegensatz zu 13 Sekunden für den Lauf mit der großen Grammatik), bedeutet dies keinen großen Zeitverlust und da das ganze System momentan auch nur auf unseren nichtparallelen Maschinen läuft, macht es sowieso keinen Unterschied.

Im ersten Durchlauf, einem normalen DTW-Lauf ohne Grammatik in umgekehrter Richtung, wird an den jeweiligen Wortanfängen (eigentlich Wortenden, wenn man es von Hinten betrachtet), an denen ein Pfad zu einem Zeitpunkt t ein Wortmodell w verlassen könnte, die aktuelle akkumulierte Pfadbewertung in $\alpha(w,t)$ gespeichert (siehe Abbildung 3.20). Zusätzlich wird für jeden Zeitpunkt t gemerkt, welche Wortmodelle jeweils aktiv sind, d. h. in welchen Wortmodellen zum Zeitpunkt t ein Pfad existiert, der nicht um *beam* schlechter ist, als der momentan beste Pfad. Diese Wortmodelle werden in Ω^t gehalten. Zusätzlich wird noch die Bewertung des besten Pfades am Ende der gesamten Suche (also bei $t = 0$) in α^0 abgelegt. Diese zum normalen DTW zusätzlichen Arbeiten brauchen quasi keine Rechenzeit, nur etwas Speicher, der sich allerdings im 100KB-Bereich bewegt und somit keine Probleme bereiten dürfte.

Am Ende des ersten Durchlaufs steht also in α^0 die Bewertung des besten Pfades

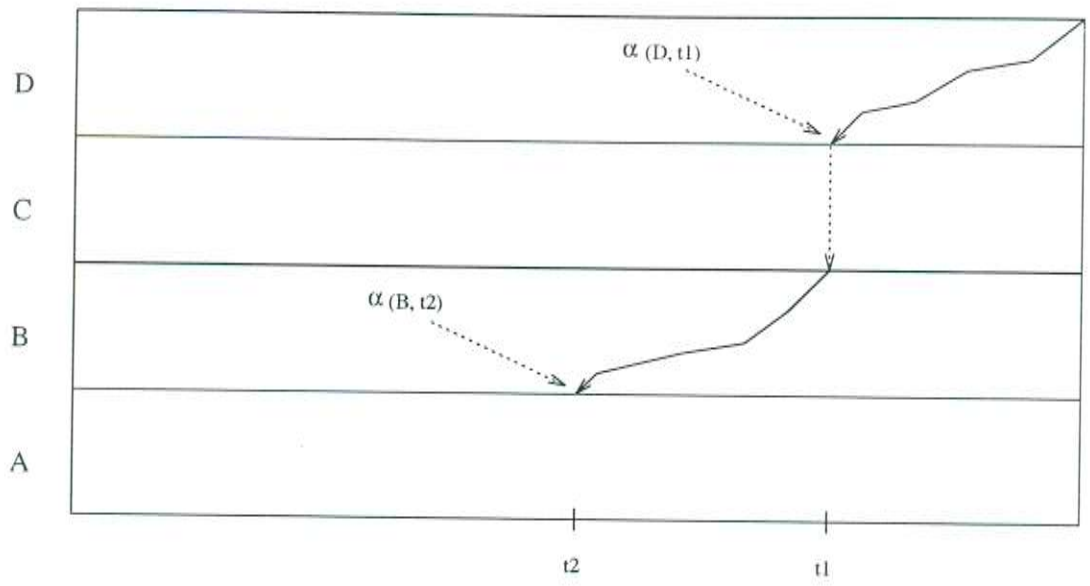

Abbildung 3.20: Im ersten Durchlauf werden die Bewertungen an den Wortübergängen gemerkt

und über die Rückwärtszeiger des Wortmodells w mit der besten Bewertung $\alpha(w, 0)$ kann der Pfad ermittelt werden, der ohne Benutzen einer Grammatik gefunden worden wäre. Dies ist jedoch nicht Ziel des Verfahrens.

