Diploma thesis (Diplomarbeit)

N-Best Hidden Markov Model Supertagging for Typing with Ambiguous Keyboards

Saša Hasan

July 2, 2003

Universität Koblenz-Landau
Campus Koblenz
Fachbereich Informatik
Universitätsstraße 1
56070 Koblenz, Germany

Supervisors (Betreuer):
Prof. Dr. Karin Harbusch
Dipl. Inform. Michael Kühn

Acknowledgements

"It seems very pretty, but it's rather hard to understand! Somehow it seems to fill my head with ideas — only I don't exactly know what they are!" These are the comments of Alice on the poem *Jabberwocky* in the book *Through the Looking Glass* written by Lewis Carroll. The quotation reflects in a way my state of mind (not only) at the beginning of working on this thesis project.

As time progressed, things fortunately became much clearer. First of all, I would like to thank my supervisors, Prof. Karin Harbusch and Michael Kühn, for creating a congenial atmosphere and pushing me back into the right direction when things got a bit out of hand. Without their valuable comments, this thesis would simply not be this thesis.

I am grateful to Jens Bäcker for providing his supertagger and other scripts that made the uncountable hours in front of the computer feel shorter than they actually were.

I am also indebted to Kerstin Wiebe and Sebastian Bochra for pointing out that there actually *is* life beside the worktable. I thank them for many pleasant discussions that helped me to keep a level head.

Thanks also go to Daniel Fastner, Christian Kölle and Jan Spiekermann who were willing to glance at early drafts of this work.

I acknowledge the generosity of Alexander Fuchs who, in times of low printing quotas, helped me with printouts which accounted for finding the one or another small error that is, funnily enough, invisible on monitors during early morning hours.

Special thanks go to Hajo Hoffmann for many insightful discussions about life, the universe and everything.

Finally, I am greatly indebted to my parents and my brother for the persistent encouragement and support and patience and love. Thank you.

Declaration (Erklärung)

I hereby declare that this diploma thesis is entirely written by myself. All sources of information are listed in the bibliography.

Hiermit versichere ich, dass ich diese Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Saša Hasan
Koblenz, July 2, 2003

Abstract

Individuals with severe speech impairment are dependent on communication aids that allow them to communicate with other people. Communication is an important factor against isolation of the disabled and is necessary in everyday life for exchanging information, making requests or expressing thoughts. So far, several predictive text entry devices have been developed that reduce the number of keystrokes needed for entering a word by proposing possible candidates to the current input prefix. By selecting one of the available completions, the number of keypresses decreases but the overall time to enter the word is not necessarily reduced because of the cognitive load that emerges while scanning the candidate lists for the target word. Another restriction arises for motor impaired users that cannot operate a common keyboard and have to rely on reduced keyboards instead. These keyboards map several letters to a single physical key which results in an ambiguous coding of the words. Thus, a code entered with this kind of keyboard usually yields a list of matching words. The primary goal is to order these candidates in a way that the most appropriate words are placed at the top for minimal selection costs. The easiest way to achieve this is to sort the list according to word frequencies obtained from large corpora.

In this study, the attention is focused towards a sentence-wise text entry system with a highly reduced ambiguous keyboard containing only three letter keys and one command key. The main idea is to utilize the syntactic dependencies that exist between the words of a sentence and use this information to adjust the candidate lists such that more likely words appear at the top. Instead of being distracted by a list of proposals after every keypress as it is the case with a predictive word-wise approach, the user can concentrate on what she wants to express in the first phase, i.e. the ambiguous typing of the words, and disambiguate the target words from the candidate lists in a second phase. In an idealistic scenario, the system would find the correct sentence hypothesis, resulting in minimal selection costs for the user. The technique that is chosen in this work to achieve this goal is *supertagging*, a procedure that associates so-called supertags, i.e. elementary trees that code local dependencies, with the corresponding words of the sentence. The core of the system is an n-best supertagger that is based on a trigram Hidden Markov Model and is able to find the n best sentence hypotheses for a sequence of coded words. An additional *lightweight dependency analysis* results in a "shallow parse" and retains only the most promising hypotheses for adjusting the final candidate lists.

The results obtained from a very small evaluation corpus (250 sentences) show that the approach yields better rankings. The percentage of target words that appear at the first position in the suggestion list could be increased from 50 to 62%, whereas on average, the words appear at the second position with the n-best approach. 95.8% are placed within the top 5 ranks. It is assumed that with a larger training corpus and thus better language model, the rankings could even be improved. Further evaluations have to be carried out in the future to prove this claim.

Contents

List of Figures

List of Tables

1 Introduction

Communication is self-evident. It is a basic necessity for everyday interaction with other people and enables individuals to convey thoughts, ideas, feelings or whatever needs to be expressed in some way. In its broadest sense, *communication* can be situated in the area of *cognitive science* which is "the study of human intelligence in all its forms from perception and action to language and reasoning" (Gleitman and Liberman, 1995). It is also covered in many other areas beside linguistics, such as neuroscience or psychology, but in the following we restrict our considerations to written or spoken communication from a natural-language processing viewpoint. For the majority of the population, it is hard to imagine what it would be like without communication. The ability to communicate is so ubiquitous that the importance of its social function is rarely noticed. The need for information exchange is natural. But there are also people who lost or even never had the ability of communicating verbally due to physical disabilities. It does not matter whether this is caused by an accident or disease, the resulting loss affects everyday life immensely. The speech impairment is often correlated with severe motor impairment. Thus, usual computer keyboards are not sufficient to help these people to enter texts and alternative text entry devices have to be developed instead.

1.1 Motivation

The area of *Augmentative and Alternative Communication* (AAC) deals with the previously mentioned problems and forms an interdisciplinary research area that combines several domains, like e.g. educational sciences, clinical psychology, speech-language pathology, speech synthesis, sociology and psycholinguistics (Loncke et al., 1999). The main attention of this thesis is turned to highly ambiguous keyboards that enable the handicapped user to type unrestricted text (or ideally operate whole computer programs) with only a few keys. The research area that deals with this task is usually known under the name of *predictive text entry* or more specifically *text entry with ambiguous keyboards* and can be situated within a series of workshops on natural language processing for communication aids (cf. (ACL97-WS, 1997; Boguraev et al., 1998; Loncke et al., 1999; EACL03-WS, 2003)). An ambiguous keyboard maps several letters to a single key and thus reduces the number of physical switches or keys needed for typing, but has the disadvantage of making additional disambiguation

steps necessary for resolving the correct target word out of the possible candidates matching the ambiguous code sequence that has been entered.

Up to now, there are quite a few text entry programs that help the user to enter texts by *completing* or *predicting* words from already entered code prefixes and hence accelerate the typing process in terms of keystroke savings. Nevertheless, these prediction lists may change with each new key being entered and also give only a limited view on all possible candidates. This results in a high cognitive load for the user since she has to scan the prediction list for the target word after every keypress. It turned out that a person who uses a prototype of a predictive text entry developed at the University of Koblenz-Landau almost never used the predictions of the system and rather typed the whole word and concentrated on the list with exact matches after that (personal communication). This observation motivated the main question of this thesis: instead of concentrating on a word-wise predictive text entry, how about investigating the process of ambiguously entering text *sentence-wise* and go through the candidate lists afterwards to make additional disambiguations if necessary? The results of the evaluation undertaken in this work look quite promising.

The basic idea of the approach is to use *syntactic* information on a sentence level to adjust the candidate lists of the corresponding target words. Common word completion software only considers a small context of already entered words (usually the previous one or two words) for arranging more likely words to come up at the top of the current suggestion list. With a sentence-wise approach, the process of entering a text is not interrupted by searching lists of word proposals as typing advances, but rather split into two phases. During the first phase, the user can concentrate on *what* she wants to express, and in the second phase, the intended words are selected from the match lists. The advantage of this kind of approach lies in the local dependencies that are postulated by the words. These syntactic relations can be used to help to adjust a sentence-based language model and provide the user with better candidate lists during the final disambiguation. For this procedure prove itself to be feasible, it needs additional evaluation sessions with real users to determine its practicability. Nevertheless, this approach gives a new point of view on ambiguous text entry.

1.2 Outline of this work

The area of AAC, ambiguous keyboards and predictive text entry is introduced in Chapter 3. It gives a short overview on the history of this research area and presents several frameworks for text entry systems using word prediction. Furthermore, it introduces a prototype of a communication aid with a highly reduced keyboard using only four keys which is used as an evaluation basis for the procedures presented in this study.

The basic characteristic of an ambiguous keyboard is that several words are coded

by the same key sequence since the keys contain sets of letters and thus, by pressing a single key, these letters are activated in parallel. The resulting list of matches can be ordered according to a *language model* such that frequently used words appear at the top and less frequently used words are placed at the bottom. This is the easiest approach and is thus referred to as the *baseline*. Language models, as the name constitutes, try to model the properties of a natural language. Ordering the words by frequencies of use is a very simple model but also the most intuitive one. More complex (and usually much better) models comprise n-grams or Hidden Markov Models (HMMs) which are described in Chapter 2. The idea of n-grams is to base the ordering of the candidates according to the context, i.e. the history, of already typed text. HMMs and related procedures, among other things, allow for finding the most likely "reading" of a sequence of observations. In automatic speech recognition (ASR), for example, HMMs are used to find the best sequence of recognized words from an utterance (Rabiner et al., 1996). These approaches to language modeling are statistically based. All probabilities of specific word occurrences given some context are estimated from large corpora that are considered to represent the language as close as possible. The employment of these techniques helps to filter out unlikely word-sequence hypotheses that are produced by the ambiguous keyboard on a sentence level.

Another way of approaching this area is to analyze the syntactic structures and use the results to again reorder the hypotheses and find the most consistent ones. Since parsing of sentences is a rather slow and complex process and due to error proneness if the sentences are ungrammatical, it is not applicable to this domain. Instead, a fast procedure that finds "working islands" within the sentence hypotheses is more desirable. *Supertagging* and the related *lightweight dependency analysis* (Srinivas, 1997a) provide the means of applying a robust and fast *shallow parse* to a sentence. So from a rank-ordered list of possible sentence hypotheses that are generated from a code sequence typed with an ambiguous keyboard, supertagging and lightweight dependency analysis ought to be able to pick the most promising ones. The adjustment of the candidate lists according to this model should give a better accuracy of the overall text entry system and thus speed up the sentence-wise typing of texts. The syntactic information needed for the lightweight dependency analyzer (LDA) is obtained from a lexicalized tree adjoining grammar (LTAG) where every lexical item has a corresponding tree that codes its local dependencies within the context of a sentence.

The basic goal of Chapter 4 is to introduce the theoretical background that is needed for the final implementation of an n-best supertagger whose implementational aspects are described in detail in Chapter 5. The n-best supertagger is based on a supertagger developed for the German language (Bäcker, 2001; Bäcker, 2002). It determines the top n supertag hypotheses for a coded sentence, i.e. a sentence that is entered with an ambiguous keyboard. At this point, a *supertag* can be seen as an

elementary unit that stands for a certain word and also specifies constraints in what context this word may appear. The output of this modified supertagger is processed with an LDA in order to retain only the most promising hypotheses. These are used to adjust the frequency-ordered candidate lists such that more likely words are boosted to the top of the list, resulting in decreased selection costs for the user.

Chapter 6 presents the results of applying the framework to a domain-specific corpus. Since an LTAG is needed for the LDA, an already available corpus and LTAG for the German language were chosen for the evaluation. The LTAG was developed in (Bäcker, 2002) for a small corpus extracted from newsgroups that deals with hardware problems (monitors and hard disks). Due to the limited time of this work, it has not been possible to also evaluate larger corpora, e.g. parts of the British National Corpus (BNC), because the language model is estimated on an *annotated* corpus and annotating corpora with supertags, i.e. parsing them with a TAG parser and selecting the correct parse trees, is a quite time-consuming job. Although the n-best supertagger improves the accuracy of the overall system, i.e. more correct words are ranked higher than with the baseline approach, most of the performance comes from the statistical methods being used and not, as expected, from the LDA step (which can be considered to be a rule-based approach).

The last chapter gives a summary of the achievements and also presents ideas for further work. The first improvement, e.g., could be the use of larger corpora for the evaluation in order to gain better language models. Additionally, possible drawbacks, like e.g. the strong assumption that no errors are produced while typing the sentence, are noted and further analyzed. Finally, the appendix gives some UML class diagrams of the main components of the system and also lists all evaluation graphs.

1.3 Organization

In Chapter 2, statistical language models, in particular Hidden Markov Models and n-grams, are introduced together with discounting and smoothing techniques that are used to reduce the problems that arise with sparse data. Chapter 3 gives an introduction to the domain of augmentative and alternative communication and predictive text entry systems, as well as the presentation of a highly ambiguous keyboard with only four keys. Partial parsing, especially supertagging and lightweight dependency analysis, and search techniques for determining more than one good sentence hypothesis are described in Chapter 4. The implementation of the overall system is presented in Chapter 5, whereas the results obtained from a small evaluation corpus are discussed in Chapter 6. Final comments are stated in Chapter 7. The Appendix lists further evaluation graphs and UML class diagrams of the system's components.

2 Statistical Language Modeling

When dealing with statistical language processing, Hidden Markov Models (HMMs, see e.g. (Rabiner, 1989; Bahl et al., 1983)) often are the favorite modeling tool of researchers, especially in the field of speech recognition. In the speech generation process, the talker's larynx and vocal tract produce a continuous speech signal in form of an acoustic waveform realizing a message that has to be decoded by the listener (for a general introduction to phonetics, see e.g. (Lieberman and Blumstein, 1988)). In speech recognition, this is achieved by extracting features from the speech signal that allow a clear distinction between the different speech sounds. Often, the signal is ambiguous when looking at a fixed time interval, i.e. different words can produce the same signal. As an example, the expressions *the sky* and *this guy* almost have the same speech signal but a totally different meaning (from (Jurafsky and Martin, 2000)). This compares to the task of ambiguous typing in a similar manner. Here, the user "codes" the message by typing the words of the sentence with the ambiguous keyboard. In order to improve the performance of selecting the correct word from the list of matching words, the computer can use already disambiguated words as an information source that provides features of the current context within the sentence. These features help to sort the candidate list such that according to our language model, more probable words, i.e. words that are syntactically and semantically more appropriate at that position, come up first.

In general, real-world signals are characterized by constructing *signal models* that can be mathematically investigated and try to reflect the statistical properties of the original source. Statistical signal processes are e.g. Gaussian, Poisson or Markov processes. The same holds for written language. The sentences of a large corpus can be used to determine a *language model* that represents the statistical properties of that language, like e.g. the frequency of the word *flower* or the likelihood of the phrase *I like* being followed by the word *swimming*. A language model therefore assigns probabilities to words or sequences of words and can be used for prediction of the next word for a given phrase or guide the search among the many hypotheses that are generated when disambiguating code sequences on a sentence level (cf. Chapter 3).

In the following, Section 2.1 introduces discrete Markov processes and HMMs, with emphasis on the Viterbi algorithm which determines the best state sequence through an HMM given some observation sequence. In Section 2.2, n-grams are presented together with discounting and back-off methods used in the framework of the supertagger. Additionally, the notion of entropy is briefly given at the end of this chapter.

2.1 Markov models

In the following sections, probabilistic models are introduced that are used very often in speech recognition. They are capable of adequately representing the statistical properties of a language. *Markov models*[1] have been used by Andrei A. Markov at the beginning of the 20^{th} century to model the letter sequences of works in Russian literature (from (Manning and Schütze, 2000)). So they actually have a linguistic background but can nevertheless be used as a general tool in other research areas that deal with statistical models. The formal notation of the following sections is based on the one used in (Rabiner, 1989) and (Manning and Schütze, 2000).

2.1.1 Discrete Markov processes

A discrete Markov process is a system that can be in one of N distinct states $\{S_1, \ldots, S_N\}$ at any discrete time instant $t = 1, 2, \ldots, T$. The current state at time t is denoted by q_t and the process of changing from one state S_i to another state S_j is associated with a conditional probability

$$a_{ij} = P(q_t = S_j | q_{t-1} = S_i), \qquad 1 \leq i, j \leq N \tag{2.1}$$

which is called a *state transition probability*. Here, the condition "$q_{t-1} = S_i$" can be understood as the *context* where the new event occurs in state q_t. The set of all state transition probabilities forms a matrix

$$A = \{a_{ij}\} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{pmatrix}$$

with the constraints

$$0 \leq a_{ij} \leq 1, \quad \forall i, j \quad \text{and} \quad \sum_{j=1}^{N} a_{ij} = 1, \quad \forall i.$$

The discrete Markov process can be easily rewritten in terms of a finite state automaton (cf. (Charniak, 1993)) whose arcs are labeled with the probabilities from matrix A. Equation 2.1 is a direct consequence of what is known as the *Markov properties*:

- Limited horizon:
  The events that happened before a time instant $t - 1$ are not relevant to the current state (*memory-less* process).

$$P(q_t = S_i | q_1, q_2, \ldots, q_{t-1}) = P(q_t = S_i | q_{t-1}) \tag{2.2}$$

[1] also Markov chains or Markov processes

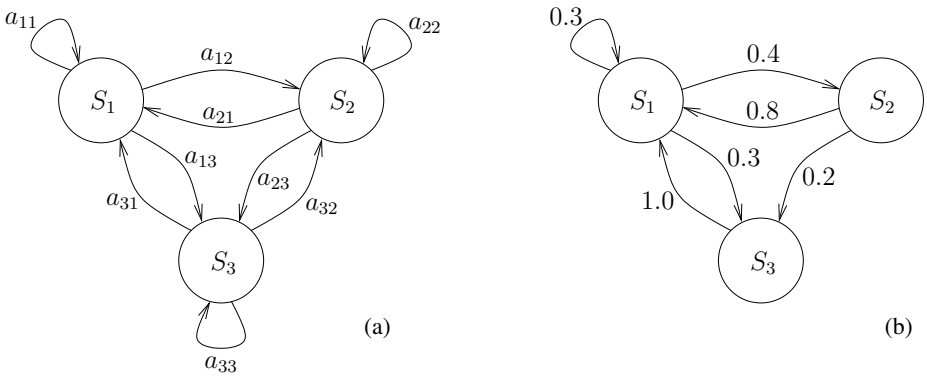

Figure 2.1: (a) A simple generic Markov model with 3 states. (b) The Markov model exemplified with annotated probabilities.

- Time invariant:
  The behavior of the Markov model is *stationary* over time, i.e. one can not conclude the time instant from observation of the states.

$$P(q_t = S_i|q_{t-1}) = P(q_{t'} = S_i|q_{t'-1}), \qquad t' = t + \Delta t \tag{2.3}$$

A simple Markov model is shown in Figure 2.1. Usually, transitions with zero probabilities, i.e. where $a_{ij} = 0$, are left out and the corresponding arcs are not shown in the state diagram. Since we know what state we currently are in at any time instant, the output of the Markov model can be regarded as a visible sequence of states. The probability of this *output sequence* $O = (q_1, q_2, \ldots, q_t)$ is

$$\begin{aligned} P(O|\lambda) &= P(q_1, q_2, \ldots, q_t|\lambda) \\ &= P(q_1)P(q_2|q_1)P(q_3|q_1, q_2)\cdots P(q_t|q_1, q_2, \ldots, q_{t-1}) \\ &= P(q_1)P(q_2|q_1)P(q_3|q_2)\cdots P(q_t|q_{t-1}) \\ &= \pi_{q_1}\prod_{i=1}^{t-1} a_{q_i q_{i+1}} \end{aligned} \tag{2.4}$$

with $\lambda = (A, \Pi)$ being the Markov model consisting of matrices A and Π, the latter denoting the *initial state probabilities*

$$\pi_i = P(q_1 = S_i) \qquad 1 \leq i \leq N. \tag{2.5}$$

Similar to the sum of the state transition probabilities of any state, the constraint $\sum_{i=1}^{N} \pi_i = 1$ holds.