Den zweiten Durchlauf stellt bis auf einige zusätzliche Abfragen, um schlechte Pfade frühzeitig zu erkennen, der neue Grammatikmodus dar. Es werden keine zusätzlichen Datenstrukturen benötigt. Wenn während der Abarbeitung des neuen Grammatikmodus der Pfad auf ein Wortende stößt, und das ist der Fall, wenn die Hypothesen weiterpropagiert werden, wird anhand der Bewertung des aktuellen Pfades (der propagiert werden soll), eine Schätzung abgegeben, wie groß die Chancen sind, daß dieser Pfad im weiteren Verlauf noch so gut wird, daß er am Ende eine Bewertung, die ähnlich gut wie α^0 ist, erhält. Nennen wir $\beta(w', t)$ die Bewertung des Pfades, der gerade propagert werden soll und an dessen Ende sich Wort w' befindet. Wenn die Grammatik erlaubt, daß man für den aktuellen Pfad von Wort w' nach w übergehen darf, dann wird man dies nur zulassen, wenn $w \in \Omega^t$, d. h. wenn w im ersten Lauf zum Zeitpunk t aktiv war, da ansonsten die Bewertungen des Pfades mit hoher Wahrscheinlichkeit sofort sehr schlecht würden. Ist $w \in \Omega^t$, so ist $\beta(w', t) + \alpha(w, t)$ eine gute Schätzung für die Bewertung, die der aktuelle Pfad am Ende erhalten wird, wenn er an dieser Stelle in Wort w fortgesetzt wird (Abbildung 3.21 versucht, diese Idee zu verdeutlichen). Man könnte nun diese Schätzung benutzen, um nur Pfade weiter zu berücksichtigen, bei denen dieser Wert höchstens um einen festen Betrag kleiner als α^0 ist:

$$\beta(w', t) + \alpha(w, t) > \alpha^0 - fb\_beam \tag{3.2}$$

Tests mit dieser Bedingung haben keine sehr guten Ergebnisse erzielt: Wenn fb_beam zu groß gewählt wurde, wurde keine Geschwindigkeitssteigerung erzielt und beim En-

germachen des beams wurden ab einem bestimmten Punkt die Erkennungsraten sofort sehr schlecht. In [ASP91] wird stattdessen das Kriterium

$$\frac{\beta(w',t)+\alpha(w,t)}{\alpha^0} > fb\_beam \tag{3.3}$$

vorgeschlagen, das Pfade erfüllen müssen, um weiterverfolgt zu werden. *fb_beam* stellt somit einen Mindestprozentsatz der besten Bewertung α^0 dar, den eine Pfadschätzung immer erreichen muß, um weiter berücksichtigt zu werden. *fb_beam* ist natürlich vollkommen unabhängig von dem *beam* des Strahlensuchverfahrens.

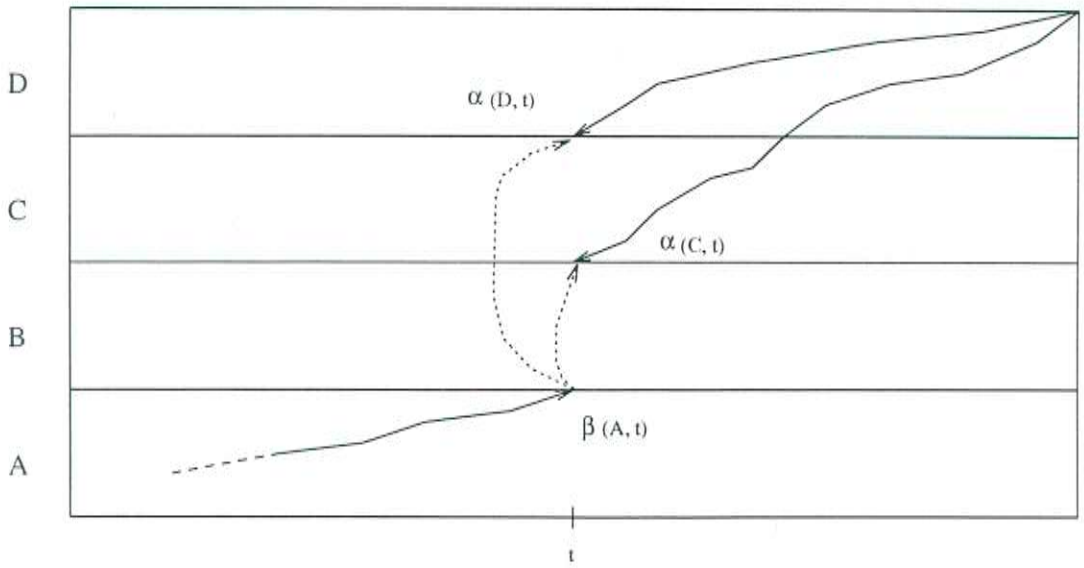

Abbildung 3.21: Im zweiten Durchlauf werden die gemerkten Bewertungen benutzt, um die weiteren Chancen eines Pfades abzuschätzen

In Tabelle 3.16 sind Testergebnisse für verschiedene solcher Prozentzahlen zusammengestellt. Man sieht, daß wenn man verlangt, daß Pfade immer mindestens 90% der besten Pfadbewertung erreichen müssen, die Satzerkennungsrate nicht abnimmt, aber die Erkennungszeit auf weniger als die Hälfte zurückgeht. Setzt man die Bedingung noch schärfer werden dann sehr schnell die Erkennungsraten niedriger, aber es sind dabei auch deutliche Geschwindigkeitssteigerungen festzustellen.

3.8.4 Kombination von Beschleunigungsverfahren

Es liegt natürlich auf der Hand, die verschiedenen vorgestellten Beschleunigungsverfahren zu kombinieren um noch kürzere Laufzeiten zu bekommen. Als sehr effektiv hat sich das gleichzeitige Benutzen des Forward-Backward Verfahrens mit dem Überspringen jedes zweiten Frames im Vorwärtslauf (also dem zweiten Lauf innerhalb der Forward-Backward Suche) erwiesen. Mit beiden Verfahren getrennt konnte die Laufzeit jeweils halbiert werden (90%-beam beim Forward-Backward Algorithmus). Eine

fb_beam	Worte	Sätze	Zeit/Satz
–	97.9%	92.2%	13.1 sec
0.90	97.7%	92.2%	6.1 sec
0.91	97.5%	91.9%	5.2 sec
0.92	97.2%	91.4%	4.7 sec
0.93	96.5%	90.6%	4.0 sec
0.94	94.9%	88.9%	3.3 sec
0.95	92.2%	86.7%	2.7 sec

Tabelle3.16: Geschwindigkeitssteigerung mit der Forward-Backward Suche. Testläufe mit 40 Hypothesen und $beam = 40$.

Kombination ergab tatsächlich eine Viertelung der Laufzeit, ohne daß die Satzerkennungsrate abnahm (Tabelle 3.17). Die hier erreichten 3.5 Sekunden für die Erkennung von im Schnitt 3 Sekunden langen Sätzen auf einem nichtparallelen Rechner bei der Verwendung einer Grammatik mit 140 000 Übergängen sind, so denke ich, eine vertretbare Zeit. Wie schon beim separaten Benutzen des Überspringens von Frames beobachtet, steigt die Erkennungsrate, verglichen mit dem normalen Verfahren ohne jegliche Beschleuniger, sogar noch etwas an, wenn man den *beam* für den Forward-Backward Algorithmus auf 0.85 setzt.

fb_beam	Worte	Sätze	Zeit/Satz
0.85	98.0%	92.6%	5.2 sec
0.90	97.8%	92.2%	3.5 sec
0.91	97.5%	91.9%	3.1 sec
0.92	96.7%	90.7%	2.6 sec
0.93	95.6%	89.5%	2.3 sec
0.94	93.9%	87.6%	1.9 sec
0.95	91.0%	84.0%	1.5 sec

Tabelle3.17: Kombination der Forward-Backward Suche mit Überspringen jedes 2ten Frames im Vorwärtsdurchgang. Testläufe wieder mit 40 Hypothesenn und $beam = 40$.