2.1.2 Hidden Markov Models

As we have seen in the previous section, each state of a discrete Markov model corresponds to an observable event. If this event is not visible, the model is called a Hidden Markov Model (HMM), i.e. the observation results in not knowing exactly what state the model is in. This behavior can be described through a doubly embedded stochastic process. On the one hand, the state transition probabilities are the same as in the discrete Markov model, and on the other hand, another stochastic process exists that defines the probabilities of the final observation sequence depending of what state it was generated in. As a more intuitive example, consider an experiment where differently colored balls are pulled out of several urns behind a curtain, i.e. it is not known out of which urn a ball was chosen (example from (Rabiner, 1989)). The first stochastic process is given by the transition probabilities between the states that are associated with the urns. The second stochastic process characterizes the final outcome, i.e. it holds the probabilities that define the likelihood of the color being drawn from the urn at the current state. The probabilities of this second stochastic process are called *observation symbol* or *symbol emission probabilities* and are denoted by

$$b_j(k) = P(v_k \text{ emitted at time } t | q_t = S_j) \qquad 1 \leq j \leq N,\ 1 \leq k \leq M, \tag{2.6}$$

where $v_k \in V = \{v_1, \ldots, v_M\}$ is the observed symbol out of a set V of possible observation symbols. The probability distribution in Equation 2.6 can again be summarized in terms of a matrix

$$B = \{b_j(k)\} = \begin{pmatrix} b_1(1) & b_1(2) & \cdots & b_1(M) \\ b_2(1) & b_2(2) & \cdots & b_2(M) \\ \vdots & \vdots & \ddots & \vdots \\ b_N(1) & b_N(2) & \cdots & b_N(M) \end{pmatrix}$$

where the columns contain the probabilities of a single observation symbol for all states and the rows hold the probabilities for all observation symbols that are emitted in a single state. Table 2.1 shows the summary of all elements of a Hidden Markov Model $\lambda = (A, B, \Pi)$.

So let's enhance the example presented in Figure 2.1 (b) with an output probability distribution for two possible observation symbols $V = \{b, w\}$ representing the color (black or white) of the balls introduced in the example above. The states S_1, S_2 and S_3 correspond to three urns which are filled with black and white balls. The observation symbol probabilities are shown in Figure 2.2 (b). Given this model and an observation sequence $O = (b, b, w, b, w, w)$, several basic questions can be asked, namely what is the probability of this observation sequence, what states does the HMM pass through most likely and how does one come up with an optimized model λ that best explains the observation sequence?

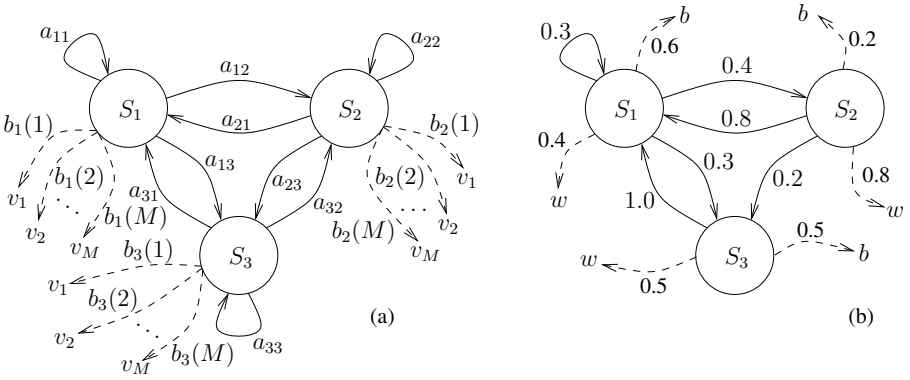

Figure 2.2: (a) A generic 3-state Hidden Markov Model with M possible output symbols. (b) The HMM representing the urns and balls example. It is assumed that each state is equally likely to be the starting state, i.e. $\pi_i = \frac{1}{3}$ for $i \in \{1, 2, 3\}$.

More formally, what follows are the three fundamental problems that emerge in the field of HMMs (cf. (Rabiner, 1989)):

1. Given a model $\lambda = (A, B, \Pi)$ and an observation sequence $O = (o_1, o_2, \ldots, o_T)$, how is $P(O|\lambda)$ computed efficiently?

2. Given a model λ and an observation sequence O, how can the optimal state sequence $Q = (q_1, q_2, \ldots, q_T)$ be obtained that best explains O, i.e. that maximizes $P(Q|O, \lambda)$?

3. Given an observation sequence O, how can the model parameters of $\lambda = (A, B, \Pi)$ be adjusted such that they maximize $P(O|\lambda)$?

Efficient solutions exist to all three problems and are described more detailed in (Rabiner, 1989) and (Manning and Schütze, 2000). All algorithms make extensive use of a technique called *dynamic programming* (see e.g. in (Cormen et al., 1990)) that is necessary in order to reduce the amount of calculations and is often applied

Set of N states	$S = \{S_1, S_2, \ldots, S_N\}$	
Set of M output symbols	$V = \{v_1, v_2, \ldots, v_M\}$	
Initial state probabilities	$\Pi = \{\pi_i\}$	$1 \leq i \leq N$
State transition probabilities	$A = \{a_{ij}\}$	$1 \leq i, j \leq N$
Observation symbol probabilities	$B = \{b_j(k)\}$	$1 \leq j \leq N, 1 \leq k \leq M$

Table 2.1: The elements of an HMM $\lambda = (A, B, \Pi)$.

to optimization problems. The general trick is to speed up the computation by storing previous results that solve smaller instances of the same problem. All problems to which dynamic programming can be applied must satisfy a *decomposition property*: an optimal solution to the problem decomposes into an optimal solution to a *smaller* instance of the *same* problem. Sometimes, this is also called the principle of optimality. The dynamic programming framework consists of four steps:

1. Characterizing the recursive structure of an optimal solution.

2. Deriving a recurrence for the value of an optimal solution by using the characterization from step 1.

3. Computing the solution value from the recurrence bottom-up, tabulating intermediate solution values.

4. Recovering an optimal solution from the table of values.

Solution to problem 1 (forward procedure)

Problem 1 of the basic HMM problems can be solved by applying the forward procedure of the *forward-backward* algorithm (Baum, 1972). In order to compute $P(O|\lambda)$ efficiently, the probabilities of partial observation sequences until a certain time t that end in state i are stored in a forward variable $\alpha_t(i) = P(o_1, o_2, \ldots, o_t, q_t = S_i|\lambda)$. The stored values for the probabilities at time $t-1$ can then be used to calculate the probabilities at the next time frame t inductively by

$$\alpha_t(j) = \Big[\sum_{i=1}^{N} \alpha_{t-1}(i)a_{ij}\Big] b_j(o_t) \qquad 2 \leq t \leq T,\ 1 \leq j \leq N, \tag{2.7}$$

whereas for $t = 1$, $\alpha_1(j) = \pi_j b_j(o_1)$ is used. The fact that previous α-values are summed over all states originates from the observation that $P(O|\lambda)$ is the sum of the joint probabilities over all possible state sequences, i.e.

$$P(O|\lambda) = \sum_{\text{all } Q} P(O|Q, \lambda)P(Q|\lambda). \tag{2.8}$$

The dynamic programming approach reduces the complexity of the calculation from $O(2TN^T)$ to $O(N^2T)$ (T is the length of the observation sequence, N is the number of HMM states). The final probability is computed as $P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$. For a detailed derivation, see (Rabiner, 1989).

Solution to problem 2 (Viterbi algorithm)

The second fundamental problem is the question of finding the optimal state sequence of the HMM for an observation sequence $O = o_1, o_2, \ldots, o_T$. The solution to this problem depends on the way we define the term "optimal". One could consider to maximize the probability for all states at each time frame $1, 2, \ldots, T$ individually, resulting in a maximization of the expected number of correct states. But this approach results in an invalid state sequence if there are state transitions with zero probability, i.e. where $a_{ij} = 0$. In order to avoid this, the optimality criterion could be extended to pairs (q_t, q_{t+1}) or triples (q_t, q_{t+1}, q_{t+2}) of states. However, the criterion most commonly used is to find the *single* best state sequence $Q = q_1, q_2, \ldots, q_T$ such that $P(Q|O, \lambda)$ is maximized. Again, the application of the dynamic programming principle allows for a computationally feasible solution. The procedure was first presented in (Viterbi, 1967) and is therefore called the *Viterbi* algorithm. In order to derive the recursive structure of the problem (step 1 of the dynamic programming framework), it has to be shown how an optimal solution *ends*. For the last time frame T, it ends in the best state S_m that maximizes the probability of the optimal state sequence computed so far, or more formally

$$P(Q, O|\lambda) = \max_{q_1, q_2, \ldots, q_T} P(q_1, q_2, \ldots, q_T = S_m, O|\lambda). \tag{2.9}$$

This can be computed more efficiently if intermediate highest probabilities along a single path at time t that end in state S_i are stored in a special variable (or table) δ, which is defined as

$$\delta_t(i) = \max_{q_1, q_2, \ldots, q_{t-1}} P(q_1, q_2, \ldots, q_{t-1}, q_t = S_i, o_1, o_2, \ldots, o_t|\lambda). \tag{2.10}$$

Using Equation 2.10, we can derive a recurrence for the problem (cf. step 2) which accounts for the first t output symbols of the observation sequence O:

$$\delta_t(j) = \max_{1 \leq i \leq N} \left[\delta_{t-1}(i) a_{ij}\right] b_j(o_t). \tag{2.11}$$

This equation differs from Equation 2.7 only in the point that it uses maximization instead of a summing criterion. Both procedures use a *lattice* or *trellis structure* for the computations.[2] Figure 2.3 shows a generic trellis for HMMs and the trellis structure for the example presented in Figure 2.2 (b).

The δ-values that have been computed so far only yield the highest probability $p^* = \max_{1 \leq i \leq N} \delta_T(i)$ for the best state sequence ending in state $q_T^* = \operatorname{argmax}_{1 \leq i \leq N} \delta_T(i)$. In order to retrieve the actual states of the whole optimal path, another table is

[2] In fact, *all* solutions to problems using the dynamic programming framework implement this sort of trellis.

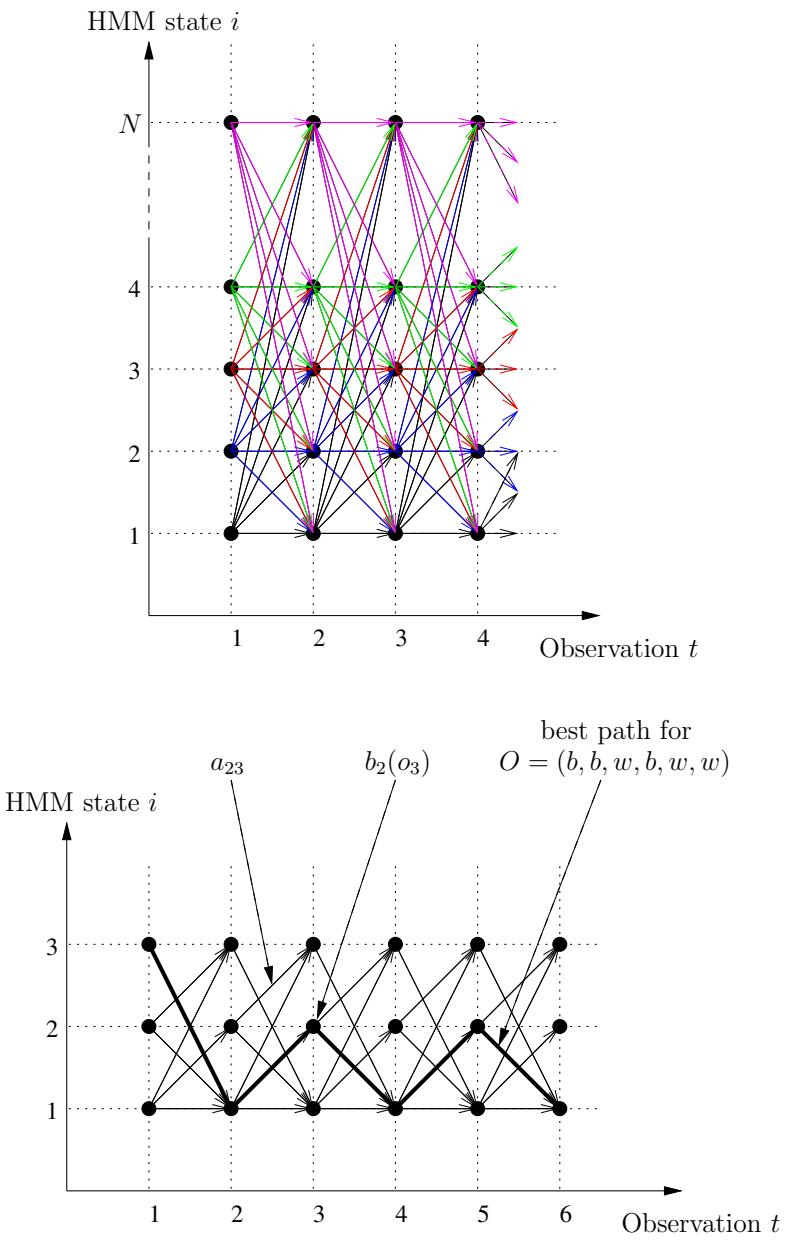

Figure 2.3: The trellis (or lattice) used in the Viterbi calculation of the δ-values. The upper structure shows a generic trellis for N states and the first four time frames of the observation sequence, whereas the lower one shows the trellis of the urns and balls HMM from Figure 2.2 (b) with annotated examples where to find the state transition and observation symbol probabilities. The best path for $O = (b, b, w, b, w, w)$ is highlighted.

needed that stores the arguments which maximize Equation 2.11. This is achieved through the variable

$$\psi_t(j) = \underset{1 \le i \le N}{\operatorname{argmax}} \left[\delta_{t-1}(i) a_{ij}\right]. \tag{2.12}$$

Step 3 of the dynamic programming framework is applied by computing the δ- and ψ-values iteratively for all time frames $t = 2, 3, \ldots, T$ after an initialization of the variables with $\delta_1(i) = \pi_i b_i(o_1)$ and $\psi_1(i) = 0$ for all states $i = 1, 2, \ldots, N$. Finally, step 4 recovers the optimal state sequence $Q^* = (q_1^*, q_2^*, \ldots, q_T^*)$ from the ψ-table by backtracking:

$$q_t^* = \psi_{t+1}(q_{t+1}^*) \qquad t = T-1, T-2, \ldots, 1. \tag{2.13}$$

As mentioned above, this algorithm finds the optimal sequence of urns for an observation of drawn balls (cf. the example HMM in Fig. 2.2 (b)). For $O = (b, b, w, b, w, w)$, it returns the best path $Q^* = (S_3, S_1, S_2, S_1, S_2, S_1)$. The algorithm is summarized in Figure 2.4.

Solution to problem 3 (forward-backward algorithm)

The solution to problem 3 of HMMs is the most difficult one. It estimates the model parameters A, B and Π by maximization of the observation's probability $P(O|\lambda)$ through

$$\underset{\lambda}{\operatorname{argmax}}\, P(O_{\text{tr}}|\lambda). \tag{2.14}$$

The technique used for this task is known as the *Baum-Welch* method or *forward-backward* algorithm (Baum, 1972) and reestimates the model parameters iteratively[3] because an analytical solution of deriving optimal model parameters has not been found so far. The parameters for $\lambda = (A, B, \Pi)$ are usually estimated on some training data O_{tr} which may reflect properties of the actual observation sequence O. The idea is that by creating a revised model of λ that fits best the observations contained in O_{tr}, it should also return higher probabilities for O. This is achieved by increasing the probabilities of the state transitions and symbol emissions that happen to be used most in the HMM. Note that the Baum-Welch method only "tweaks" the probabilities in A, B and Π. The architecture of the HMM is not altered. The algorithm is usually run until a convergence criterion is met and therefore only finds locally optimal models, depending on the (possibly randomly chosen) initial values of λ. Since the model parameters of the HMM used in this thesis can be directly estimated on the annotated training corpus, the application of this reestimation method is not needed and a more detailed presentation is left out. For a deeper introduction, see (Rabiner, 1989).

[3] Actually, this procedure is a special case of the *Expectation-Maximization* (EM) algorithm (Dempster et al., 1977).

1. Initialization:

$$\begin{aligned} \delta_1(i) &= \pi_i b_i(o_1) && 1 \leq i \leq N \\ \psi_1(i) &= 0 && 1 \leq i \leq N \end{aligned}$$

2. Recursion:

$$\begin{aligned} \delta_t(j) &= \max_{1 \leq i \leq N} \left[\delta_{t-1}(i) a_{ij}\right] b_j(o_t) && 2 \leq t \leq T, \quad 1 \leq j \leq N \\ \psi_t(j) &= \operatorname*{argmax}_{1 \leq i \leq N} \left[\delta_{t-1}(i) a_{ij}\right] && 2 \leq t \leq T, \quad 1 \leq j \leq N \end{aligned}$$

3. Termination:

$$\begin{aligned} p^* &= \max_{1 \leq i \leq N} \left[\delta_T(i)\right] \\ q_T^* &= \operatorname*{argmax}_{1 \leq i \leq N} \left[\delta_T(i)\right] \end{aligned}$$

4. Path (state sequence) backtracking:

$$q_t^* = \psi_{t+1}(q_{t+1}^*) \qquad t = T-1, T-2, \ldots, 1$$

Figure 2.4: The Viterbi algorithm for finding the optimal state sequence of an HMM $\lambda = (A, B, \Pi)$ for an observation sequence $O = o_1, o_2, \ldots, o_T$ (from (Rabiner, 1989)).

2.2 N-gram models

The relative word frequency that is going to be used in Section 3.4 to order the candidates of the ambiguous keyboard is a very simple model of the language. If the dictionary is based on m word tokens and n word types of a corpus, the distinct words w_i are ordered according to the probability

$$P(w_i) = \frac{C(w_i)}{\sum_{j=1}^{n} C(w_j)} = \frac{C(w_i)}{m} \qquad 1 \leq i \leq n, \tag{2.15}$$

where $C(w_i)$ denotes the frequency of word w_i. It does not take into account what context the words actually appear in. If the user typed *the* and the next word has the same code, it does not make much sense to predict another *the* at the first position of the match list. It is better to offer nouns or adjectives at first place. So instead of using the probability of a single word for determining its rank in the match list, one can use the probability of n-grams for this task. When dealing with n-grams, the probability of the current word is based on the *history* h of $n-1$ words, or more formally

$$P(w_n|h) = P(w_n|w_1 w_2 \cdots w_{n-1}). \tag{2.16}$$

For the already known case of a single word, $n = 1$ and we also speak of *unigram* probabilities which ignore the context entirely. For larger values ($n = 2, 3, 4$), the n-grams are referred to as bigrams, trigrams and four-grams, respectively. It is obvious that the larger the value of n gets, the less predictive power the model provides since it is very unlikely that the sentence on which the history is based on will occur multiple times (sparseness of data, cf. (Manning and Schütze, 2000)). Therefore, one has to group the history into several equivalence classes based on the Markov assumption presented in Equation 2.2, i.e. taking into account only the last few words. For n-grams, the history of length $n-1$ is a Markov model of order $n-1$.

The most simple estimator that can be used for computing the probabilities of n-grams is *maximum likelihood estimation* (MLE). Let N denote the overall number of n-grams available for training[4] and $C(w_1 \cdots w_n)$ be the frequency of a specific n-gram consisting of the word sequence $w_1 \cdots w_n$. The maximum likelihood estimate is defined by

$$P_{\text{MLE}}(w_1 \cdots w_n) = \frac{C(w_1 \cdots w_n)}{N} \tag{2.17}$$

for the likelihood of the word sequence $w_1 \cdots w_n$ and

$$P_{\text{MLE}}(w_n|w_1 \cdots w_{n-1}) = \frac{C(w_1 \cdots w_n)}{C(w_1 \cdots w_{n-1})} \tag{2.18}$$

[4] If we add $n-1$ empty symbols, so called *pseudo categories* (e.g. $\varnothing$), at the beginning, the first n-gram actually starts with the first word of the training data and therefore N is the overall number of words in the text.

for the likelihood of the word w_n given the history of previous words $w_1 \cdots w_{n-1}$. The former equation is used to estimate the probability of n-grams, whereas the latter provides a way of finding the most likely word for prediction. As mentioned above, sparseness of data does not make this approach suitable for long n-grams because many word sequences will not appear in the training data and end up with a zero probability. For the case of the Viterbi algorithm presented in Section 2.1.2, this means that an unknown n-gram propagates the zero probability until the end of the observation sequence. Since the probabilities along the trellis path are multiplied, the whole sequence finally has a probability of zero. As reported by (Bahl et al., 1983), training on a 1.5 million corpus resulted in 23% of the test data trigrams being unknown. One way of decreasing this percentage is to increase the size of the training corpus. Nevertheless, it is not possible to handle this problem entirely since there will always be instances of very rare events that are not covered by the training data.