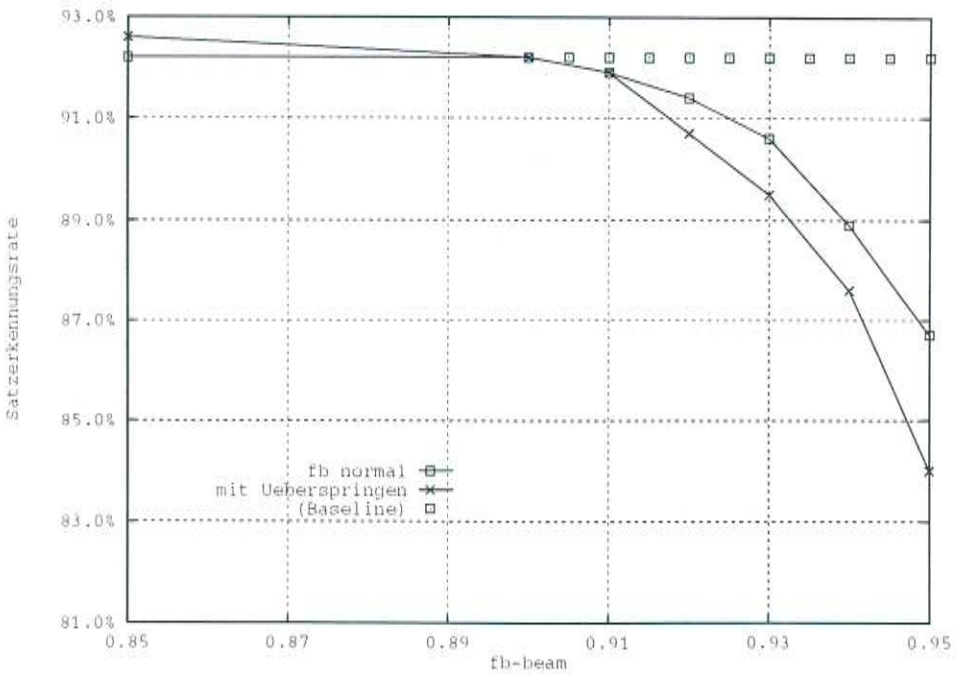

Abbildung 3.22: Gegenüberstellung der Satzerkennungsraten des normalen Forward-Backward Algorithmus mit dessen Kombination mit dem Überspringen jedes zweiten Frames im Vorwärtsdurchgang. Zur Orientierung stellt die gepunktete Linie die Erkennungsrate ohne Forward-Backward Suche dar.

Kapitel 4

Zusammenfassung

Es konnte anhand konkreter Implementierungen und Tests gezeigt werden, daß klassische Uni- und Bigamm Sprachmodelle, die sich in Aufgabengebieten, in denen große Vokabulare vorkommen als günstig erwiesen haben, beim Erkennen buchstabierter Namen nicht wesentlich besser als das Arbeiten ohne jegliche Grammatik abschneiden (Tabelle 4.1). Auch das Verfahren mit Stringabständen und das nach Austin und Schwartz implementierte n-best Verfahren lieferten keine optimalen Ergebnisse.

Modus	Worte	Sätze	Zeit/Satz
ohne Grammatik	90.1%	56.8%	0.4 sec
Unigramme	90.1%	56.8%	1.1 sec
Bigramme	92.0%	62.1%	23.1 sec
Stringabstände diskret	92.5%	76.8%	1.6 sec
Stringabstände kontinuierlich	94.4%	82.8%	1.6 sec
n-best	94.9%	82.8%	16.5 sec
Neuer Grammatikmodus	97.7%	92.7%	3.2 sec

Tabelle4.1: Zusammenfassung der Ergebnisse – Teil 1

Eine neue Methode, das Verwenden einer Grammatik mit vollem Linkskontext, die man auch als ∞-gram bezeichnen könnte, brachte sehr gute Erkennungsraten. In einer ersten Implementierung ergaben sich bei großen Grammatiken Schwierigkeiten mit dem Speicherplatz. Durch einen Algorithmus, der mit einer Hypothesenbewertung arbeitet, konnte der benötigte Speicherplatz soweit reduziert werden, daß jetzt fast beliebig große Grammatiken verwendbar sind.

Ausgehend von der Version ohne Grammatik, die nur 56% der Nachnamen korrekt erkannte (erste Zeile der Tabelle 4.2) und der ersten Version mit Grammatik, die zwar sehr gute Erkennungsraten lieferte, dafür aber viel Speicher benötigte und keine Wortlisten, die mehr als etwa 1000 Namen enthalten, zuließ (zweite Zeile der Tabelle), konnte durch eine neue Suchtechnik der Speicherplatzbedarf bei einer nur geringen

Verschlechterung der Erkennungsleistung drastisch reduziert werden (dritte Zeile der Tabelle).