A common way of improving the maximum likelihood estimation is to add a small positive value to the frequency counts $C(w_1 \cdots w_n)$ such that the numerator of Equation 2.17 cannot be zero any more. If the n-grams are examined under the viewpoint of a random variable X that can take different values x_i, the maximum likelihood estimation can be restated as

$$P(X = x_i) = \frac{v_i}{\sum_{i=1}^{V} v_i}, \tag{2.19}$$

where $v_i = |x_i|$ is the relative frequency of event x_i and V is the overall number of equivalence classes being considered, i.e. V is the number of different n-grams encountered in the training data. In order to prevent the zero probability problem, a small value λ is added to v_i:

$$P(X = x_i) = \frac{v_i + \lambda}{\sum_{i=1}^{V}(v_i + \lambda)} = \frac{v_i + \lambda}{\sum_{i=1}^{V} v_i + V\lambda}. \tag{2.20}$$

For the common value of $\lambda = \frac{1}{2}$, this equation is also known as *expected likelihood estimation* (ELE) or Jeffrey-Perks law. In terms of Equation 2.17, it can be written as

$$P_{\mathrm{ELE}}(w_1 \cdots w_n) = \frac{C(w_1 \cdots w_n) + \lambda}{N + V\lambda}. \tag{2.21}$$

The use of n-grams as a language model has been thoroughly investigated in the past. With the development of faster computer technology and the creation of huge corpora, it has been made possible to process larger and larger amounts of data in less time. Actually, statistical approaches that utilize very large corpora have formed a separate research domain (see e.g. (Armstrong et al., 1999)). Powerful language modeling tools are freely available and can be easily used to experiment with corpora,

like e.g. the CMU-Cambridge Statistical Language Modeling Toolkit[5] (Clarkson and Rosenfeld, 1997) or the SRILM toolkit[6] (Stolcke, 2002).

In addition to ELE, one usually uses *smoothing* or *discounting* techniques which lower the probability of common events in order to increase the probability of rare events, i.e. a part of the probability mass is reserved for events that were previously unseen. The discounting techniques implemented by the supertagger used for this work (Good-Turing discounting) and other techniques that fall back to n-grams for smaller values of n (Katz back-off) are presented in the following sections.

2.2.1 Discounting

As already noted, a fundamental problem when dealing with statistical techniques for language processing is sparseness of data. It is very likely that the test corpus contains trigrams that have not been encountered in the training data before. These trigrams end up with a zero probability (when using MLE) or a very low probability (with ELE), thus resulting in a wrong estimate for a whole sequence which is not desirable.[7] Therefore, *discounting* techniques are often applied. The basic idea is to lower the counts of frequent events and preserve some of the overall probability mass for the unseen trigrams that have a zero frequency. The MLE approach used in Equations 2.17 and 2.18 estimates the probabilities of words by using their relative frequencies. The same equations hold for other types of linguistic units, such as POS tags or supertags (cf. Sections 4.1 and 4.3). In the following formulas, the t_i's refer to supertags because of their direct relation to the actual implementation within the supertagger.[8]

One of the discounting strategies that is used quite often is *Good-Turing* discounting (Good, 1953). The smoothed version of Equation 4.7 on page 40 (which denotes the probability of a supertag trigram, cf. also Equation 2.18 with $n = 3$) is

$$\tilde{P}(t_i|t_{i-2}t_{i-1}) = \frac{C^*(t_{i-2}t_{i-1}t_i)}{C(t_{i-2}t_{i-1})}, \tag{2.22}$$

where $C^*(t_{i-2}t_{i-1}t_i)$ (or abbreviated c^*) denotes the discounted frequency of the supertag triple $t_{i-2}t_{i-1}t_i$. In general, let N_c be the number of N-grams that occur c times. This value is referred to as the frequency of frequency c (also known as *count-counts*) and it is used by the Good-Turing estimate for unseen events as follows:

[5] `http://mi.eng.cam.ac.uk/~prc14/toolkit.html`

[6] `http://www.speech.sri.com/projects/srilm/`

[7] If we look at the MLE formulas, the whole sequence results in a zero probability because of the multiplication. By using logprobs (cf. Section 4.3.2), this effect is avoidable since the expressions are now summed and zero probabilities do not "break up" the whole sequence any more. Still, the overall estimates are distorted.

[8] At this point, one could still consider the t_i's to stand for words w_i if the familiarity with supertags is not fully established yet.

the smoothed count c^* for an N-gram that never occurred ($c = 0$) is obtained by dividing the number of N-grams that occurred once (N_1) by the number of N-grams that never occurred (N_0). The same holds for N-grams that have been seen only one time. Their discounted frequency is based on those N-grams which occurred two times and so on. So, the Good-Turing estimate that gives a discounted frequency c^* is calculated by

$$c^* = (c+1)\frac{N_{c+1}}{N_c}. \tag{2.23}$$

Usually, large counts (e.g. $c > 5$) are considered to be reliable. Therefore, a threshold k is introduced and the smoothed value is set to the original relative frequency $c^* = c$ for $c > k$. With this approach, it is necessary to renormalize the discounting estimates. The final equation for Good-Turing discounting with threshold k is (from (Jurafsky and Martin, 2000)):

$$c^* = \frac{(c+1)\frac{N_{c+1}}{N_c} - c\frac{(k+1)N_{k+1}}{N_1}}{1 - \frac{(k+1)N_{k+1}}{N_1}} \qquad \text{for } 1 \leq c \leq k. \tag{2.24}$$

2.2.2 Back-off

So far, the Good-Turing discounting strategy helps us to avoid the zero probability problem for unknown n-grams. Another strategy that is commonly used is so-called *back-off*. The idea behind this is to rely on lower order $(N-1)$-grams if the counts for the N-grams come out to be zero. The chain can be continued progressively down to unigrams if necessary. For the case of trigrams, the back-off formula that was introduced by (Katz, 1987) looks as follows:

$$P_{\text{KBO}}(t_i|t_{i-2}t_{i-1}) = \begin{cases} \tilde{P}(t_i|t_{i-2}t_{i-1}) & \text{for } C(t_{i-2}t_{i-1}t_i) > 0 \\ \alpha_1\tilde{P}(t_i|t_{i-1}) & \text{for } C(t_{i-2}t_{i-1}t_i) = 0 \text{ and } C(t_{i-1}t_i) > 0 \\ \alpha_2\tilde{P}(t_i) & \text{otherwise.} \end{cases} \tag{2.25}$$

The probabilities are the discounted versions of the relative frequencies (cf. Equation 2.22). In fact, backing-off a language model always requires discounting. This is due to the additional probability mass that would be added with each back-off to a lower model if the initial (undiscounted) probability was zero. Therefore, the sum of all probabilities for a supertag t_i given the two previous supertags t_{i-2} and t_{i-1} would be greater than 1, but the constraint $\sum_{j,k} P(t_i|t_jt_k) = 1$ must hold. The amount of probability mass that is actually distributed from a higher order to a lower order N-gram is represented by the normalization factors α_1 (from trigrams to bigrams) and α_2 (from bigrams to unigrams). They are computed by summing all discounted N-gram probabilities that start with the same context and subtracting the value from 1.

After that, a normalization step (the denominators in Equations 2.26 and 2.27) is applied which ensures that the lower order model only gets the correct fraction of the overall higher order model. The α-values are computed by:

$$\alpha_1 = \frac{1 - \sum_{t_i:C(t_{i-2}t_{i-1}t_i)>0} \tilde{P}(t_i|t_{i-2}t_{i-1})}{1 - \sum_{t_i:C(t_{i-2}t_{i-1}t_i)>0} \tilde{P}(t_i|t_{i-1})} \tag{2.26}$$

$$\alpha_2 = \frac{1 - \sum_{t_i:C(t_{i-1}t_i)>0} \tilde{P}(t_i|t_{i-1})}{1 - \sum_{t_i:C(t_{i-1}t_i)>0} \tilde{P}(t_i)}. \tag{2.27}$$

A more detailed derivation that also generalizes on N-grams (by introducing a function $\alpha(t_{n-N+1}^{n-1})$) is given in (Jurafsky and Martin, 2000). There exist many other discounting and back-off methods. A comprehensive overview with in-depth mathematical derivations can be found in (Ney et al., 1997; Ney, 1999).

2.2.3 Entropy

In statistical language modeling, *entropy* and the related term *perplexity* are usually used to determine the quality of the language model. The term *entropy* was defined by Claude Shannon who "invented" the field of information theory in the 1940s. Entropy is measured in bits and expresses the amount of information in a random variable X by

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x), \tag{2.28}$$

where $\mathcal{X}$ denotes the alphabet and $p(x) = P(X = x)$ is the probability mass function (pmf) for all $x \in \mathcal{X}$ (from (Manning and Schütze, 2000)). For example, if each letter of the English alphabet[9] was equally likely then its entropy would be

$$-\sum_{i=1}^{27} \frac{1}{27} \log_2 \frac{1}{27} = \log_2 27 \approx 4.76,$$

which means that each character can be coded with 4.76 bits. If we determine the letter frequencies of a large text collection, e.g. extracted from the English dictionary of the CELEX lexical database (Baayen et al., 1995), then the entropy drops down to approx. 4.17, i.e. entropy decreases with increasing order of the model.

In this context, entropy can also be used to measure the ability of some n-gram model to predict words given a history of already typed words. Since we do not know the real pmf $p(x)$ that underlies a natural language L in terms of word probabilities, the common way is to compute the *cross entropy* that uses another pmf $m(x)$ which is

[9] We only consider lower-case letters "a–z" and the space character "␣".

estimated on large corpora consisting of n word tokens (written as $x_{1n} = x_1 x_2 \cdots x_n$):

$$H(L, m) = -\lim_{n\to\infty} \frac{1}{n} \sum_{x_{1n}} p(x_{1n}) \log_2 m(x_{1n}). \tag{2.29}$$

Still, this term cannot be computed because of the unknown pmf $p(\cdot)$. If the additional assumption is made that the language is *ergodic*, i.e. that the sample length is long enough (or in other words: the sample represents the language sufficiently well), Equation 2.29 can be written as

$$H(L, m) = -\frac{1}{n} \log_2 m(x_{1n}) \tag{2.30}$$

for sufficiently large n. The perplexity of a language model is closely related to its entropy and is simply given by

$$PP(X, m) = 2^{H(X,m)} = m(x_{1n})^{\frac{1}{n}}. \tag{2.31}$$

Obviously, the real entropy of a language is unknown but we can try to derive upper bounds for the cross entropy by utilizing more and more complex language models. In (Brown et al., 1992), a language model based on trigrams is reported to reach an entropy of 1.75 bits per character when trained on approx. 583 million words. Other entropy values, as they are e.g. encountered in the domain of text compression, are reported in (Church and Mercer, 1993), together with the upper bound of 1.25 bits derived by Shannon as a result of experiments dealing with humans that guessed the next letter of a text. This clearly shows that humans easily outperform state-of-the-art language models.

In this study, the evaluation is not based on entropy since the approach that will be presented in Chapter 5 is hybrid in a sense that it makes use of statistical and rule-based techniques to adjust the candidate lists and therefore does not provide actual probabilities of the words. The evaluation measures are motivated more pragmatically and will be introduced in Section 6.2.

3 Text Entry with Ambiguous Keyboards

Communication aids play an important role for people that are not able to express themselves directly through speech or writing. The loss of these abilities can have several reasons, e.g. spinal cord injury, cerebral vascular accident (stroke), Lou Gehrig's disease or cerebral palsy. According to (Darragh and Witten, 1992), physically impaired persons can be grouped into those who are incapable of speech, those who lack the physical ability to write and those who fall into both of these groups. The latter category is the most frequent one since a disability that impairs speech usually also affects other parts of the body by diminishing their strength and dexterity, resulting in a very restricted ability to move. So "expressive" communication aids such as ambiguous keyboards have to be developed to provide the users access to communicative means.

The structure of this chapter is as follows. Section 3.1 gives a short introduction to the field of AAC and describes possible methods that exist for selecting keys on virtual keyboards, i.e. on-screen keyboards that present some kind of mapping of letters to keys that can be selected by the impaired user through physical switches. Section 3.2 presents some predictive text entry systems that offer word completions to the user, whereby the emphasis is laid on processing techniques using statistical language models. An alternative approach to text entry is to utilize *ambiguous keyboards* which are introduced in Section 3.3. In particular, a system is described that allows for entering the whole sentence ambiguously rather than disambiguating every word after it has been typed. Section 3.4 focuses on a prototype of a highly ambiguous keyboard (using only three letter keys and one command key) which is being developed at the University of Koblenz-Landau. The pilot user is Kathrin, a 16-year old girl with cerebral palsy who nevertheless attends a regular high school. Her experience with the software (and its predecessors) partly inspired the work on this thesis.

3.1 Augmentative and Alternative Communication

The area of *Augmentative and Alternative Communication* (AAC) is the research field concerned with assisted communication for people with speech impairment (Newell et al., 1998). The goals hereof depend on the primary field where the communication aid is going to be used. For writing long texts it is essential to reduce the effort of typing as much as possible, whereas for direct conversational communication, e.g.

greeting someone or making a request, the intended expression has to be issued as fast as possible. In this connection, the voice output is synthesized by a text-to-speech component (see e.g. (Venkatagiri and Ramabadran, 1995)). So the task is to develop a text entry device that has both these properties and thus improves the non-speaker's communicative skills. A possible scenario is to mount this kind of device on a wheelchair and thus provide the disabled person the flexibility and communicative mobility needed for everyday situations, e.g. going shopping or meeting friends. The social closeness to other people is an important factor against isolation because of the absence of communicative means.

Therefore, there have been many approaches to this domain and numerous text entry devices exist nowadays. The range goes from manual systems (such as letter or picture boards)[1] to high-tech dedicated voice output systems.[2] The employment of specialized hardware solutions is not only considerably expensive but also makes adjustments to the special needs of an individual hard to put into practice. More flexibility is achieved with software solutions that run on any computing device. The use of mobile devices such as ordinary or ruggedized laptops, handhelds or even mobile phones has become available to the broad masses due to sinking costs and increasing computing power. The separation of the text entry system (software) and the actual physical input methods (hardware) is reasonable since every user will have different, specifically adapted switches, keys and/or other devices such as joysticks, head- or eye-trackers (see e.g. (Tech Connections, 2002)), that she or he is able to operate. The use of the text entry program is achieved by mapping the input signals (e.g. by pressing a special switch in the headrest of a wheelchair) to general keyboard or mouse actions understood by the software. Special interfaces that transform any physical signal into a keypress of an ordinary keyboard exist for this purpose.

The rate at which speech impaired people can communicate with the help of AAC systems is directly correlated with the speed at which they can type. In (Darragh and Witten, 1992), several rates are compared with each other. An able-bodied non-professional typist usually achieves rates between 15–25 words per minute (wpm) with an ordinary keyboard, spoken conversation is even as high as 150–200 wpm. In contrast to these values, a disabled person using a single-switch scan[3] of a virtual keyboard achieves the unacceptable rate of only a few words per minute. The rates reach from 1–5 wpm, but can even drop down below 1 wpm. It is argued that 3 wpm is the absolute minimum that is tolerable for interactive conversation. Unfortunately, everything below 9 wpm promotes the impatience of the receiver and hinders the overall communicative fluency. So one of the basic goals of AAC can be seen as

[1] see e.g. Crestwood Communication Aids at `http://www.communicationaids.com/`

[2] see e.g. Prentke Romich Company at `http://www.prentrom.com/` or Words+ at `http://www.words-plus.com/`

[3] *Scanning* denotes the process of stepping through a series of options that are activated by the user when the desired item is reached.

how to increase the rates at which words are typed. Another problem arises with people who have language dysfunction (e.g. *aphasia*), i.e. they cannot access the words that are necessary for their message. In this case, iconic codings can help to use or comprehend words again (Beck and Fritz, 1998). Since the questions in this study are mainly concerned about the rate enhancement for people with speech impairment because of physical disabilities, this aspect is not further considered in the following course.

Among other things, conversational communication also comprises initiating interactions ("how are you"), requests ("could you please open the door") or corrections ("I meant the other one"). A possible way of entering such expressions fast is the use of sentence compansion (see e.g. in (Copestake, 1997; McCoy et al., 1998)). So from the input "open window", the system would generate "could you please open the window". This cogeneration approach needs three knowledge sources, namely a grammar and lexicon, statistical information about collocations, i.e. syntagmatically related words,[4] and a set of templates. A thinkable drawback for the user might be that the expanded sentences seem monotonous after some time. In contrast, flexibility and individuality are valuable features of direct and unrestricted text entry systems. Thus, the motivation of typing on a sentence level is reasonable. The idea is to make use of the syntactic relations that exist in the sentence the user wants to express and use them to present more accurate candidate lists that allow for faster selection by moving likely matches to the top.

Section 3.2 gives an overview on word completion and word prediction approaches that try to speed up the typing process by presenting the user a list of words that (partially) match the current input. For an introduction to other research fields of augmentative communication systems, see e.g. (Fazly, 2002; Langer and Hickey, 1999; Copestake, 1996). In general, a good information source yield the official journals of the International Society for Augmentative and Alternative Communication (ISAAC), namely *AAC Augmentative and Alternative Communication*, published by B.C. Decker Inc.

3.1.1 Selection methods for augmentative devices

The selection of the keys is generally possible in two ways. Keys can be directly activated by pressing the corresponding buttons or they can be scanned progressively if the handicapped person has severe impairment. The activation of a highlighted key is then performed with only one physical switch, e.g. triggered by movement of the eyelid. The scanning process of a virtual keyboard can be either in a circular manner, i.e. highlighting all items successively and starting all over when reaching the last key, or by row-/column-wise scanning (see e.g. (Demasco and McCoy, 1992)).

[4] For a detailed definition of the term, see (Manning and Schütze, 2000).

The former method can be used when there are only a few keys, as is the case for ambiguous keyboards (see e.g. (Kühn and Garbe, 2001)), whereas the latter is preferred when there are many item sets available, e.g. a virtual keyboard with each letter on a separate key or additional entries that activate macros or issue commands to the operating system. In general, entering texts by scanning is more time-consuming and fatiguing since the user needs much concentration to activate the currently highlighted key at the right time (cf. (Horn and Jones, 1996)). If two switches are available, the automatic scanning routine can be shifted to one of the switches, i.e. one switch moves the selection to the next item or item set and the other switch selects the current entry. The evaluation measures used in this thesis are independent of the selection method (cf. Section 6.2). Therefore, a more abstract evaluation criterion is obtained which allows for a general assessment of the techniques presented in this study, regardless of how the user actually operates the system.

A flexible program for creating arbitrary item sets that can trigger any functionality provided by the graphical user interface is e.g. SAW (Switch Access to Windows) by the ACE Centre Advisory Trust[5] for Microsoft Windows operating systems. They also provide a word prediction software called Prophet that can be integrated into SAW. For UNIX and UNIX-like systems (such as Linux, FreeBSD and Solaris), GNOME (GNU Network Object Model Environment)[6] is a user-friendly graphical desktop environment which has a configurable accessibility interface that enables the operation of all GUI-specific actions via simple keystrokes, like moving or resizing windows and switching the focus of applications or several virtual desktops, as well as individually adjusting keyboard or mouse control parameters.