Die neue Arbeitsweise läßt zu, beliebig lange Wortlisten zu benutzen. Ein Test mit einer 32000 Namen umfassenden Liste konnte nun durchgeführt werden. Durch die Erweiterung des Wortschatzes sinkt die Satzerkennungsrate von 98% auf 92% und durch den erhöhten Rechenaufwand steigt die Erkennungszeit ungefähr auf das doppelte an (vierte Zeile der Tabelle).

Durch den Einbau spezieller Verfahren zur Geschwindigkeitssteigerung konnte die Laufzeit soweit reduziert werden, daß das Erkennen mit Benutzen der 32000-Namen-Liste schneller geht als bei der alten Implementierung mit 1000 Namen (Zeile 5 der Tabelle).

Eine Steigerung der Erkennungsrate um einen Prozentpunkt konnte durch das Benutzen von Wahrscheinlichkeiten erreicht werden, ohne dabei Geschwindigkeitseinbußen hinnehmen zu müssen (Zeilen 6 und 7 der Tabelle).

Modus	Worte	Sätze	Zeit/Satz	Speicher
ohne Grammatik	90.1%	56.8%	0.4 sec	3.5 MB
alte Grammatikimplementierung mit kleiner Grammatik	99.5%	98.4%	5.0 sec	49.0 MB
neue Grammatikimplementierung mit kleiner Grammatik	99.1%	98.0%	6.3 sec	5.5 MB
neue Grammatikimplementierung mit großer Grammatik	97.9%	92.2%	13.1 sec	8.5 MB
geschwindigkeitsoptimiert	97.8%	92.2%	3.5 sec	8.5 MB
mit Wahrscheinlichkeiten	98.2%	93.2%	13.6 sec	8.5 MB
mit Wahrscheinlichkeiten, geschwindigkeitsoptimiert	97.7%	92.7%	3.2 sec	8.5 MB

Tabelle4.2: Zusammenfassung der Ergebnisse – Teil 2

Kapitel 5

Ausblick

5.1 Verbesserungen am jetzigen Programm

Ein Projekt wie dieses Buchstabiererkennungssystem ist natürlich nie *fertig*. Es wird immer Stellen geben, an denen man Details verbessern kann oder es entsteht der Wunsch, am ganzen Konzept etwas zu ändern. Im Verlauf meiner Arbeit sind mir einige Punkte aufgefallen, an denen weiterführende Arbeiten ansetzen können. Ich will sie hier wiedergeben:

- Mathematisch korrektes Vorgehen beim Berechnen der DTW (kein Addieren von Wahrscheinlichkeiten!).
- Einbau von Heuristiken, um die Güte einer gefundenen Lösung zu bewerten.
- Erkennen von Versprechern ("*... äh - nein, ich meine ...*").
- Erkennen von Hilfestellungen ("*H wie Haus, A wie Auto, ...*").
- Implementierung auf einem Parallelrechner.

Anhang A

Die Testumgebung

Zum Testen stand ein Set buchstabierter Nachnamen zur Verfügung, die aus Sprachspenden von Studenten der Universität Karlsruhe stammen. Die von den verschiedenen Sprechern jeweils zu buchstabierenden Namen wurden vorher per Zufall aus dem Karlsruher Telefonbuch ausgewählt. Um Trainingsdaten für seltene Phoneme und Buchstaben (z. B. Q, X, Y) zu gewinnen, wurden einige Sprecher noch gebeten, Zufallssequenzen zu sprechen.

Das gesamte Testset wurde in 2 disjunkte Teile aufgeteilt. Mit dem einen Teil, der 8133 Sätze inklusive der Zufallssequenzen enthielt, wurde das TDNN zur Phonemerkennung trainiert. Den anderen Teil, der noch 1316 legale Nachnamen umfaßt, benutzte ich für meine Tests. Die Trainingsdaten stammen von 70 und die Testdaten von 23 verschiedenen Sprechern.

Die Testläufe, deren Ergebnisse in dieser Arbeit zusammengefaßt sind, beinhalten immer einen Lauf über alle 1316 Sätze. Die durchschnittliche Länge der Sätze beträgt 3 Sekunden. Die in manchen Tabellen angegebenen Laufzeiten des Programms wurden mit dem Programm *time* auf einer HP735 berechnet. *time* summiert nur die tatsächlich vom Programm verbrauchte CPU-Zeit auf. Zeit, die für das Ein- und Auslagern von Programmteilen auf Platte (swapping) verbraucht wird, ist darin nicht enthalten. Das bedeutet, daß die Zeit, die man tatsächlich auf die Ausgaben des Programms warten muß u. U. deutlich höher sind, besonders wenn auf der Maschine auch andere Prozesse laufen oder wenn man den alten Grammatikmodus mit großen Grammatiken benutzt, da dann viel Speicher benötigt wird, und somit oft ge*swappt* werden muß.