3.2 Word completion and prediction

The two terms, namely word *completion* and word *prediction* are most often used synonymously although there is a small difference (Klund and Novak, 2001). The former denotes a system that presents a list of words as soon as the user starts typing a word. The possible candidates for completion all match the already entered prefix of the word. The latter method predicts whole words after a word has been completely entered or selected. Usually, word completion is used more frequently but is also called word prediction which often leads to confusion. If the letters that are typed are distinct from one another, i.e. they can be chosen unambiguously by e.g. using a virtual keyboard with separate keys for all letters of the alphabet and utilizing a scanning technique that first selects the rows and then the columns, the words in the candidate lists actually are completions to the current word. If

[5] `http://www.ace-centre.org.uk/`
[6] `http://www.gnome.org/`

the input technique is ambiguous, e.g. by using reduced keyboards (which will be outlined in Section 3.3), then it is also legitimate to call the process *prediction* since the ambiguous codes now allow for much more word suggestions that often have different prefixes. Since the task of arranging these candidates according to some likelihood estimate is technically the same as for completion, we will use both terms (completion and prediction) synonymously in the following. Furthermore, the process of completion is also referred to as *disambiguation* of the user input.

3.2.1 Letter-wise approaches

A large collection of assistive text entry systems is available (see e.g. (Newell et al., 1998)), partly commercially and partly developed as prototypes within research projects. Most of these systems suggest possible word completions to the user as typing progresses. A few programs do not provide a word-wise prediction but work on a letter-wise prediction instead, such as LetterWise (MacKenzie et al., 2001) or the Reactive Keyboard (Darragh and Witten, 1992). The former uses a four-gram letter model, i.e. the current letter is predicted given the preceding three letters. The probabilities are obtained from an *a priori* analysis of likely letter sequences of some dictionary.

In the latter method, the prediction mechanism of the Reactive Keyboard actually works on a variable-length context matching technique that fetches the predictions from a "long-term" memory that is primed with large texts at the beginning of the session. The variability of the match length (k-tuples) reaches from predicting single letters over words even to short phrases. If a higher-order model fails to match the current context, the system falls back to lower-order models. The whole process is *adaptive*, i.e. while the user is entering new text, rarely used tuples are discarded in order to make way for new ones that are created from the current context. The frequency count of recurring tuples is increased until a limit is reached. This limit depends on the size of the variable that is used for storing the frequency counts. It is argued that smaller sizes of 6–8 bits, i.e. counts up to 256, perform much better than standard 32-bit variables. All frequency counts are halved as a result of the "overflow" and tuples that end in a zero frequency are marked for deletion to allow the storing of new models. This behavior can be interpreted as a simple "forgetting" algorithm which is necessary if the system has to be able to "learn" new predictions from the currently entered text. The higher the number of bits of the counting variables, the longer it takes the system to forget low-frequent entries and adapt to new user input. The variable-length models are stored in a tree structure that has the property of finding partial matches between the context and the model very efficiently and being quite economic in memory usage since lower-order models are contained in the higher-order ones. The Reactive Keyboard core is integrated in an editor and is available for many machine platforms. For further details, see (Darragh and Witten, 1992).

3.2.2 Word-wise approaches

The next level of prediction is the word-wise approach. Instead of suggesting likely characters, the system offers possible completions to the entered key sequence or even predicts additional words after a word has been typed and selected. Usually, these predictors utilize a statistical language model that implies the probabilities of words given the history of already typed words. These likelihoods are determined, i.e. *trained*, on large corpora that are considered to reflect a language as close as possible. The more comprehensive the training corpus, the more accurate the language model (cf. rule of large numbers, e.g. (Feller, 1968)). Obviously, statistical language models can never be perfectly "balanced". There will always be situations where unlikely or no suggestions are presented to the user because this kind of context never occurred before. This is also known as the *sparse data problem* (see e.g. in (Charniak, 1993; Manning and Schütze, 2000)). Additionally, there are situations where certain word suggestions depend on a syntactic, semantic or even pragmatic analysis of the context. This kind of knowledge cannot be "coded" into a simple statistical language model that only relies on counts of word n-gram occurrences. Interestingly, this simple approach often yields very good results and it is a challenging task to improve the performance of a system significantly by utilizing more complex methods. In (Fazly and Hirst, 2003) e.g., using additional part-of-speech information (cf. Section 4.1) only yields a very small improvement on the keystroke saving rate but is much slower when compared to the use of plain word n-grams.

Some recent frameworks for word completion are described in (Copestake, 1997; Matiasek et al., 2002) (based on word n-grams with additional part-of-speech information) and (Tanaka-Ishii et al., 2002) (based on an adaptive language model utilizing PPM (prediction by partial match) and using a reduced keyboard, cf. Section 3.3). The latter approach actually originates from the information theory domain and deals with the problem of improving the compression rates of arithmetic coding. PPM (Cleary and Witten, 1984; Witten et al., 1987) lowers the *entropy* of a language model by maintaining a list of already seen contexts and its corresponding successors. For a character-based approach, the possible contexts of the string *fork*, e.g., are the empty context, *f*, *o*, *r*, *k*, *fo*, *or*, *rk*, *for*, *ork* and *fork*. For each of these contexts, a separate list of characters that appeared after the context is stored as well as their frequencies. The PPM model can then assign probabilities to subsequent characters, thus resulting in the ability to predict the most likely continuation of a current context. The same approach can be used with words instead of characters. In (Tanaka-Ishii et al., 2002), PPM is used to interpolate n-gram counts from a small user-specific corpus with unigram probabilities from a larger base dictionary. The system is adaptive since selected words are instantly added to the user dictionary. The result is a dynamic language model with increased cognitive load because words often appear at different locations, so the user has to adjust to this fact every time a word is entered repeatedly, possibly slowing down the selection process.

The previously mentioned entropy is usually used to measure the "quality" of a language model. Low entropy values denote a better model. In general, entropy can be used to evaluate systems that are based on n-grams (cf. Section 2.2.3). More formally, if the words we are predicting are expressed as a random variable, entropy stands for the average length (in bits) of an indicator that transmits an outcome of that variable. In the word prediction domain, we are interested in computing the likelihood of a word suggestion given a history of already typed words. If the existence of a sufficiently large corpus for determining a language model is presumed, it is obvious that a prediction scheme based on the previous three or four words will yield better results than just considering a single preceding word by decreasing the size of the suggestion lists. This size of the "search space" is exactly what is expressed by *perplexity*, a notion that is closely related to entropy. If we assume that a suggestion list has 8 entries, e.g., the position of the target word could be coded with a 3-bit indicator, whereas if it has only 2 entries, one bit is sufficient. So entropy and perplexity can be seen as a matter of how surprised we will be with some candidate list for a word that is currently typed. In this context, surprise rises with increasing suggestion list length.[7]

Although most word completion systems rely on statistical language models, an approach that is entirely based on grammatical and semantic rules is the KOMBE project (Guenthner et al., 1993). The prototypes utilize grammars and lexicons that capture conceptual and contextual knowledge for fragments of German and French. The syntactic rules allow for valid word predictions in terms of grammatical correctness, whereas the conceptual knowledge component favors words that constitute a likely continuation of a sentence typed so far. The user can choose one of three modes to determine what components are activated. In *full mode*, where all modules are available, the first word is chosen from a list of syntactically possible sentence beginnings which imposes constraints on the following continuation. The *restricted mode* disables the conceptual model and bases the predictions only on grammatical rules. In *free mode*, the user can enter each word freely without syntactic restrictions. If a word is unknown to the system, it has to be entered letter by letter and classified by the user with the help of a dialog-based lexical acquisition module. This step takes some time but is necessary in order to link syntactic and conceptual features to the word such that it can be used properly by the analysis mechanisms in upcoming sentences. Unfortunately, due to funding problems, the system never reached the stage of user trials.

[7] If our language model is perfect in a sense that it always predicts the correct word, i.e. its entropy is 0, then we would actually be very surprised. This kind of astonishment is *not* meant by the notion of the term "surprise".

3.3 Ambiguous keyboards

The idea to use *ambiguous keyboards*[8] goes back to the early 80s where a telephone keypad layout (a modern representative is depicted in Figure 3.1)[9] is used for data entry (Witten, 1982). As already stated in the introduction, an ambiguous keyboard maps several letters to a single key and thus reduces the number of physical switches or keys needed for typing. Thereby, entered words need to be disambiguated since more than one word can be encoded with that particular key sequence. For example, if we take the English lexicon of the CELEX lexical database (Baayen et al., 1995) and type the key sequence [2] [2] [7] [3] [7] with the keyboard from Figure 3.1, then the result is a list with eleven candidates (sorted by frequency), namely "*cases cards acres bases cares caper capes bards baser bares barer*", that needs to be further disambiguated for the intended word. Obviously, the less keys the reduced keyboard has, the longer the lists with matching words grow. If we consider a highly reduced keyboard containing only three letter keys (cf. the English mapping in Table 3.1 on page 34), typing the word *cases*, i.e. code [3] [2] [1] [3] [1], already gives 28 candidates ("*makes cases cabin cakes lakes caves haven gases ...*"). Note that since the letter mappings changed in these two examples, the words from the candidate list of the 8-letter keyboard are not necessarily contained in the list of the 3-letter keyboard.

Early methods involved direct disambiguation by tipping the key with the intended letter multiple times, e.g. hitting the [2] button (in Fig. 3.1) three times for the letter "c". Obviously, determining a useful timing interval that defines when the letter is finally accepted is quite problematic and user-specific. Alternative approaches need one additional separator key that is used to manually terminate the current selection. It is possible, e.g., to use the lowest row, i.e. the keys [∗], [␣] and [#], for this kind of disambiguation. The letter combinations [2] [∗], [2] [␣] and [2] [#] then stand for an "a", "b" and "c", respectively.

The multi-tap method is commonly accepted when writing text messages via SMS with todays mobile phones. Interestingly, (Witten, 1982) notes that, assuming a dictionary of 24,500 words, only approx. 8% of the words are actually coded ambiguously, i.e. where an additional disambiguation step is needed in order to select the target word. So the majority of words can be typed by simply pressing the appropriate letter buttons just once, thus increasing input rate significantly. The word *communication*, e.g., is then obtained by tapping the key sequence [2] [6] [6] [6] [8] [6] [4] [2] [2] [8] [4] [6] [6] without further disambiguation. Meanwhile, this interesting feature is used commercially in many modern mobile phones through the

[8] also called *reduced* or *cluster keyboards*

[9] The keypad layout in (Witten, 1982) contains only keys with a maximum of three letters, i.e. the letters "q" and "z" are shifted to button [1].

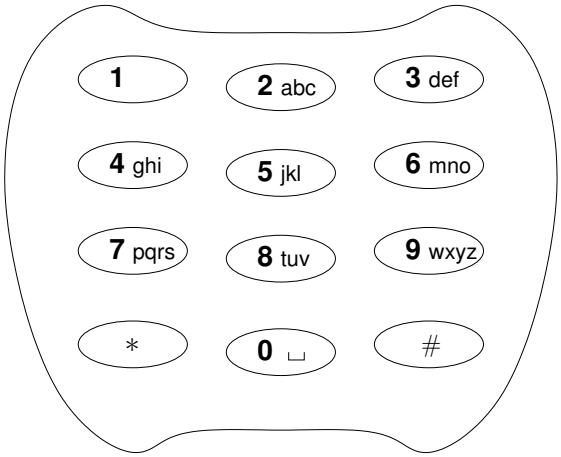

Figure 3.1: The reduced keyboard of a modern cellular phone.

proprietary T9™ system which was developed by Tegic Communications (Kushler, 1998). It basically distributes the letters of an alphabet on seven or eight keys and uses an additional unambiguous key serving as the separator between words ("space"). On mobile phones, the letters are usually distributed in alphabetical order, but the number of ambiguously encoded words can be reduced significantly if the reduced keyboard is optimized for a specific dictionary. We will come back to this in Section 3.4 where a highly reduced and optimized keyboard with only three letter keys is presented.

3.3.1 Sentence-wise text entry

An approach that goes beyond a word-wise step-by-step disambiguation is reported in (Rau and Skiena, 1996) which is the only reference we found to utilize a similar approach as undertaken in this study. Instead of permanently changing between two modes, i.e. a phase where a word is typed and a phase where it is disambiguated in a list of current suggestions, the user can solely concentrate on the process of text entry in a sentence-wise approach. In (Rau and Skiena, 1996), the words are typed one after another with a telephone keypad (cf. Figure 3.1) and are delimited by an overloaded space key. Similar to (Witten, 1982), the letters "q" and "z" are placed on the $\boxed{*}$ key, which also holds the space delimiter "␣". Only the end of a sentence is marked unambiguously by pressing the $\boxed{\#}$ key. Four phases are used to determine a disambiguated reading of the entered key sequence:

1. Recognizing blanks:
   Because of overloading the $\boxed{*}$ key with "␣", "q" and "z", a window that captures

the three preceding and following keys is used to determine whether a space is involved or not. The blank confidence is based on counts of how often a blank occurred given the preceding and following key trigrams. The blank recognition accuracy is stated to be 81.63% without making mistakes, i.e. falsely assigning the blank to that position although it is the letter "q" or "z". Higher blank recognition rates are achieved by lowering the confidence threshold, but this also results in more recognition errors.

2. Constructing word candidate sets:
   The tokens that are obtained after the blank recognition are used to build lists of matching candidates from a large dictionary containing approx. 50,000 words. Most of the words map to one unique key sequence but 16% are coded ambiguously.

3. Handling unknown words:
   If the token has no corresponding dictionary entry, an interpretation is constructed using various features such as partial dictionary matches, affix and suffix analysis and transitional letter probabilities.

4. Sentence disambiguation:
   A dynamic programming technique, namely the Viterbi algorithm (cf. Section 2.1.2), is used to find the most likely sentence that matches the current tokens. The features that guide the search are obtained from examining word pairs in the Brown Corpus. Additional grammatical constraints are used to tag the words with part-of-speech information (cf. Section 4.1) and thus help eliminating unlikely or implausible candidates.

The results obtained by simulating the typing of various text samples with this framework look very good. For various domains, the percentage of correct words reaches from 92.86 to 97.67%. This is due to the relatively high number of keys and low number of ambiguous words, respectively. The overall procedures seem to have a distinctive potential that is sufficient to correctly resolve a large amount of ambiguous words. An additional user-acceptance test was performed with 13 test persons that had to type three sentences of increasing length with the ten-key telephone pad. One of the main conclusions is that typing comfort decreases with increasing length of the sentences. This result is reasonable since a long sequence of ambiguous codes might confuse the user such that the current position in the intended sentence, i.e. the message that is going to be expressed, might become blurry after some time of typing, resulting in higher error rates (e.g. by word repetition or omission) or even breaking off and restarting the text entry process.

Nevertheless, a sentence-wise ambiguous text entry system is also focused in this study. The main difference to the system in (Rau and Skiena, 1996) is that the

ambiguous keyboard used in our approach is highly reduced, containing only three letter keys and one delimiter key that unambiguously indicates the end of a word. A consequence of this is that the amount of ambiguous words, i.e. words that do not have a unique coding, is much higher than with a keyboard using more letters.

3.3.2 Unknown words

As with all dictionary-based text entry methods, one of the biggest problems is concerned with words not being in the dictionary. This is also known as out-of-vocabulary rate (OOV). It is obvious that for any text entry method that relies on a lexical database, the user will sooner or later get to know its limits. If an unknown word is not within the candidate list, it has to be disambiguated letter by letter. The obtained word is then stored in a special user dictionary. The question of how to merge the words from the user dictionary into the suggestion lists of the general model is discussed in (Harbusch et al., 2003). A simple linear interpolation of the two dictionaries has a positive effect on the OOV rate with increasing user dictionary size. This approach can also be used with domain- or topic-specific lexicons that "prime" the overall language model of the system with expressions from a specific field. The corpus used for an evaluation of the sentence-wise text entry framework elaborated in this study can be considered to be domain-specific. The advantage for a speech impaired user lies within additional speed-up. It is thinkable, e.g., to prime the general language model of the base dictionary with domain-specific corpora for scenarios like shopping, different school subjects (history, biology, etc.) and discussing politics or the like. The effect is that words which are likely to occur in these situations are ranked higher than in the usual candidate lists of the base dictionary.

For the unknown words, additional spelling modes have to be supplied by the system such that the user can add them once to a personal dictionary. The easiest possibility is to separately disambiguate letter by letter in alphabetic order. A better approach is to use n-gram letter probabilities to present the user more likely letters first, given the already disambiguated context (cf. (MacKenzie et al., 2001)). Especially in highly inflected languages as German, the amount of unknown words for a communication aid with a limited lexicon is assumed to be much larger than for English. One could consider a method that composes new words from an unknown code by taking existing words from the dictionary that partly match the code sequence and try to "glue" them together to form compounds. In English, most compounds are made up of separate words (e.g. *bed linen*), whereas in German, these words are concatenated (*Bettwäsche*). One possibility is to type the words separately (*Bett* and *Wäsche*) and concatenate them via a special command that has to be implemented in the system and builds a compound from the two last words. Unfortunately, this mode is disadvantageous for the user since she already has to suspect that a word

is not in the dictionary. This kind of planning activity might slow down the overall text entry rate significantly (similar to the cognitive load of scanning the candidate lists for the target word during word prediction). Therefore, it would be practicable if the system determined a possible compound for the code of *Bettwäsche* by finding the codes for *Bett* and *Wäsche* and presented all such suggestions to the user. If no valid compound is generated, a fallback to a letter-wise disambiguation should be possible.

3.4 The UKO-II communication aid

The systems presented so far (i.e. T9™ and (Rau and Skiena, 1996)) use an ambiguous keyboard containing up to 10 keys with only three or four letters per key. The advantage of such keyboards is the relatively low number of actual ambiguous words, i.e. the biggest part of a dictionary is coded uniquely. Since users with impairment are often capable of only operating a small number of physical switches, the use of such keyboards is problematic. If all keys cannot be accessed directly then scanning techniques have to be used (Lesher et al., 1998). Undoubtedly, text entry via scanning is much slower and probably more frustrating than being able to directly activate the keys. Therefore, a further reduction in the number of keys is appropriate. The disadvantage of this step is a rise in ambiguity. Since the alphabet is now distributed on even fewer keys, more words share the same coding, resulting in higher selection costs. Assuming a reduced keyboard with three letter keys and the CELEX database (cf. also Section 3.4.1), the candidate lists can grow as long as 50 entries for English (key sequence [3] [1] [1] [1], cf. Tab. 3.1: *look loss moon guns ...*) and 75 entries for German ([1] [2] [2] [2] [3]: *mußte Woche müsse wußte ...*), see also (Harbusch et al., 2003).

The final realization of an assisting text entry system is very user-dependent. In our approach, a communication aid using a highly reduced keyboard serves as a starting point. The UKO-II system (Harbusch and Kühn, 2003b) is a configurable open architecture that allows for variable keyboard layouts and utilizes a layered language model. It is implemented in Emacs Lisp[10] and currently runs within the XEmacs text editor[11] which has the advantage of being an extensible and fully customizable display editor and Lisp interpreter. With additional modules (written in Emacs Lisp, short Elisp), it provides nearly everything for everyday communicative needs, such as e.g. an e-mail and newsgroup client, instant messaging and much more. If some feature is missing, it can be easily implemented since many modules are open source or free software.

[10] `http://www.gnu.org/manual/emacs-lisp-intro/`
[11] `http://www.xemacs.org/`

The predecessor of the UKO-II system was introduced in (Garbe, 2001) and distributed the letters on four keys with two additional command keys. Kathrin, the pilot user suffering from cerebral palsy, had to utilize a circular scanning method to operate the program with one physical key (one single switch in the headrest of her wheelchair). The prototype was then further improved by Jörn Garbe and the next version (called ERIC) enabled Kathrin to directly access three letter keys (with two additional switches in the headrest) and one command key (with a special switch mounted to her knee) (cf. (Garbe et al., 2001)). After a short time, she accustomed herself to the new setting and could now determine the speed of text entry on her own since she did not have to wait for the highlighting of the desired item set any more. One disadvantage of this system was the fixed command hierarchy. The command mode (which is entered by activating the command switch and allows for actions such as *delete last key*, *accept current match* or *speak last sentence*) is "hard-coded" and cannot be easily extended with own macros or additional commands. With the new approach taken in the UKO-II framework, the user will be able to directly implement the desired commands in Elisp in future versions. Since the language is interpreted, the new functionality is instantly available and provides a flexible way of extending the basic command set.