Die verschiedenen Parameter (z. B. die Offsets und Faktoren für die Grammatik oder die verschiedenen Beams) sind immer aufgrund der gleichen Daten optimiert und getestet worden. Infolge dessen sind die daraus gewonnen Größen evtl. nicht optimal für neue Sprachdaten. Aus diesem Grund bietet es sich an, die Parameter, die für Geschwindigkeitssteigerungen verantwortlich sind, nicht zu scharf zu setzen und für den Offset beim Lauf mit Grammatik den von der entsprechenden Perplexität abhängigen Wert, wie in Abschnitt 3.4.1 beschrieben, zu benutzen.

Anhang B

Die Grammatiken

Für die Testläufe wurden zwei verschiedene Grammatiken benutzt. Der Grund liegt darin, daß in dem alten Grammatikmodus die Grammatik, die das komplette Karlsruher Telefonbuch enthält, nicht benutzt werden kann, da sie zu groß ist. Um Vergleichstests durchführen zu können, wurde noch eine Grammatik erzeugt, die nur genau die 1316 Namen enthält, die in dem Testset vorkamen. Diese Grammatik wurde als *kleine Grammatik* bezeichnet, die andere dementsprechend als *große Grammatik*. Alle Testergebnisse, ausgenommen derer aus Tabelle 3.2 von Seite 37, wurden mit der großen Grammatik erzeugt.

Bei beiden Grammatiken sind als minimale Automaten dargestellt. Es wurden die in Abschnitt 2.3.1 dargelegten Mechanismen zur Berücksichtigung der verschiedenen Aussprachevarianten für ß und den Bindestrich eingebaut. Nach jedem Zustand darf optional ein silence folgen. Soweit nicht anders angegeben, wurden die Tests ohne Verwenden von Wahrscheinlichkeiten durchgeführt.

Aus Vergleichsgründen wurde auch noch jeweils für die kleine und die große Grammatik eine Version als Baum (sonst: minimaler Graph) erzeugt. Tests mit den Baum-Versionen ergaben allerdings keine Unterschiede, so daß die speicherplatzschonendere Alternative gewählt wurde.

Grammatik	Sätze	verschiedene S.	Buchstaben	Zustände	Übergänge
klein	1316	1113	8905	3662	7994
als Baum				12166	18248
groß	111882	32267	274974	54970	142912
als Baum				278630	421448

TabelleB.1: Gegenüberstellungen der verschiedenen Grammatiken.

Anhang C

Technischer Anhang

Es folgt eine Beschreibung der Parameter, die das Verwenden der Sprachmodelle steuern. Sie können über ein `.par`-file oder interaktiv gesetzt werden.

`DP_grammar` (Typ INT):
Setzen des Grammatikmodus. Erlaubte Werte: 0, 1, 2, 3, -3, 4.

- 0 – Keine Grammatik
- 1 – Mit Stringabständen
- 2 – Alter Grammatikmodus
- 3 – Neuer Grammatikmodus
- -3 – Neuer Grammatikmodus mit Forward-Backward Suche
- 4 – N-best Suche

`grammar_dfafile` (Typ STRING):
Gibt den Filenamen des DFA-Files an, in dem die Grammatik steht. Darf nicht gesetzt werden, wenn n-gramme benutzt werden sollen.

`DP_beam` (Typ FLOAT):
Der *beam* für die Berechnung der DTW in allen Grammatikmodi. Steuert zusätzlich die Aktivierung/Deaktivierung in Grammatikmodus 2 und das Einsortieren in die Hypothesenlisten in Grammatikmodus 3. Sinnvolle Werte liegen zwischen 10 und 40.

`DP_fb_beam` (Typ FLOAT):
Der *beam* für den Forward-Backward Modus. Sinnvolle Werte liegen zwischen 0.85 und 0.95.