3.4.1 Keyboard layout

The current prototype uses the minimum of three keys for the letters and one additional key for entering a command mode that enables the user to issue commands to the system, e.g. for deletion of the last keypress or denoting the end of a word. Going below this value, i.e. distributing the letters on only two keys, would result in unfeasible selection costs. Strongly impaired users that can only operate 1–3 switches have to rely on a scanning mode for the four-button keyboard (cf. (Harbusch and Kühn, 2003a)). The distribution of the letters on the keys is optimized for a specific lexicon. For German and English, the dictionaries are based on the CELEX lexical database (Baayen et al., 1995). A genetic algorithm (see e.g. (Goldberg, 1989)) optimizes the layout by minimizing the length of the candidate lists and the number of scanning steps if appropriate (Garbe, 2001; Kühn and Garbe, 2001). The current keyboard layouts are depicted in Table 3.1.

Since the distribution seems to be arbitrary, one could consider that an alphabetic arrangement would result in more readability, but the gain in reduction of the suggestion list length is significantly higher with an optimized layout (Garbe, 2001). The effort for "learning" the keyboard layout is negligible. After a short training period, the user should be able to operate the keyboard blindly. The German layout of the keyboard is used for simulating the evaluation corpus that will be introduced in Section 6.1. A direct selection of the keys is assumed with one additional key serving as an unambiguous "space" that separates the words of a sentence. The evaluation

German

a g j l m q r w z ä	c f h k o s t u v x y ü ß	b d e i n p ö -
1	2	3

English

b j k n o s v w x u	a d f p q r t - '	c e g h i l m y z
1	2	3

Table 3.1: The keyboard layouts of the UKO-II prototype for German and English. The additional command button is omitted.

results for the baseline that utilizes a simple frequency-based language model (which is described in the following section) are presented in Section 6.2.

3.4.2 Frequency-based static language model

As mentioned above, the underlying lexicon is derived from the CELEX database (cf. also Section 6.1.2) which provides frequencies of the words based on large corpora. So far, the candidate lists of the prototype are vertically ordered according to this very simple language model, i.e. the matching words are listed with decreasing frequency, the most frequent word appears at the first place. Since Kathrin attends a regular school and uses the ERIC communication aid in everyday life,[12] her experiences can be seen as a direct evaluation of the usability. The most interesting thing is that the word completion was rarely used. The early version provided a prefix-based completion of words that are currently typed. It turned out that the time needed to glance at the possible word completions as typing advances and scan them for the target word lessens the benefit of possible reduced motor requirements. The prevailing mode of the pilot user was to simply enter the whole word in one go and concentrate on the candidate list after that. In general, most of the evaluations concerning word completion software take the *keystroke saving rate* (ksr) as one of the main performance criteria (see e.g. (Higginbotham, 1992)). Keystroke savings are measured by counting the number of keypresses needed for accomplishing a task (e.g. writing a short text) with and without some features like word prediction enabled. It may sound good if a system has a keystroke saving rate of 50%, but the statement has no expressiveness about the actual time needed to enter the text.

It is elaborately shown in (Horstmann Koester and Levine, 1994)[13] that word prediction can slow down user performance significantly because of cognitive and perceptual activities. These activities have to be considered when evaluating systems that offer word prediction. The evaluation is based on trials with 14 subjects, of which eight were able-bodied and six had spinal cord injuries. The able-bodied persons had to use mouthstick typing to access the keyboard, whereas the ones

[12] The UKO-II system is still an early prototype.
[13] cf. also (Horstmann Koester and Levine, 1996)

with spinal cord injuries relied on their usual input methods (mouthstick or hand splint typing). After a training phase where the users could accommodate to the two available interfaces, namely a "Letters-only" mode (letter-by-letter spelling) and a "Letters+WP" mode (single letter entry with additional six-word prediction list), the testing phase comprised seven sessions of typing 14 sentences in both modes. All system data was logged in real-time, e.g. the selected items with the corresponding time at which they were selected. The evaluation of the material supports the hypothesis that the increased cognitive load of searching the candidates in the suggestion list has a negative effect on the overall performance. For the impaired users, the text generation rate was always significantly higher in the "letters-only mode" (approx. 100–125 characters per minute). When compared to the word prediction mode (only approx. 60–80 chars/min), this results in a decrease of the rate by 40% when word prediction is used. For further details, see (Horstmann Koester and Levine, 1994). Although unambiguous keyboards were used for this evaluation, the results also hold for ambiguous ones since the word predictions do not depend on the type of keyboard. There are much more candidates when using ambiguous keyboards but the suggestion lists are usually of limited length (5–10 entries). Still, if the user has a severe disability, the reduction of motor requirements (which is directly correlated with a high keystroke saving rate) might be one of the primary goals.

3.5 Summary

So far, the field of AAC was introduced and an overview on ambiguous keyboards and existing word prediction approaches was given. In our case, the pilot user of the UKO-II communication device is able to operate the three switches with her head quite fast, and thus, a sentence-wise text entry mode is considered for further examination. It is important to be able to show communicative initiative in conversations, which means that the intended message has to be expressed as fast as possible. As was already argued, glancing over possible candidates generated by some word prediction module while typing progresses is not always feasible. Therefore, a sentence-wise approach where the user concentrates on the message and makes adjustments to the final candidate lists afterwards is considered in this study. So the primary task is to examine possible means that result in minimal final selection costs of the target words. In the following chapter, techniques are presented that, in addition to the statistical language modeling given in Chapter 2, allow for a *partial parse* of sentences[14] (cf. (Abney, 1991)). The basic idea is to determine a more detailed syntactic analysis of a sentence. Simple chunking techniques try to find only the noun phrases of a sentence, e.g. (Church, 1988), whereas more complex ones provide as much syntactic description as possible (Hindle, 1994).

[14] also called *shallow parsing* or *chunking*

Chapter 4 introduces part-of-speech tagging and the related supertagging which provides the necessary concept towards a shallow parsing technique. In the context of this study, those sentence hypotheses that have a maximum syntactic consistency are determined with the help of a *lightweight dependency analysis* (Srinivas, 1997a) and used to reorder the suggestion lists such that more likely candidates appear at the top, thus reducing overall selection costs. Since the level of ambiguity with the four-button ambiguous keyboard is very high, i.e. the candidate lists usually have more than one entry, we also need a way of finding more than one good hypothesis. This is generally achieved by using search techniques which are also discussed in the next chapter. The final system is presented in Chapter 5.

4 Partial Parsing and Search Techniques

This chapter describes the theoretical background needed for improving the results of the ambiguous keyboard with the use of an n-best supertagger. *Supertagging* (Srinivas and Joshi, 1999) uses the Lexicalized Tree Adjoining Grammar formalism (LTAG) and is comparable to part-of-speech tagging. Instead of parts of speech being annotated to the words of a sentence, supertagging uses so-called supertags that represent an elementary tree structure with some lexical anchor. The advantage of this is the ability to find dependencies between these supertags on a phrase or even sentence level, thus resulting in a partial parse of the sentence by likely identifying its chunks. This information is obtained by a so-called *lightweight dependency analysis* (Srinivas, 2000). With the methods presented in Chapter 2, a supertagger can find the most likely supertag sequence for a sentence via the Viterbi algorithm. Since we deal with ambiguous codings that each expand to a list of matching words, a single best hypothesis probably will not be enough to improve the system significantly. This claim motivates the employment of n-best techniques that are able to find more than one good hypothesis.

Section 4.1 introduces part-of-speech (POS) tagging and the basic methods how to estimate probabilities of POS-tag sequences given a sentence on the basis of trigrams. Section 4.2 gives an overview on the TAG formalism and Section 4.3 describes the related supertagging framework. In particular, the supertagger by (Bäcker, 2002) is presented in detail since it is used as a basis for the n-best supertagger from Chapter 5. Section 4.4 sketches alternative shallow parsing techniques. Various search techniques from the field of Artificial Intelligence and n-best approaches from the speech recognition area are described in Section 4.5.

4.1 Part-of-Speech Tagging

One main goal of *Natural Language Processing* (NLP) is to find ways for representing, parsing and therefore understanding the structures that underlie natural languages (see e.g. (Allen, 1995; Jurafsky and Martin, 2000)). One possible approach is to build grammars that reflect regularities of a specific language. Usually, coming up with an elaborate grammar takes a lot of time and is a rather complicated task. Often, one is interested in simpler models because they are easier to understand, implement and reconstruct. One such model is *part-of-speech tagging*, or simply *tagging*. Parts

of speech (POS) are the labels for words that are used for differentiating their morphosyntactic function within the sentence, e.g. partitioning different words into their classes like nouns, verbs or adjectives. Now, tagging is the process of labeling each word in a sentence with an appropriate part of speech tag. Interestingly, annotating each word with the likeliest POS already results in an accuracy of approx. 90% because more than half of the words are not ambiguous in most corpora (Allen, 1995). Since it is impossible to always know the correct tag for all words in any sentence (which would require an infinitely large training corpus), we can try to find the most likely interpretation within a fixed scope using the techniques presented in Chapter 2 by using n-grams (cf. Section 4.1.2). A comprehensive introduction on the field of syntactic word-class tagging is given in (van Halteren, 1999).

4.1.1 History of taggers

The first taggers appeared in the late 1950s and were mostly based on linguistic approaches, i.e. hand-written disambiguation rules. Interestingly, (Joshi and Hopely, 1996) report one of the pioneering taggers that were implemented within a parser by cascading finite-state automata. The basic idea behind taggers was to reduce the number of possible hypotheses a parser has to check in order to find a valid syntactic parse tree of a sentence. With this kind of filter for the candidates of each word, the speed of the parser can be improved significantly. One possible drawback emerges if the tagger also discards candidates that actually belong to the solution. If the parser solely relies on the output of the tagger in that case, it cannot find the correct parse any more. Early taggers that used linguistic rules rarely assigned more than 80% of the words a correct tag. The TAGGIT tagger (Greene and Rubin, 1971) used about 3,300 rules of the form $W\ X\ ?\ Y\ Z \rightarrow A$ (called *context frame rules*), where "?" symbolizes an ambiguous word and $W\ X$ and $Y\ Z$ represent the tags of the words to the left and right context, respectively. In general, A specifies the tag that is assigned to the ambiguous word in the middle if the context matches and A is in the lexicon of possible tags for that word. The tag A could also appear negated on the right-hand side, as e.g. in NNS $?\ \rightarrow$ *not* VBZ (which simply states that a plural noun is never followed by a verb in third person singular form). TAGGIT used the Brown tag set containing 77 tags and was designed to help annotate the Brown Corpus (Francis and Kučera, 1964), one of the first extensive text collections comprising 15 genres of written American English with a total size of over one million word tokens (500 text samples with about 2,000 words each). It is reported in (Leech, 1997) that it reached an accuracy of 77% on the Brown Corpus.

A giant leap in word tagging was made with the rise of data-driven methods in the late 1970s. By "borrowing" the advances made in speech recognition with HMMs, the taggers could be used on large corpora and automatically improve their accuracy by expectation maximization (cf. Section 2.1.2). After these n-gram HMM taggers

reached the limit of what seemed to be feasible (an accuracy of approx. 97%), new approaches emerged in the early 90s and rule-based taggers became popular again. The big difference compared to early rule-based taggers was that they were now data-driven, i.e. they learned the rule sets automatically on a large corpus. Especially Brill's tagger (Brill, 1992) aroused the interest of many researchers. Modern POS taggers are mostly hybrid system, using the "best of both worlds".

4.1.2 Probabilistic data-driven tagging

Although good results can be achieved with hand-written disambiguation rules, the work put into the development of consistent rules that also stay maintainable is rather time-consuming. The breakthrough of statistically driven methods came with the development of annotated corpora. Actually, the first data-driven tagger, a system called CLAWS (Constituent-Likelihood Automatic Word Tagging System) (Garside, 1987), was used to help annotate the Lancaster-Oslo/Bergen (LOB) Corpus (Johansson, 1986). The first version of CLAWS was based on bigram Markov models and achieved an accuracy of 96–97%. This accuracy turned out to be the upper bound of what is feasible with the data-driven approach. Generally, improving the accuracy beyond 97% is a very difficult problem. In fact, (Marcus et al., 1993) note that even human annotators only agreed on 96–97% of the tags assigned for the Penn Treebank sentences of the Brown Corpus.

As mentioned above, data-driven taggers annotate the words of a sentence with a POS tag sequence "learned" from a large training corpus by maximizing the probability of each word given a tag and given preceding tags of the previous context. The most common approach is to use Hidden Markov Models in order to determine the most likely tags for the words. The output symbols of the HMM are the words w_i of the sentence, whereas the states correspond to the different tags t_i. One big advantage of HMMs is that they can be automatically trained with the Baum-Welch algorithm. Let $T = t_1 t_2 \cdots t_N$ be a sequence of part-of-speech tags for the sentence $W = w_1 w_2 \cdots w_N$. The most probable tag sequence $\widehat{T}$ is defined by

$$\widehat{T} = \text{argmax}_T \, P(T|W). \tag{4.1}$$

According to Bayes' law,

$$P(T|W) = \frac{P(T)P(W|T)}{P(W)}, \tag{4.2}$$

whereof the denominator $P(W)$ can be left out since it is the same for all tag sequences and the task is to maximize the expression. Thus, Equation 4.1 can be restated as

$$\widehat{T} = \text{argmax}_T \, P(T)P(W|T). \tag{4.3}$$

The probability of the tag sequence, $P(T) = P(t_1 t_2 \cdots t_N)$, can be expressed through the chain rule (cf. Equation 2.4) as $P(t_1)P(t_2|t_1)P(t_3|t_1 t_2) \cdots P(t_N|t_1 t_2 \cdots t_{N-1})$. If we consider only a fixed word frame (cf. the limited horizon Markov assumption) and assuming pseudo-tags at the beginning of the sentence, the probability can be approximated by

$$P(T) \approx \prod_{i=1}^{N} P(t_i|t_{i-n+1} \cdots t_{i-2} t_{i-1}), \tag{4.4}$$

where n is the size of the window, i.e. the value for the n-gram being used. In order to compute the probability $P(W|T) = P(w_1 w_2 \cdots w_N|t_1 t_2 \cdots t_N)$ efficiently, the additional assumption is made that all words w_i are independent of each other. This results in another approximation, namely

$$P(W|T) \approx \prod_{i=1}^{N} P(w_i|t_i). \tag{4.5}$$

The most favorite model in terms of both computational efficiency and achieved accuracy used in POS tagging has turned out to be based on trigrams (see e.g. (Brants, 2000; Thede and Harper, 1999)). A solution to the task of finding the best tag sequence $\widehat{T}$ in Equation 4.3 using a trigram model can finally be found through the following equation:[1]

$$\widehat{T} = \text{argmax}_T \prod_{i=1}^{N} P(t_i|t_{i-2} t_{i-1}) P(w_i|t_i). \tag{4.6}$$

The term $P(t_i|t_{i-2} t_{i-1})$ is also known as the *contextual probability*, whereas $P(w_i|t_i)$ is called the *word emit probability*. As presented in Section 2.2, one can use relative frequencies and MLE to determine these probabilities,

$$P(t_i|t_{i-2} t_{i-1}) = \frac{C(t_{i-2} t_{i-1} t_i)}{C(t_{i-2} t_{i-1})} \quad \text{and} \quad P(w_i|t_i) = \frac{C(w_i, t_i)}{C(t_i)}, \tag{4.7}$$

and apply smoothing or back-off techniques in order to prevent the zero probability problem. The values returned by $C(t_{i-2} t_{i-1} t_i)$, $C(t_{i-2} t_{i-1})$ and $C(t_i)$ denote the counts of the POS trigrams, bigrams and unigrams, respectively, whereas $C(w_i, t_i)$ returns the frequency of word w_i that is tagged with the part-of-speech t_i. For the supertagger used in this thesis, a detailed overview on these techniques was given in Sections 2.2.1 and 2.2.2. Additional properties are discussed in Section 4.3.2.

[1] Again, pseudo-tags are used at the beginning of the sentence (for t_i with $i < 1$).

4.1.3 Rule-based data-driven tagging

The above paragraphs focused on the stochastic approach to part-of-speech taggers. When it was realized that probabilistic HMM taggers reached the boundary of what is possible, rule-based approaches gained popularity once more. A tagger that was particularly attracting the interest of researchers was that of (Brill, 1992). It also works on a training corpus but instead of deriving the probability distribution of n-grams via stochastic models, it automatically acquires rules that specify how to change tags of already tagged words based on the current context. For this purpose, the tagger firstly assigns to each word the most likely tag which is estimated from a large annotated corpus by using unigrams, i.e. no contextual information. After this initial run, the tagger acquires *patches* that specify simple tag change rules as follows: it applies the initial tagger to a separate patch corpus and gathers tagging error triples (t_w, t_c, n) which indicate that n words are wrongly tagged as t_w instead of the correct t_c. These triples are used to determine the best template from a prespecified set of rule templates of the form "change a tag from X to Y if some condition C is true" that reduces the error rate most, i.e. where n turns out to be smallest. The rule that yields the best improvement is applied permanently to the patch corpus and the patch acquisition process continues. The interesting thing to note is that despite most rules being very simple, the overall relation of the rules is quite complex because the i^{th} rule depends on the corpus state after having applied all $i-1$ patches. Brill trained his tagger with 90% of the tagged Brown Corpus and generated patches on 5% of the remaining data. The rest (5%) was used for testing. Some of the first patches this procedure finds are e.g. "TO → IN, if next tag is AT", "NN → VB, if previous tag is TO" or "TO → IN, if next word is capitalized".[2] The error rate of the initial tagger based on the unigram tags of the training set is 7.9% and drops down to 5.1% after applying 71 patches.

Later improvements (Brill, 1994; Brill, 1995) of the transformation-based error-driven approach are reported to even outperform taggers based on HMMs. One of the advantages of the rule-based tagger when compared to the probabilistic approach is the size of the trained model in terms of memory consumption. A few hundred rules are far easier to store and handle than the large probability matrices of the HMM $\lambda = (A, B, \Pi)$. The space complexity of A, B and Π is in $O(N^2)$, $O(NM)$ and $O(N)$, respectively. Here, N is the number of possible tags (the number of states) and M is the size of the vocabulary (the number of output symbols, cf. Sections 2.1.1 and 2.1.2). If we consider the use of a trigram HMM tagger, the possible states are coded by pairs of tags, so N has to be squared. This results in an overall space complexity of $O(N^4 + N^2M)$. Modern programming languages provide the data type `double` which can hold a 64-bit precision floating point number (e.g. (Arnold and Gosling, 1997)). If we assume a tagset containing 80 tags (the Brown Corpus

[2] TO = infinitive marker *to*, IN = preposition, AT = article, NN = singular noun, VB = verb

distinguishes 87 tags) and a vocabulary of 20,000 words, the storage needed for the HMM matrices is approx. 1,289 MBytes. Fortunately, many tag pairs do not occur in natural languages, so the memory requirements are not as huge as presented here. It is shown in Section 4.3.2 how to reduce the storing space for the matrices by using an associative memory technique, so-called hash tables.

Modern *hybrid* taggers try to combine the best of both probabilistic and rule-based approaches (Garside and Smith, 1997; Tapanainen and Voutilainen, 1994). The reached accuracy is between 96 to 98 per cent. There also exist new approaches to the task. In (Nietzio, 2002), *Support Vector Machines* (SVMs) are used for tagging. The advantage lies in the ability to constitute more complex linguistic objects than is the case for rule-based or statistical systems. SVMs fall into the class of kernel-based algorithms (Cristianini and Shawe-Taylor, 2000) and are used to represent word-tag-sequences as arbitrary feature vectors. The evaluation on a small training corpus consisting of 500–2,000 sentences resulted in an accuracy of 96.1–98.4% for known words, whereas if unknown words are encountered, values of only 91.9–93.9% are reached.

4.2 Tree Adjoining Grammars

When speaking of NLP in general, one is basically interested in parsing and understanding language. In order to achieve this task, we have to come up with rules that reflect the syntactic regularities of a specific language such that, together with a lexicon where the valid words are stored, these rules make up a grammar for this language. By now, research in Computational Linguistics and Linguistics in general has produced many different grammar formalisms, each with certain strengths and weaknesses (e.g. GPSG (Gazdar et al., 1985), HPSG (Pollard and Sag, 1994), LFG (Kaplan and Bresnan, 1982)). At this point, there still is no formalism that spans the complex rules with all their exceptions perfectly well. And as long as mankind is not even sure how it actually is able to understand and speak natural languages so well, any grammar formalism will fail on sometimes the simplest sentence constructions sooner or later.