`DP_prop_pruning` (Typ INT):
Steuert das Überspringen von Frames. Es wird im Vorwärtslauf bei Grammatikmodus 3, -3 oder 4 nur in jeden k-ten Frame propagiert. Sinnvolle Werte sind 0 (kein Überspringen) und 2 bis 10.

`histo_size` (Typ INT):
Gibt an, wieviele Hypothesen in Grammatikmodus 3, -3 und 4 pro Zeiteinheit gemerkt werden sollen. Sinnvolle Werte liegen zwischen 10 und 100.

`histo_hash_size` (Typ INT):
Gibt die Größe der Hashtables an, über die auf die Hypothesenlisten zugegriffen wird. Sollte mindestens so groß sein wie `histo_size`.

`vb_dpmatch2` (Typ INT):
Steuert die Bildschirmausgaben während der Erkennung. `vb_dpmatch2` ist ein Bit-Vektor mit folgender Belegung:

Bit	Wert	Beschreibung
0	1	Zeige am Ende der DTW die gefundenen Wortgrenzen an.
1	2	Zeige nach dem DTW-Lauf die Abweichung des Ergebnisses vom besten Pfad an 10 Stützstellen an (um `DP_beam` einzustellen) und zusätzlich, wenn im Grammatikmodus 3 die Position des Ergebnispfades innerhalb der Hypothesenlisten (um `histo_size` einzustellen) an.
2	4	Zeige die Aktivierung der Wortmodelle am Anfang, am Höhepunkt und am Ende an, falls man sich in Grammatikmodus 2 befindet.
3	8	Wenn im Grammatikmodus 3: Zeige während des DTW-Laufes einen Count-Down der noch zu bearbeitenden Frames an. Sinnvoll evtl. bei sehr großen Hypothesenlisten um zu sehen, ob das Programm noch läuft.
4	16	Zeige den absoluten Score des gefundenen Pfades und den relativen Score (= Score geteilt durch Zahl der Frames)
5	32	Ähnlich wie Bit 2. Zeige die Aktivierung zu jedem Frame. Lange Ausgabe!
6	64	Zeig die n-best Liste an, wenn im n-best Modus.
7	128	Berechne die Perplexität des Testsets. Ausgabe direkt vor dem Erkennen des ersten Satzes.
8	256	Zeigt im Forward-Backward-Mode an, wieviele % der Hypothesen durch die FB-Regeln weggeworfen wurden.
9	512	Zeigt für den Grammatik-Modus 1 an, was die Lösung ohne Grammatik war.

`magic_grammar_factor` und `magic_grammar_offset` (Typ FLOAT):
Steuern die Gewichtung der Wahrscheinlichkeiten. Die Penalties Berechnen sich nach $pen = factor * (log(p) + offset)$. Sinnvolle Werte sind 0.0 für den factor, wenn ohne Wahrscheinlichkeiten gerechnet werden soll, ansonsten 0.1 bis 1.0 für den factor und $log(1/pp)$ für den offset.

`DP_n_best_n` (Typ INT):

Das n für den n-best Algorithmus, z. B. 500.

`ngramfile` (Typ STRING):
Das File mit Sätzen, aus denen n-gramme berechnet werden sollen. Wenn n-gramme benutzt werden sollen, darf `grammar_dfafile` nicht gesetzt sein.

`ngram_mode` (Typ STRING):
Gibt an, welche n-gramme benutzt werden sollen. Erlaubt ist 'uni' und 'bi'.

`ngram_silmode` (Typ BOOL):
Gibt an, ob bei n-grammen optionale silences eingebaut werden sollen.

`sentfile` (Typ STRING):
Gibt das sentence-file an und wird im Grammatikmodus 1 (mit Stringabständen) benötigt. Beinhaltet erlaubte Nachnamen.

`mindfa_variant_mode` (Typ INT):
Wird im Grammatik-Modus 1 benötigt und gibt an, wie die Aussprachevarianten für die aus `sentfile` eingelesenen Sätze behandelt werden sollen:

Bit	Wert	Beschreibung
0	1	Einsetzen von 'doppel'-Alternativen
1	2	Aussprachevarianten für Spezialworte

`word_distances_file` (Typ STRING):
Beinhaltet Filename aus dem die Wortpaarabstände für Grammatikmodus 1 initialisiert werden sollen.

`pp_test_set_sentences` (Typ STRING):
Wenn über `vb_dpmatch2` die Perplexitätsberechnung eingeschaltet wurde, werden aus diesem File die Testsätze gelesen.

Literaturverzeichnis

[WHH+89] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, K. Lang. Phoneme Recognition using Time-Delay Neural Networks. In: *IEEE Transactions on Acoustics, Speech and Signal Processing* Vol. 37, No. 3 March 1989.

[HW92] Hermann Hild, Alex Waibel. Multi-Speaker/Speaker-Independent Architectures for the Multi-State Time Delay Neural Network. In: *IEEE Proceedings of the International Conference on Acoustics, Speech and Signal Processing*. Minneapolis, Minnesota, USA, April 1993.

[HW93] Hermann Hild, Alex Waibel. Speaker Independent Connected Letter Recognition with a Multi-State Time Delay Neural Network. In: *3rd European Conference on Speech Communication and Technology*. Berlin, Germany, Sept. 1993.

[HW93.2] Hermann Hild, Alex Waibel. Connected Letter Recognition with a Multi-State Time Delay Neural Network. In: *Advances in Neural Information Processiong Systems 5 (NIPS*5)*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[Ney90] Herman Ney. The Use of a One-Stage Dynamic Programming Algorithm for Connected Word Recognition. In: *Readings in Speech Recognition*. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[Jel] F. Jelinek. Self-Organized Language Modelling for Speech Recognition. In: *Readings in Speech Recognition*. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[JMR] F. Jelinek, R. L. Mercher, S. Roukos. Principals of Lexical Language Modelling for Speech Recognition. In: *Advances in Speech Signal Processing*. Marcel Dekker, Inc.

[Rab] L. R. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. In: *Readings in Speech Recognition*. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[SA91] Richard Schwartz, Steve Austin. A Comparision of Several Approximate Algorithms For Finding Multiple (N-BEST) Sentence Hypotheses. *IEEE Proceedings of the International Conference on Accoustics, Speech and Signal Processing*. 1991.

[ASP91] Steve Austin, Richard Schwartz, Paul Placeway. The Forward-Backward Search Algorithm. *IEEE Proceedings of the International Conference on Accoustics, Speech and Signal Processing*. 1991.

[Fer] M. Ferreti, G. Maltese, S. Scarci. Language Model and Accoustic Model Information in Probabilistic Speech Recognition. In: *IEEE Proceedings of the International Conference on Accoustics, Speech and Signal Processing*. Glasgow, 1989.

[SCF+93] P. Schmid, R. Cole, M. Fanty, H. Bourland, M. Haessen. Real-Time, Neural Network-Based, French Alphabet Recognition with Telephone Speech. In: *3rd European Conference on Speech Communication and Technology*. Berlin, Germany, Sept. 1993.

[MSYM93] Yasuhiro Minami, Kiyohiro Shikano, Tomokazu Yamada, Tatsumo Matsuoka. Very-Large-Vocabulary Continuous Speech Recognition Algorithm for Telephone Directory Assistance. In: *3rd European Conference on Speech Communication and technology*. Berlin, Germany, Sept. 1993.

[BW88] H. Bourland, C. J. Wellekens. Links between Markov Models and Multilayer Perceptrons. In: *Advances in Neural Information Processiong Systems 1 (NIPS*1)*. Morgan Kaufmann Publishers, San Mateo, CA, 1988.

[BCN+93] Eric B. Buhrke, Regis Cardin, Yves Normandin, Mazin Rahim, Jay Wilpon. Application of Vector Quantized Hidden Markov Models to the Recognition of Connected Digit Strings in the Telephone Network. In: *Proceedings of the 1993 IEEE Workshop on Automatic Speech Recognition*. Snowbird, Utah, Dec. 1993.

[Wil] Jay G. Wilpon. Application of Speech Recognition Technology in Telecommunications. In: *Proceedings of the 1993 IEEE Workshop on Automatic Speech Recognition*. Snowbird, Utah, Dec. 1993.

[Lee93] Kai-Fu Lee. Speech Recognition for Personal Computing. In: *Proceedings of the 1993 IEEE Workshop on Automatic Speech Recognition*. Snowbird, Utah, Dec. 1993.

[Knuth73] Donald E. Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching. Addison Wesley, 1973.