Basically, the distinction between two basic grammar paradigms, namely string-rewriting grammar formalisms and tree-rewriting formalisms, can be made. The former is represented by e.g. Definite-Clause Grammars (DCGs, see e.g. in (Covington, 1994)) where context-free phrase structure rules as in $S \rightarrow NP\ VP$ or $NP \rightarrow John$ describe how to derive non-terminal or terminal symbols from previous non-terminals (thus substituting, i.e. rewriting, strings with other strings). Tree Adjoining Grammars (Joshi and Schabes, 1997) fall into the latter class. Here, one deals with primitive elements that are called *elementary trees* which can be divided into

- *initial trees* that define basic phrase structure trees of simple sentences (without recursive behavior) and

- *auxiliary trees* representing recursive structures.

The nodes of elementary trees can be characterized as follows:

- interior nodes are labeled by non-terminal symbols,
- frontier nodes are labeled by terminal or non-terminal symbols,
- non-terminal symbols on the frontier of initial trees are marked for substitution,[3]
- non-terminal symbols on the frontier of auxiliary trees are marked for substitution except for one *foot node*[4] which has the same label as the root node.

The elementary trees are combined by two operations, namely *substitution* and *adjunction*, resulting in a *derived tree*. In Figure 4.1, the node Z on the frontier of the tree γ_1 is marked with a down arrow and thus can be substituted by an initial tree whose root has the same label (α_1). Any adjunction on a node marked for substitution is strictly forbidden. When adjunction is applied (cf. Figure 4.2), the root node of the adjoining auxiliary tree (here Y in β_1) replaces the non-substitution node of the sub-tree[5] which is detached and moved to the corresponding foot node of the adjoining tree. The derived trees being generated by these operations are shown on the right of the figures, respectively.

A summary on the formal properties of TAGs and the language class they belong to (i.e. tree-adjoining languages, TAL) are given in (Joshi and Schabes, 1997). These properties also hold for lexicalized TAGs which are presented in the following section.

4.2.1 Lexicalized Tree Adjoining Grammars (LTAG)

So far, we have introduced the basic TAG formalism with the two elementary tree types and the two composition operations that are used to generate more complex tree structures by substitution and adjunction. Since the elementary trees are structured objects and therefore have the advantage to relate to what is called the strong generative capacity,[6] TAGs are more relevant to linguistic descriptions than CFGs. (Schabes et al., 1988) show how to enhance context-free grammars by the process of *lexicalization* and the operation of adjoining in order to extend their domain of locality and generally simplify the task of a parser. The result is formally equivalent to lexicalized TAGs (Schabes et al., 1988).

Lexicalization allows us to associate each elementary tree with a lexical item called the *anchor*. In LTAGs, every elementary tree has such a lexical anchor. It is also

[3] annotated with a down arrow ($\downarrow$)
[4] annotated with an asterisk ($*$)
[5] The overall target (γ_2 in Figure 4.2) can be any tree: initial, auxiliary or derived.
[6] in contrast to weak generative capacity as encountered with sets of strings, e.g. in CFGs

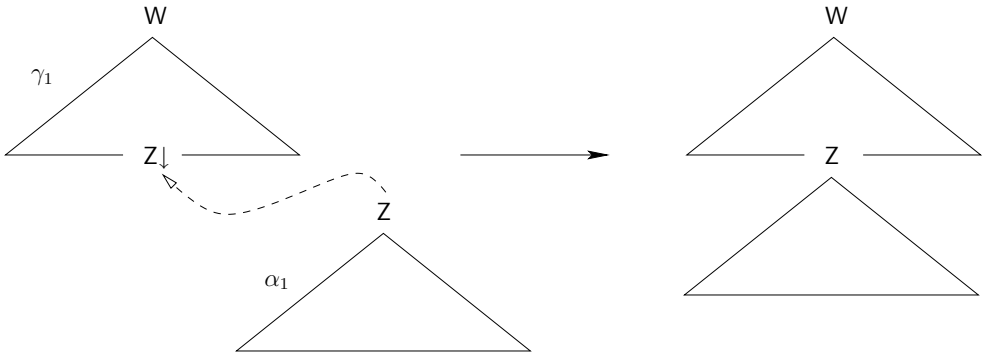

Figure 4.1: The *substitution* operation: the initial tree α_1 is inserted into a substitution node of the elementary tree γ_1.

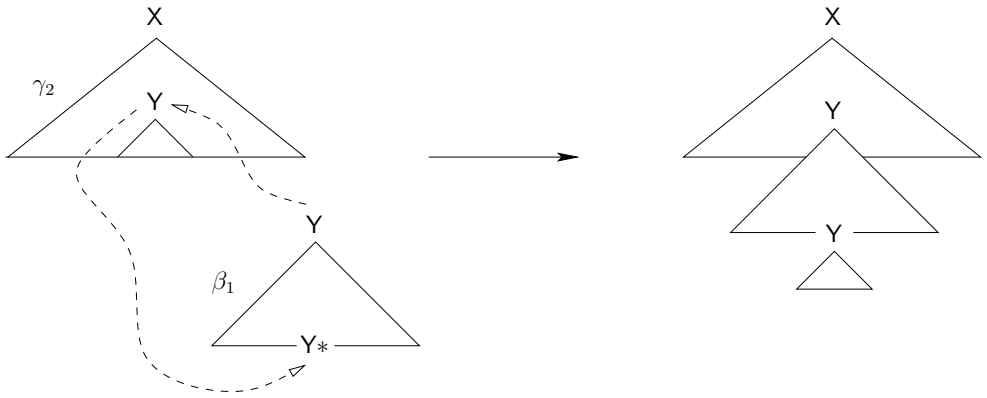

Figure 4.2: The *adjunction* operation: the auxiliary tree β_1 is adjoined on the dominating node of the sub-tree in γ_2.

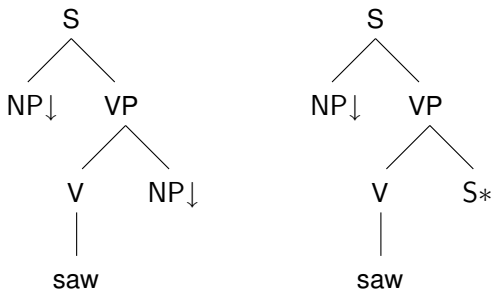

Figure 4.3: Two lexicalized elementary trees for the verb *saw*.

possible that there is more than one elementary structure associated with a lexical item, as e.g. for the case of verbs with different subcategorization frames. The elementary structures in Figure 4.3 allow e.g. for sentences like *Mary saw John* or *John saw that Mary left*.

The important key properties of LTAGs are summarized in (XTAG Research Group, 2001). The extended domain of locality (EDL) and adjoining as described above directly allow for factoring of recursion from the domain of dependencies (FRD). Therefore, all dependencies in LTAGs are local and are specified by the set of initial and auxiliary trees. One part of the EDL property requires that for every environment a lexical item might appear in, the grammar must have an elementary structure for this item. The consequence of this property is that the information of a dependency structure is contained in derivation structures of LTAGs. We will come back to this in Section 4.3.1 where an algorithm is presented that is able to find these dependencies in linear time, thus providing an efficient shallow parse of the sentence. Otherwise, traditional parsing of a lexicalized TAG can be achieved in polynomial time ($O(n^6)$, (Joshi and Schabes, 1997)). The language class tree-adjoining grammars belong to is mildly context-sensitive which places them in between context-free and context-sensitive languages: CFL $\subset$ TAL $\subset$ CSL. Another advantage of LTAGs in general is that they can handle *long distance dependencies* (cf. Section 4.2.2).

A development tool based on the LTAG formalism that comprises a morphological analyzer, a tagger, a parser and graphical interfaces for easy access to a lexicalized tree adjoining grammar for the English language is presented in (Schabes et al., 1993; Doran et al., 1994).

4.2.2 LTAG example

The following section will give a small excerpt of a lexicalized tree adjoining grammar in order to exemplify the properties of LTAGs as described in the previous paragraphs.

The sentence that is examined contains an example for a long distance dependency:

who does Bill think Harry likes (from (XTAG Research Group, 2001)). This is a typical example for the problem class of *wh*-particles that emerge in relative clauses and interrogative sentences. Here, the nominal phrase that comes after *Harry likes* has moved to the front and is substituted by the interrogative *who*. In other grammar paradigms, the resulting *gap* has to be explicitly modeled by features or meta rules (cf. the *slash* feature in GPSG (Gazdar et al., 1985) and *controller/controllee* pairs in LFG (Kaplan and Bresnan, 1982)). In LTAG, the long distance dependencies are localized such that all and only the dependent elements are present within the same structure. Thus, the *wh*-constituent of the nominal phrase complement is present within the same elementary tree. In Figure 4.4, the initial tree α_2 reflects this fact. The process of adjoining and substitution is delineated in Figure 4.5. The initial trees α_3, α_4 and α_5 are merged into the corresponding nodes marked for substitution. The auxiliary tree β_1 is then adjoined to the main tree α_2. Finally, β_2 is adjoined to β_1. The result is the derived tree of the sentence *who does Bill think Harry likes*. We will come back to this example in Section 4.3.1 where the lightweight dependency analyzer, a procedure that directly determines the fillers for the substitution and adjunction nodes from the complement requirements coded in the elementary trees without parsing the whole sentence, is introduced.

4.3 Supertagging

In previous sections, the lexicalized tree adjoining grammar formalism was presented. The elementary structures, i.e. the initial and auxiliary trees, hold all dependent elements within the same structure, thus imposing constraints on the lexical anchors in a local context. Basically, *supertagging* (Joshi and Srinivas, 1994; Srinivas and Joshi, 1999) is very similar to part-of-speech tagging which was introduced in Section 4.1. Instead of POS tags, richer descriptions, namely the elementary structures of LTAGs, are annotated to the words of a sentence. For this purpose, they are called *supertags* in order to distinguish them from ordinary POS tags. The result is an "almost parse" because of the dependencies coded within the supertags. Usually, a lexical item can have many supertags, depending on the various contexts it appears in, and therefore the local ambiguity is larger than with the case of POS tags. An LTAG parser for this scenario can be very slow because of the large number of supertags, i.e. elementary trees, that have to be examined during a parse. Once again, we can apply n-gram models on a supertag basis in order to filter out incompatible descriptions and thus improve the performance of the parser. In (Srinivas and Joshi, 1999), a trigram supertagger with smoothing and back-off is reported that achieves an accuracy of 92.2% when trained on 1,000,000 words. The equation for finding the best supertag sequence for a sentence is the same as presented in Equation 4.6 on page 40, with the only difference that the variables t_i refer to supertags instead of

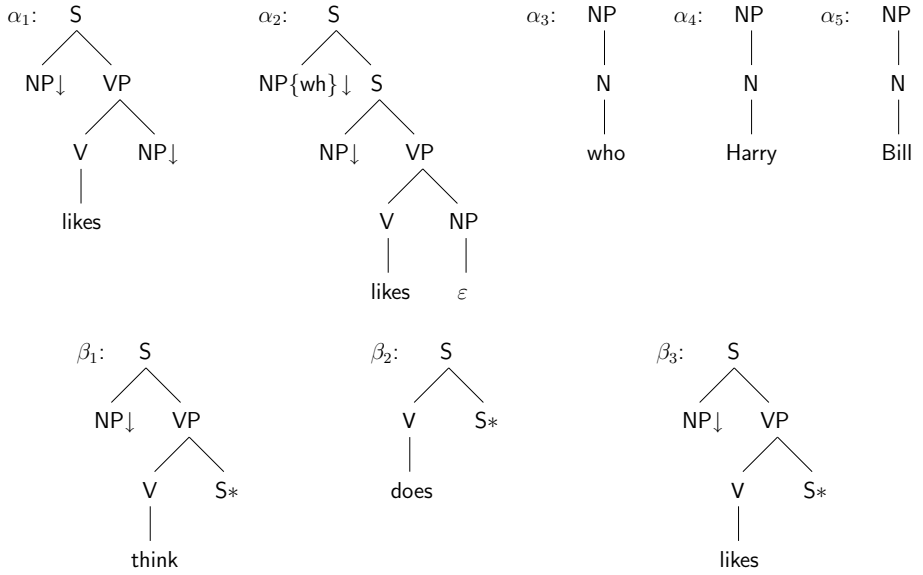

Figure 4.4: The LTAG example for the sentence *who does Bill think Harry likes* (XTAG Research Group, 2001). The missing nominal phrase in α_2 is localized within the same structure.

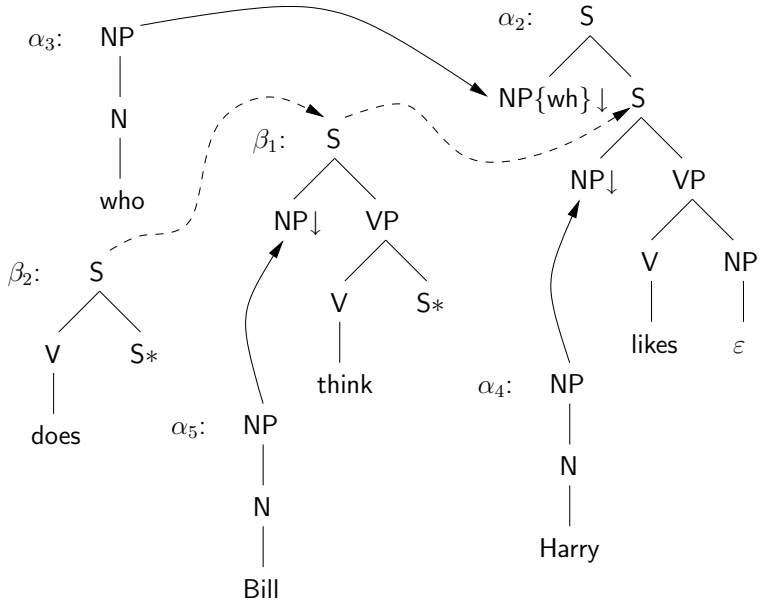

Figure 4.5: The derived structure for the sentence *who does Bill think Harry likes*. Solid arrows stand for substitution, dashed arrows denote the adjunction operation.

It//PRP//A_NXG	single//JJ//B_An	must//MD//B_Vvx
is//VBZ//B_Vvx	man//NN//A_NXN	be//VB//B_Vvx
a//DT//B_Dnx	in//IN//B_nxPnx	in//IN//A_nx0PNP
truth//NN//A_nx0N1	possession//NN//A_NXN	want//VBP//A_nx0V
universally//RB//B_vxARB	of//IN//B_nxPnx	of//IN//B_vxPnx
acknowledged//VBD//A_nx0V	a//DT//B_Dnx	a//DT//B_Dnx
,//,//B_PUvxpu	good//JJ//B_An	wife.//NN//A_NXN
that//IN//B_COMPs	fortune//NN//A_NXN	...EOS...//...EOS...
a//DT//B_Dnx	,//,//A_PU	...EOS...//...EOS...

Table 4.1: The sentence "It is a truth universally acknowledged, that a single man in possession of a good fortune, must be in want of a wife" analyzed with a trigram supertagger. The entries are of the form w_i//tag$_i$//supertag$_i$.

simple POS tags.

The XTAG research group at the University of Pennsylvania[7] provides a freely available supertagger. As an example of the output of such a supertagger, we consider the first sentence of the novel *Pride and Prejudice* by Jane Austen. The output is shown in Table 4.1 in form of triples. The first item corresponds to the word, the second to its POS tag and the third item is the supertag identifier. For a list of all supertags used by the supertagger, see (Srinivas, 1997a), Appendix A.

4.3.1 Lightweight Dependency Analysis

The simple supertagging approach based on n-grams helps to reduce the possible number of supertags for each word of a sentence and hence facilitates the task of the parser. But there is another aspect to the dependencies coded in the elementary structures. We can use them to actually derive a shallow parse of the sentence in linear time. The procedure is presented in (Srinivas, 1997a; Srinivas, 2000) and is called *lightweight dependency analysis*. The concept is comparable to the *chunking* technique of (Abney, 1991). The lightweight dependency analyzer (LDA) finds the arguments for the encoded dependency requirements. As mentioned in Section 4.2, there exist two types of "slots" that can be filled. On the one hand, nodes marked for substitution have to be filled by the complements of the lexical anchor, whereas on the other, the foot nodes (i.e. nodes marked for adjunction) take words that are being modified by the supertag. In the supertagging terminology, one distinguishes a *derived* tree from a *derivation* tree. The derived tree is used for showing the phrase structure of the parsed sentence and the derivation tree embodies the dependency links obtained from the LDA. Both trees for the sentence *who does Bill think Harry likes* from Section 4.2.2 are shown in Figures 4.6 and 4.7. The parent nodes of

[7] see `http://www.cis.upenn.edu/~xtag/`

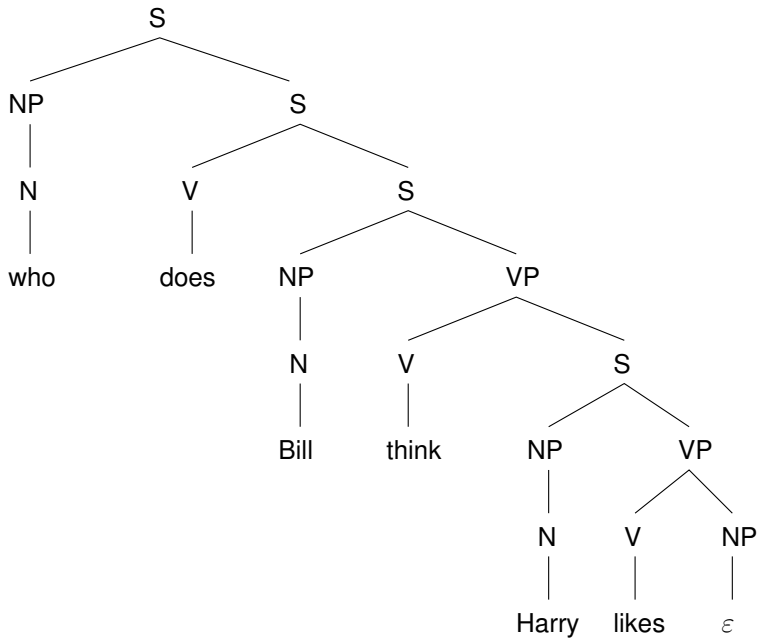

Figure 4.6: The derived tree for the sentence *who does Bill think Harry likes*.

the derivation tree can be interpreted as the head words, i.e. the lexical anchors, of the supertags (cf. Fig. 4.4) and their children are the complements and modifiers, accordingly.

The algorithm of the lightweight dependency analyzer is given in Figure 4.8. It computes the dependency links for the n words of a given sentence in linear time. In order to achieve this, it first takes the modifier supertags, i.e. the ones that are adjoined to a node, and computes the dependency links for them. Every node is associated with polarity values that reflect the directions of its arguments. For example, the tree β_1 in Figure 4.4 takes an NP (complement) to the left and an S (modifier) to the right of the anchor *think*, which is noted with a plus or minus sign in front of the node, respectively. In this case, the node requirements of β_1 can be coded as "$-\mathsf{NP}\bullet\ +\mathsf{S}*$", where the bullet "$\bullet$" symbolizes a complement relation and the asterisk "$*$" a modifier relation. After having computed all dependencies for the modifier supertags of the sentence, the second step works on the remaining supertags (the substitution nodes) in order to obtain the links for their complements similar to the procedure in the first step. For the example, i.e. the sentence dealing with long distance extraction, the LDA result is summarized in Table 4.2. The notation of "$\bullet$" and "$*$" was introduced in (Srinivas, 1997a) and is also used by the supertagger that is freely available from the XTAG research group.

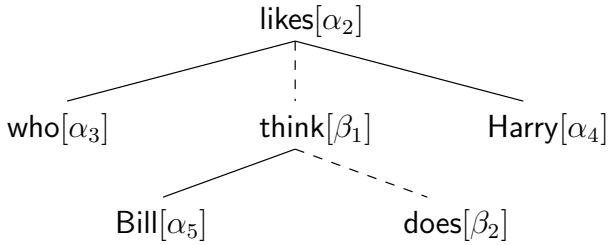

Figure 4.7: The derivation tree that is computed by the LDA for the sentence from Figure 4.6. Dashed lines are adjunction operations, solid lines denote substitution.

Step 1: For each modifier supertag s in the sentence

- compute the dependencies for s
- mark the words serving as complements as unavailable for step 2

Step 2: For the non-modifier supertags s in the sentence

- compute the dependencies for s

Compute dependencies for s_i of w_i: For each external node d_{ij} in s_i do

- connect word w_i to the nearest word w_k to the left or right of w_i depending on the direction of d_{ij}, such that label(d_{ij}) $\in$ internal_nodes(s_k) skipping over marked supertags, if any.

Figure 4.8: The two-pass LDA algorithm (from (Srinivas, 2000)).

Pos	Word	Supertag	Requirements	Step 1	Step 2	Dependencies
0	who	α_3	$\varnothing$	–	–	–
1	does	β_2	$+\mathsf{S}*$	$3*$	–	$3*$
2	Bill	α_5	$\varnothing$	–	–	–
3	think	β_1	$-\mathsf{NP}\bullet\ +\mathsf{S}*$	$2\bullet\ 5*$	–	$2\bullet\ 5*$
4	Harry	α_4	$\varnothing$	–	–	–
5	likes	α_2	$-\mathsf{NP}\bullet\ -\mathsf{NP}\bullet$	–	$0\bullet\ 4\bullet$	$0\bullet\ 4\bullet$

Table 4.2: The summary of the LDA on the sentence *who does Bill think Harry likes*. The last column shows the final dependency links found by the procedure. The signs "+" and "−" indicate whether the dependencies are located to the right or left of the lexical anchor, whereas "•" and "∗" denote the type of the relation, namely complement or modifier, respectively.

The lightweight dependency analyzer presented in this section is an efficient means of determining the dependency links of the supertags that are annotated to the words of a sentence. For a sentence length of n, the LDA's runtime is linear, i.e. $O(n)$. It is a robust procedure and thus can be used to produce even partial linkages that span only a fraction of the sentence. If there are wrongly annotated supertags, an LTAG parser fails to parse the whole sentence. But the LDA is able to find "working islands" within the sentence without failing, thus resulting in a partial parse.

4.3.2 Bäcker's Supertagger

The supertagger for the German language was firstly introduced in (Bäcker, 2001). The initial version was based on bigrams and used only simple smoothing (*add-one* smoothing, see e.g. in (Jurafsky and Martin, 2000)). It achieved an accuracy of 65.5%. In (Bäcker, 2002), this supertagger was enhanced with a language model based on trigrams instead of bigrams and also implemented Good-Turing discounting and Katz's back-off (similar to the English supertagger in (Srinivas and Joshi, 1999)), resulting in a better accuracy of 78.3% (see also (Bäcker and Harbusch, 2002)). It is used as a starting point for the n-best supertagger developed in this thesis. As described in Section 4.1.2, the probabilities of the supertag sequences are estimated by several independence assumptions, resulting in

$$P(t_1 t_2 \cdots t_N) \approx \prod_{i=1}^{N} P(t_i|t_{i-2}t_{i-1})P(w_i|t_i) \tag{4.8}$$

with pseudo supertags at the beginning of the sentence, i.e. for t_j with $j < 1$. One of the problems when thinking of an implementation is the limited precision of computers. For longer sentences, the probabilities of Equation 4.8 are multiplied and

the product gets smaller and smaller. Since many probabilities have a value $p \ll 1$, the resulting value cannot be expressed correctly since it tends to converge to zero very fast. One trick to prevent this numerical underflow is to use *logarithms* of the probabilities and sum the values instead of multiplying them. This is also known under the name of *logprob* or *log likelihood* (cf. (Manning and Schütze, 2000)). So wherever probabilities are multiplied in formulas, the implementation actually sums the logprobs in order to prevent numerical underflow.

The supertagger implements a second-order Hidden Markov Model (i.e. the probabilities are based on supertag trigrams) and estimates the parameters directly on the training set, which is a collection of sentences where the words are annotated with the corresponding supertags (cf. Section 6.1). The basic estimation technique is ELE and the final values are smoothed by discounting and back-off as presented in Sections 2.2.1 and 2.2.2. The implementation is very efficient in terms of both memory consumption and runtime. In the POS tagging (and also supertagging) framework, the states of a second-order HMM are pairs of tags $\langle t_{i-1}, t_i \rangle$ (or in this case supertags) (see e.g. in (Abney, 1997)).[8] This allows for writing $P(\langle t_{i-1}, t_i \rangle | \langle t_{i-2}, t_{i-1} \rangle)$ for the probability $P(t_i | t_{i-2} t_{i-1})$ (which relates to the state transition probabilities of matrix A) and $P(w_i | \langle t_{i-1}, t_i \rangle)$ for the probability $P(w_i | t_i)$ (which relates to the observation symbol probabilities of matrix B). These modifications can be used for a direct implementation of the Viterbi algorithm presented in Section 2.1.2. The output symbols correspond to words that are tagged with supertags t_i and are emitted in the states $\langle \_, t_i \rangle$. The pseudo category $\varnothing$ at the beginning of a sentence is used to allow for the states $\langle \varnothing, t_i \rangle$ for the first time frame (i.e. first word).

In addition to Good-Turing discounting and Katz's back-off, another back-off that is implemented in the supertagger is the handling of unknown words after (Weischedel et al., 1993). If an unknown word is encountered, the probability $P(w_i | t_i)$ in Equation 4.8 is zero. An intuitive way to estimate this probability is to consider the distribution of categories on large corpora. It is more likely that unknown words are nouns or verbs than e.g. particles. For general taggers, another characteristic that can be examined is the spelling of the word. The prefix and suffix of an unknown word might be a good indicator for its tag or supertag. Common word endings of German nouns are e.g. *-heit* and *-keit*, whereas adjectives often end in *-lich* or *-bar*. Bäcker's supertagger uses the following back-off formula to handle unknown words:

$$P(w_i | t_i) = \begin{cases} P_{\mathrm{MLE}}(w_i | t_i) & \text{if } C(w_i, t_i) > 0 \\ \frac{N_1(t_i)}{C(t_i)} \cdot P(\text{features} | t_i) & \text{otherwise.} \end{cases} \tag{4.9}$$

The term $\frac{N_1(t_i)}{C(t_i)}$ estimates the likelihood of the unknown word by counting the words that occurred in the training corpus exactly one time with supertag t_i and dividing

[8] If we considered bigrams (i.e. a first-order HMM), the states would be single tags.

it by the total count of t_i.[9] The second term, $P(\text{features}|t_i)$, looks at the affixes of w_i and uses the probabilities of words already encountered during training with the same pre- and suffixes, such that they were annotated with supertag t_i.

The memory requirements for the matrices A and B of the HMM are minimized by storing the probabilities associatively with so-called *hash tables* (cf. (Knuth, 1998)). This technique enables us to associate the states (i.e. supertag pairs) with the corresponding logprobs and retrieve the values very efficiently. The HMM parameters are calculated in the training phase where expected likelihood estimation with Good-Turing discounting and Katz's back-off is applied to the annotated sentences. It is obvious that only states with a non-zero probability need to be stored. Unnecessary states are thus discarded which saves additional space. It is important to note that the memory requirements of the HMM with $N^2 \times N^2$ for matrix A (N being the number of supertags) and $M \times N^2$ for matrix B (M being the number of words) are still in $O(N^4 + N^2M)$ (N has to be squared since we consider *pairs* of supertags for states, cf. also Section 4.1.3). But this worst case usually never happens since most of the word/supertag combinations do not occur in natural languages.

The supertagger is modeled in a way that allows for easy extension. An overview in form of an UML class diagram is given in Appendix A, Figure A.1 on page 102. The training of the HMM is realized by the class TrigramSuperTaggingTrainer. It reads the sentences of a corpus and adds the smoothed trigram, bigram and unigram estimates (the latter ones for back-off) to an instance of the class TrigramDataManager which manages all probabilities of the language model. This data manager is also used within the class TrigramSuperTagger which implements the Viterbi algorithm and thus finds the most likely supertag sequence for a sentence according to the language model stored in the data manager. The class SuperTaggingEvaluator takes all sentences of a test corpus and applies the trigram supertagger to each. It also gathers some statistics and prints them out after having processed a sentence. The other classes are of abstract nature or denote interfaces. They allow for extension of the framework with other language models that are not based on HMMs or that use e.g. four-grams instead of trigrams.

These classes serve as a starting point for the extended supertagger developed for this thesis. One modification is the handling of ambiguous codings, adding a keyboard specific lexicon that is used for disambiguation and implementing several evaluation modules. The other complex extension is an n-best approach that will be presented in Section 5.1. This step is necessary because it is not sufficient to only look at the best supertag sequence of the Viterbi algorithm and base the improvements to the candidate lists solely on this hypothesis. We need at least several good hypotheses

[9] It is quite common to treat *hapax legomena*, i.e. words that occur only once, as if they never occurred. This is a key concept, as formulated in (Jurafsky and Martin, 2000): "use the count of things you've seen once to help estimate the count of things you've never seen".

to improve the results significantly. An overview on traditional search techniques is described in Section 4.5. But first, additional shallow parsing methods beside supertagging are discussed in the next section.

4.4 Other shallow parsing techniques

So far, supertagging has been presented as a possible means to partially parse a sentence and thus gain knowledge about the syntactical structures to a certain extent without having to derive a full parse tree. Other existing approaches to this domain are e.g. the *chunking parser* in (Abney, 1991) and a partial parser called *Fidditch* by (Hindle, 1994). The basic goal of partial parsing is to overcome the disadvantages of traditional parsing techniques which often utilize huge grammars with thousands of rules that try to reflect as many language-specific patterns of correct word orderings as possible, thus making them rather slow and error-prone for ill-formed input (e.g. for partly ungrammatical input from an automatic speech recognition system). Therefore, the development of robust and efficient parsing alternatives is a desirable task. In *principle-based parsing* (Berwick, 1991), the numerous rules of traditional grammars are covered by much smaller sets of fundamental principles that can act as *filters* (used to rule out possible structures) or *generators* (allowing for new structures). The combination of these principles by a deductive inference procedure results in new constructions that are not explicitly coded in the principle set (e.g. passive structures as in *Mary was kissed by John*).

The following two sections give a short overview on alternative approaches in the area of partial parsing. The decision to use supertagging for the task of sentence-wise ambiguous text entry was motivated more pragmatically. First, TAGs provide a concise formalism that is easily grasped but nevertheless has a high descriptive power. Second, the employment of lightweight dependency analysis yields an efficient means to derive partial analyses of text in linear time. Last but not least, the existing supertagger was a good starting point for further improvements, like e.g. the n-best approach (cf. Section 5.1).

4.4.1 Chunking

A more psycholinguistically motivated approach to partial parsing is *chunking* (Abney, 1991) which clusters the words of a sentence according to so-called *major heads*. These major heads can be basically seen as the content words and form the core of a chunk. For example, in *the bald man was sitting on his suitcase* the words *man*, *sitting* and *suitcase* are the major heads (from (Abney, 1991)). The chunking parser is split into two modules, namely the *chunker* and the *attacher*. The former renders a stream of words into a stream of chunks, e.g. "[*the bald man*][*was sitting*][*on his*

suitcase]", whereas the latter takes the chunks and attaches them to each other resulting in a complete parse tree. The chunker is a nondeterministic LR parser (see e.g. (Aho et al., 1986)) and has to cope with two problems:

- the endings of chunks might not be deterministic, i.e. it has to deal with ambiguities concerning different chunk lengths and
- ambiguities arise for words that can have more than one category, e.g. *time* which can be a noun or verb.

The chunker uses best-first search (cf. Section 4.5.2) to find the most promising task (a tuple comprising the current configuration, a next action and a score) for a set of possible actions. The score is determined by several heuristics that estimate the likelihood of a particular action leading to the overall best parse. Since the syntactic attachment ambiguities are dealt with in the attacher, the chunker itself is quite efficient and uses a simple context-free grammar with 17 basic rules[10] that is able to produce most of the chunks that arise in English texts. The basic task of the attacher is to resolve attachment ambiguities. It is based on the same machinery as the chunker, but utilizes additional heuristics and also incorporates subcategorization frames which define a list of slots that can be filled by arguments the heads of each chunk can take. This relates to the approach of the lightweight dependency analysis which also tries to satisfy the dependency requirements of each supertag. In (Srinivas, 1997b), supertagging and LDA are used for text chunking and reach a precision and recall of 91.8% and 93% for noun chunking and 91.4% and 86.5% for verb chunking, respectively.[11]

4.4.2 Deterministic partial parsers

In (Hindle, 1994), it is stated that a parser for unrestricted text should fulfill the following requirements:

- ignore ungrammaticality, i.e. always provide a syntactic analysis for any input,
- return a partial analysis when a complete analysis cannot be achieved,
- give only one single analysis for each text input, i.e. deterministically arrive at a partial analysis (which is then probably underspecified),
- process text reasonably fast and
- represent the linguistic information obtained by the parser in a declarative form such that it can be reused by other systems or improved versions of the parser for further analysis.

[10] More rules are actually covered by regular expressions within the right-hand sides of some rules.
[11] For a definition of the terms *precision* and *recall*, see e.g. (Manning and Schütze, 2000).

These constraints are primarily dealt with in the system *Fidditch* (Hindle, 1994) which is a deterministic parser and thus implicitly covers several of the above requirements. After a lexical analysis step (word tokenization and retrieval of lexical features from the lexicon), the system first applies part-of-speech disambiguation to the text either by an own internal rule-based POS tagger or by using the program described in (Church, 1988). After the preprocessing, Fidditch incrementally builds a syntactic analysis for the word sequence. The different parser states are managed by nodes on a stack. An additional buffer holds already completed nodes which can be *bound*, i.e. attached, to nodes on the stack resulting in partial descriptions spanning more than one word. If some nodes cannot be further processed or if there are ambiguities, they are left unattached. Thus, vagueness and ambiguity are encoded through "underspecification".

Another partial parsing technique is based on *finite state automata* (FSA) (see e.g. (Roche and Schabes, 1997)) which can efficiently parse regular expressions. Since simple regular expressions are not sufficient to represent more complex phrases, *cascaded* FSA are used to cover recursive phenomena to some extent. The FASTUS system (Hobbs et al., 1997), e.g., is divided into five levels of processing: recognition of complex words, basic phrases, complex phrases, domain events and merging structures. So the system starts with small chunks in the beginning and groups them into larger units with each next level. FASTUS is primarily used in the area of *information retrieval* (see e.g. (Sparck Jones and Willett, 1997)) from natural-language texts where this kind of "simple" processing (when compared to traditional approaches that tried full parsing and in-depth semantic analysis) shows fruitful results.

4.5 Search methods

In the field of classic *Artificial Intelligence*, one often deals with finding ways to solve general problems. This can reach from small toy problems like e.g. the 8-puzzle or playing Tic-Tac-Toe to real-world problems including more complex tasks such as route finding, VLSI layout or assembly sequencing (Russell and Norvig, 1995). Usually, this is achieved by systematically searching through the *state space* of the problem. The definitions and properties of the search methods presented in this section are based on (Russell and Norvig, 1995), an excellent introduction to this field.

A *problem*, in the most general sense, can be defined as a 4-tuple consisting of

- an *initial state* where the search starts,
- a set of *operators* that define what state is reached after carrying out an action,
- a *goal test* that determines whether the current state description is a solution to the problem and

- a *path cost function* that assigns costs to the partial paths and is often used to guide the search towards a solution.

The state space can be modeled in terms of graph theory. A *graph* $G = (V, E)$ consists of a set of vertices $V = \{v_1, v_2, \ldots, v_n\}$ denoting states of the problem and a set $E = \{e_{ij}\}$, $1 \leq i, j \leq n$ that holds all edges that each connect two vertices $(v_i, v_j) \in V^2$. A *path* from vertex v_i to v_j is a list of vertices $v_i, v_{k_1}, \ldots, v_{k_m}, v_j$ that are successively connected by edges $e_{ik_1}, e_{k_1k_2}, \ldots, e_{k_mj}$. If the graph contains no cycles, we call it a *tree*.[12] When speaking of trees, the vertices are often called nodes. In this thesis, the graph representing the trellis structure used in the forward Viterbi step (cf. Section 2.1.2) is a tree, i.e. it contains no cycles. At any position in the sentence, there are no edges back to previous words.

The state space of a problem can be examined through its representation as a graph. There are various algorithms that are capable of looking at all vertices of a graph in order to find a solution, i.e. find a vertex where the goal test introduced in the problem definition above is positive. As two main search paradigms, one can distinguish uninformed from informed search methods. While the former systematically expands all nodes in a fixed order, the latter uses path cost functions and heuristics to guide the search towards the goal.

4.5.1 Uninformed search

The basic approach of tree searching algorithms is to successively generate new sets of states by expanding the current state. This expansion of nodes builds up a search tree. The general search algorithms differ in the way which nodes are expanded next. Figure 4.9 shows the two main approaches, namely expanding the deepest node first (*depth-first* search) and expanding all nodes at depth d before expanding the nodes at depth $d+1$ (*breadth-first* search). In this example, the branching factor b, i.e. the number of possible successors for a node, is of fixed length 3, whereas the solution is expected at depth $d = 2$ (the root node is considered to be at depth 0).

There are four criteria that play an important role when dealing with search strategies:

- completeness: does the search strategy find a solution if there is one?
- optimality: is the solution found by the search strategy also the best solution to the problem?
- time complexity: how long does it take to find a solution?
- space complexity: how much memory does the search need?

[12] Even if the graph contains cycles, we can transform it to a sub-graph that contains no cycles which is called a *spanning tree*. It has the same number of vertices but only as much edges as to make up a proper tree.

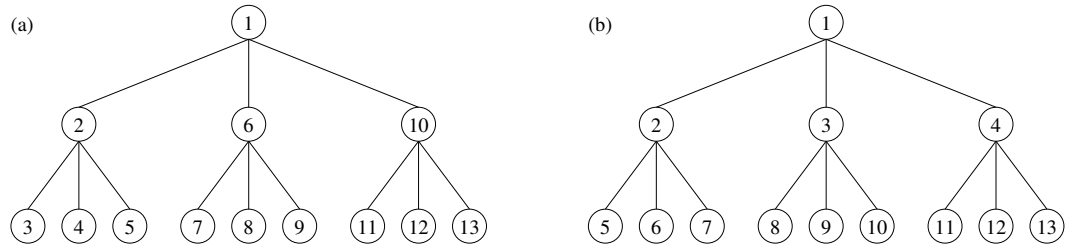

Figure 4.9: (a) Depth-first search. (b) Breadth-first search. The numbers show the order in which the nodes are visited.

The complexity measures are expressible in terms of the branching factor b and the depth of the solution d or the maximum depth m of the tree. In general, time complexities of search algorithms are always exponential. The upper bound for breadth-first search is $O(b^d)$ because if one has to look at all nodes, the maximum number of expansions is $1 + b + b^2 + \cdots + b^d$ in the worst case. For depth-first, the time complexity is $O(b^m)$. One pitfall of depth-first search is that it is not guaranteed to find a solution, i.e. it is not complete. The fact that the deepest node is always expanded next results in getting stuck if the search descends an infinite path with no solution in it. This also leads to depth-first being not optimal. Consider the left tree in Figure 4.9. If the nodes labeled with 5 and 6 are solutions, depth-first will find the lower-quality solution first. In contrast, breadth-first is both optimal and complete. But it also has a big disadvantage when compared to depth-first, namely its memory requirements. In breadth-first, we have to keep all nodes on the current frontier in memory. In this case, the maximum number of nodes is in $O(b^d)$, resulting in exponential memory usage which can be quite a bigger problem than the exponential running time. On the other hand, depth-first only needs to store $O(bm)$ nodes at a time.

There exists a search strategy that combines the moderate memory usage of depth-first with the optimality and completeness criteria of breadth-first called *iterative deepening*. It iteratively applies depth-first searches with an increasing maximum depth d_m. So all possible depths $0, 1, 2, \ldots$ are tried until a solution is found. As mentioned above, the algorithms for depth-first and breadth-first search differ only in which nodes are expanded next. The queuing function of the former search method enqueues nodes at the front whereas the latter search strategy enqueues nodes at the end.[13] A detailed overview on these and additional search strategies can be found in (Russell and Norvig, 1995).

[13] These operations can be easily implemented through the use of a stack and a queue, respectively (see e.g. (Knuth, 1997)).

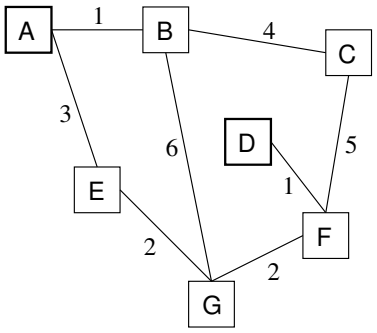

Figure 4.10: The city map example. Annotated numbers are the distances between the cities. The starting point is city A, the goal is city D.

4.5.2 Informed search

The search strategies presented in the previous section do not take into account problem-specific knowledge. Uninformed search techniques end up in a systematic exploration of the search space. In most cases, this can be very inefficient. Usually, the different states can be associated with certain costs for their expansion. Or to put it another way, we can define an evaluation function $f(n)$ which determines the utility of expanding a given node n. With this knowledge, it is possible to guide the search towards a solution much faster.

Best-first search

The most intuitive way of optimizing uninformed search methods is to order the nodes in the queue according to the value of an evaluation function. Nodes with low costs, i.e. where the evaluation function returns a high utility, are moved to the front of the queue thus minimizing the total path cost. So, the best node is always expanded next. Consider the example of traveling from one city to another (cf. Fig. 4.10). The value of the path cost function $g(n)$ denotes the accumulated distance between the starting city and the current city n that one traveled to so far. This strategy is usually called *best-first search* since the node with overall smallest cost appears to be the best choice at the moment. Although this strategy is optimal and complete, it tends to expand many unnecessary nodes that are not part of the optimal path thus making it very inefficient. The nodes that are expanded in the example are $\mathsf{A}_0, \mathsf{B}_1^{\mathsf{A}}, \mathsf{E}_3^{\mathsf{A}}, \mathsf{C}_5^{\mathsf{B}}, \mathsf{G}_7^{\mathsf{B}}, \mathsf{G}_5^{\mathsf{E}}, \mathsf{F}_{10}^{\mathsf{C}}, \mathsf{F}_7^{\mathsf{G}}, \mathsf{F}_9^{\mathsf{G}}, \mathsf{D}_9^{\mathsf{F}}$, where $\mathsf{X}_n^{\mathsf{Y}}$ means that n is the total path cost of node X whose parent is node Y (cf. also Fig. 4.11 (a)). It is assumed here that an additional check prevents parent nodes being expanded from a child node which would result in even worse behavior.

city	A	B	C	D	E	F	G
$h(n)$	5	3	4	0	3	1	2

Table 4.3: The straight-line distances for the city example.

Greedy search

If a different strategy is chosen to reach the goal, one might be able to expand less nodes than with the simple best-first search. For this purpose, the strategy takes into account the estimated path cost from the current to the goal node. So instead of keeping track of the summed path cost so far, a different evaluation function called a *heuristic* $h(n)$ is used to estimate the cost from the state at node n to reach the goal. This search strategy is called *greedy search* because it always tries to jump towards the goal in biggest possible steps. It is also similar to depth-first search since it follows down a path and only backs up when it hits a dead end. Unfortunately, greedy search is neither optimal nor complete. Nevertheless, one characteristic is that it finds good-quality solutions very fast,[14] although they often are not the optimal ones. Both time and space complexity has a worst-case upper bound in $O(b^m)$. For the example from Fig. 4.10, the heuristic is assumed to be the straight-line distance of the cities to the goal which is shown in Table 4.3. With this heuristic, the search expands the following nodes: $\mathsf{A}_0, \mathsf{B}_3^{\mathsf{A}}, \mathsf{E}_3^{\mathsf{A}}, \mathsf{G}_2^{\mathsf{B}}, \mathsf{C}_4^{\mathsf{B}}, \mathsf{F}_1^{\mathsf{G}}, \mathsf{D}_0^{\mathsf{F}}$. Now, the node costs represent the corresponding distances to the goal and not the cumulative path cost, so as the search progresses, the values are not added. As can be seen, greedy search generates less nodes than best-first but it does not find the optimal solution, which is the trip from A over E, G, F to D. Greedy search "chooses" the alternative path via city B instead of E.

A* search

So far, the two approaches best-first and greedy search have been introduced. The former strategy minimizes the path cost so far and is optimal and complete but tends to expand many nodes thus making the search very inefficient, whereas the latter minimizes the estimated path cost to the goal but is neither optimal nor complete. Intuitively, the combination of these two evaluation functions should result in a strategy that merges the advantages of both approaches, namely the optimality and completeness of best-first and the efficiency of greedy search. Fortunately, this is achieved by simply summing the two evaluation functions:

$$f(n) = g(n) + h(n). \tag{4.10}$$

[14] depending on the quality of the heuristic

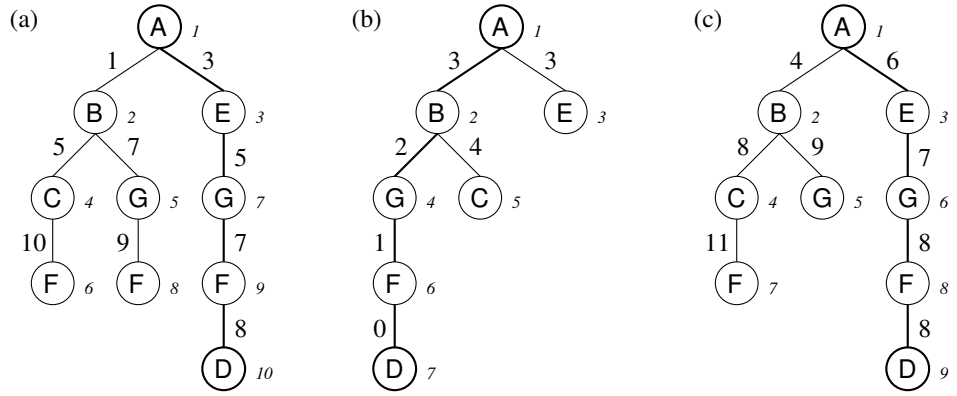

Figure 4.11: The search trees for the city map example: (a) Best-first search. (b) Greedy search. (c) A* search. The values of the corresponding evaluation functions are annotated at the edges, italic numbers denote the order in which the nodes are expanded.

One important criterion of the heuristic is that it is not allowed to overestimate the cost towards the goal state, i.e. it is *admissible*. Along with the property that the total path cost never decreases, one can prove that this strategy yields an optimal and complete approach. The use of the best-first strategy together with the evaluation function in Equation 4.10 is known as *A* search* (Nilsson, 1998).

A* search is used throughout a large variety of applications and is the standard search algorithm for problems in the area of Artificial Intelligence. Nevertheless, it is only as good as the heuristic h. In fact, it can be shown that an exponential growth of node expansions will occur if the error in the heuristic grows faster than the logarithm of the real path cost. Unfortunately, this is the usual case with most practical heuristics. So again, we have to live with a drawback of possibly running out of memory. For the given example, A* search only expands one node less than the best-first approach. This is probably due to the (rather arbitrary constructed) values of the heuristic. But for larger problems, the heuristic is essential to a fast process of locating the optimal solution. In the city map example, A* search produces the following nodes: $\mathsf{A}_0, \mathsf{B}_4^{\mathsf{A}}, \mathsf{E}_6^{\mathsf{A}}, \mathsf{C}_8^{\mathsf{B}}, \mathsf{G}_9^{\mathsf{B}}, \mathsf{G}_7^{\mathsf{E}}, \mathsf{F}_8^{\mathsf{G}}, \mathsf{F}_{11}^{\mathsf{C}}, \mathsf{D}_8^{\mathsf{F}}$. The overall search trees of the three approaches presented in this section are delineated in Figure 4.11.

4.5.3 N-best search

The Viterbi algorithm presented in Section 2.1.2 is capable of finding the best state sequence very fast. If the task is enhanced to find other optimal state sequences, or more generally the n best ones, the Viterbi algorithm is not sufficient. One approach is to use general informed search techniques as presented in the previous section. One

problem with greedy and A* search for the particular case of finding best supertag sequences is the definition of a good heuristic. Since it is not known what final probability will be assigned to the overall sequence, it is hard to determine a working heuristic. But we can take advantage of the probabilities obtained in the Viterbi step. If a small modification is applied to the forward Viterbi search, namely that all instead of the best predecessor states are stored for every state of the HMM, these values can be used as a heuristic when searching in reverse order, i.e. starting at the end of the sentence and successively searching towards the beginning. The advantage of this kind of "heuristic" is that it provides the optimal values, thus maximizing the efficiency of the search process.

There are several n-best algorithms, especially in the domain of speech recognition (see e.g. (Schwartz et al., 1996)). The task of speech recognition requires ways of finding more than one good hypothesis for an utterance since the speech signal is usually noisy or the sentence is phonetically ambiguous. The noise corrupts the feature vectors and results in not recognizing the sentence correctly. In order to improve recognition accuracy, the search for the best hypothesis is divided into more than one step. The n-best search paradigm (Chow and Schwartz, 1989) uses different *knowledge sources* (KS) which are applied successively to sort out bad hypotheses. The first KS generates a list of n best sentence hypotheses out of the speech input. This ordered sentence list is passed to other KSs which reorder the list according to more complex properties of natural language, as e.g. analysis of syntax, semantics, discourse and pragmatics. The KSs are ordered such that the KS with lesser cost but more constraint comes first, resulting in a fast way of filtering or rescoring the hypotheses. KSs that cost most are applied last when the list of sentence hypotheses has shrunk to only a few top choices.

The task of finding n-best sentence hypotheses when typing with ambiguous keyboards, as presented in Section 3.3.1, is not as complex as in speech recognition. Here, we assume that the word boundaries are known to the system. In speech recognition, hypotheses for different words may span over different time frames, like e.g. in *chart switch* vs. *charts which* where the first word is slightly longer in the latter case. Therefore, the search state space is much larger because the recognition system has to identify the word boundaries as well. For example, consider the sentence "*eight sheep can each eat cheaply*" (from (Laver, 1994)). For the speech recognizer, this looks something like "eɪtʃiːpkəniːtʃiːttʃiːplɪ". In particular, the difficulty lies within the task to distinguish the spelling sequences "–ght sh–", "–ch" and "–t ch–" which all produce the same sound "tʃ". In ambiguous typing, we "only" have to determine the most likely word among a list of candidates for a given code sequence. As was mentioned in Section 3.4, these lists can grow up to 75 entries. So especially for long sentences, the search space can be very large.

A comparison of n-best algorithms (traceback-based, exact sentence-dependent and word-dependent n-best algorithm) is given in (Schwartz et al., 1996). The first

one (traceback-based n-best)[15] is an algorithm that finds only an approximate list of the n best sentences. It consists of a forward step similar to a Viterbi search where the only difference lies in the information stored in the backpointer lists. Viterbi stores only the best predecessor for each frame and state. In the traceback-based algorithm, a list of predecessor states together with their scores is stored. For each state, the score and a backpointer of the best hypothesis is passed forward for future scoring. A simple recursive search is applied at the end of the sentence which gathers all sentences with a score above some threshold below the best theory. The advantage of this procedure is that it is only slightly slower than the Viterbi search (which equals 1-best). Unfortunately, it is prone to underestimate or completely miss high scoring hypotheses.

The exact n-best algorithm (Chow and Schwartz, 1989) finds the top n sentence hypotheses but it suffers from being very inefficient for large values of n. It computes the probabilities of word sequences rather than state sequences.[16] The algorithm keeps separate records for theories, i.e. paths through the word lattice, with different word sequence histories. If more than one theory with the same history crosses the same state and time frame, the probabilities are added. Only a specified maximum number m of theories whose probabilities are within a threshold of the most likely word sequence's probability is kept for each state and the rest is pruned due to computational issues.

The word-dependent n-best algorithm (Schwartz and Austin, 1991) is a compromise between the traceback-based and exact (sentence-dependent) algorithm. The idea is that the best starting time of a word only depends on the previous word and probably not on any word before that. Therefore, the local theories are not based on the whole preceding sequence, they only consider the previous word. Again, a threshold m specifies how many different preceding word probabilities are stored. At the end of the sentence, a recursive traceback is applied that retrieves the list of best sentences. Although the word-dependent algorithm is not optimal, its performance in terms of speed is better than that of the exact search and it tends to find better solutions than the traceback-based algorithm because it does not prune away word sequences that could turn out to be correct.

The basic features of the n-best search that is used in this thesis to find the top n supertag sequences of a coded sentence are presented in (Soong and Huang, 1991) for the task of speech recognition. A more worked out version of this algorithm can also be found in (Purnhagen, 1994). The next chapter gives a detailed description of this n-best tree-trellis search and the necessary steps to incorporate the codes of the ambiguous keyboard into the overall framework of the n-best supertagger.

[15] In (Schwartz and Austin, 1991), it is called lattice n-best.

[16] In speech recognition, the states of the HMM usually correspond to sub-word units like triphones, diphones or even phonemes.

5 N-best Supertagger for Ambiguous Typing

This chapter gives a detailed overview on the n-best supertagger and its implementational aspects. So far, the theoretical background has been introduced in the previous chapter. In the following sections and based on the supertagger developed in (Bäcker, 2002), the modifications that are necessary for handling ambiguous codes will be presented. The primary goal of this study is to examine whether a supertagging-based approach to ambiguous typing on a sentence level is promising. The overall architecture of the system therefore consists of various components that take plain sentences and simulate the user by typing the words with a given ambiguous keyboard, build supertag hypotheses of ambiguous code sequences, generate the candidate lists according to the n-best framework and finally compute all statistics which are summarized in Chapter 6.

The basic idea is to use a better language model than the simple model based on unigram frequencies that was presented in Chapter 3. The sequence of supertags for a given corpus is modeled with a trigram Hidden Markov Model (HMM) and its parameters are estimated from an annotated training corpus. Usually, a dynamic programming technique (i.e. the Viterbi algorithm) finds the best supertag sequence of a sentence for the given HMM efficiently. Here, in addition to this forward-trellis step, a backward-tree search is applied in order to find the n most promising supertag sequences which are used to adjust the candidate lists and move likely matches to the top. In a second step, a *lightweight dependency analysis* (Srinivas, 1997a) on the list of supertag hypotheses found by the n-best search is used to determine likely chunks of the sentence. This method is applied in order to discard hypotheses that have syntactic inconsistencies. The information that is provided by the surviving hypotheses is used for additional final adjustments of the candidate lists.

In the following, Section 5.1 describes the two-pass n-best approach to supertagging. In a forward run, a modified Viterbi search determines the locally optimal scores of supertag sequences. A backward A* search then gathers the n best hypotheses from the end of the trellis to the front by using the stored results from the Viterbi step. The necessary modifications for integrating the n-best approach into the ambiguous framework, i.e. the handling of ambiguity and how promising hypotheses are kept for the final adjustment of the candidate lists, are given in Section 5.2.

5.1 N-best tree-trellis algorithm

The n-best tree-trellis algorithm basically combines the Viterbi trellis search from Section 2.1.2 and the A* tree search presented in Section 4.5.2. The terminology is based on the one for Markov models from Section 2.1 and the algorithm itself is presented in a way that it can be directly applied to any HMM. So it is actually independent of the target domain. For example, if we speak of *output symbols* being emitted at some *time frame* in the following, the corresponding term in the supertagging framework would be the *words* at some *position* in the sentence. As will be shown, the Viterbi algorithm has to be modified slightly such that the tabulated results of the dynamic programming technique yield additional information for finding n best state sequences instead of the single best one. For this task, the backpointer variable ψ from Equation 2.12 is enhanced to hold a rank-ordered list of predecessor states from the previous time frame. The list is sorted according to the score stored in the δ-table. Figure 5.1 summarizes the modified Viterbi algorithm based on its initial version in Figure 2.4 on page 14. The only change is Equation 5.6 where another dimension is added to the variable ψ in order to store all predecessors of the current state. This allows for finding all paths through the HMM sorted according to their score when using an optimal and complete search strategy. Since we are only interested in n best paths, the storage can be limited to n best predecessors.[1] The values for j, k and i in the expression $\psi_t(j,k) = i$ stand for the current state j which has state i as predecessor with rank k, i.e. the states are sorted according to their score $\phi_t(j,i)$ in descending order. Equation 5.7 defines the rank-ordered states for the last time frame T. This ordering is needed for the initialization of the backward tree search. We can still retrieve the best state sequence as with the normal Viterbi algorithm by simply using the states in $\psi_t(j,1)$, since the best predecessors have rank $k = 1$ (cf. Equation 5.10).

The n-best tree-trellis search is divided into two phases. First, steps 1 and 2 (initialization and recursion) of the modified Viterbi algorithm are applied in a forward manner.[2] The result is a trellis as delineated in Figure 2.3 on page 12. After that, a time-reverse (backward) tree search is carried out which asynchronously gathers all best hypotheses with the A* technique. One interesting thing to note in this approach is that the heuristic h of the A* search is not an estimated but actually the exact path cost since it utilizes the probabilities stored in the δ-table. Thus, besides the optimality of A* (i.e. its guarantee to find the best solution), the runtime of the algorithm is minimized, too. The complete path cost f of a node is obtained by merging the forward and backward partial path cost, denoted by h and g, respectively. This relation is shown in Figure 5.2.

[1] So, the algorithm in Fig. 5.1 can be used for an exhaustive search that finds *all* hypotheses but the actual implementation uses only n best predecessors.

[2] Step 3 in Figure 5.1 is only applied for a 1-best search where the A* backward search is superfluous.

1. Initialization:

$$\delta_1(i) = \pi_i b_i(o_1) \qquad 1 \le i \le N \tag{5.1}$$
$$\psi_1(j,k) = 0 \qquad 1 \le j,k \le N \tag{5.2}$$
$$\phi_1(j,0) = -\infty \qquad 1 \le j \le N \tag{5.3}$$

2. Recursion:

$$\delta_t(j) = \max_{1\le i\le N} \big[\delta_{t-1}(i)a_{ij}\big] b_j(o_t) \qquad 2 \le t \le T, \quad 1 \le j \le N \tag{5.4}$$
$$\phi_t(j,i) = \delta_{t-1}(i)a_{ij}b_j(o_t) \qquad 2 \le t \le T, \quad 1 \le i,j \le N \tag{5.5}$$
$$\begin{aligned}\psi_t(j,k) = i \qquad & 1 \le i,j,k \le N, \quad 2 \le t \le T, \\ \text{s.t. } \forall k_l,k_m,i_l,i_m: \quad & \big(\psi_t(j,k_l) = i_l \wedge \psi_t(j,k_m) = i_m \\ & \wedge\ \phi_t(j,i_l) > \phi_t(j,i_m) \\ & \wedge\ k_l < k_m\big)\end{aligned} \tag{5.6}$$
$$\begin{aligned}\psi'_T(k) = i \qquad & 1 \le i,k \le N, \\ \text{s.t. } \forall k_l,k_m,i_l,i_m: \quad & \big(\psi'_T(k_l) = i_l \wedge \psi'_T(k_m) = i_m \\ & \wedge\ \delta_T(i_l) > \delta_T(i_m) \\ & \wedge\ k_l < k_m\big)\end{aligned} \tag{5.7}$$

3. Termination and path (state sequence) backtracking:

$$p^* = \max_{1\le i\le N} \big[\delta_T(i)\big] \tag{5.8}$$
$$q_T^* = \operatorname*{argmax}_{1\le i\le N} \big[\delta_T(i)\big] \tag{5.9}$$
$$q_t^* = \psi_{t+1}(q_{t+1}^*, 1) \qquad t = T-1, T-2, \ldots, 1 \tag{5.10}$$

Figure 5.1: The modified Viterbi algorithm for the forward trellis search (notation from (Rabiner, 1989)).

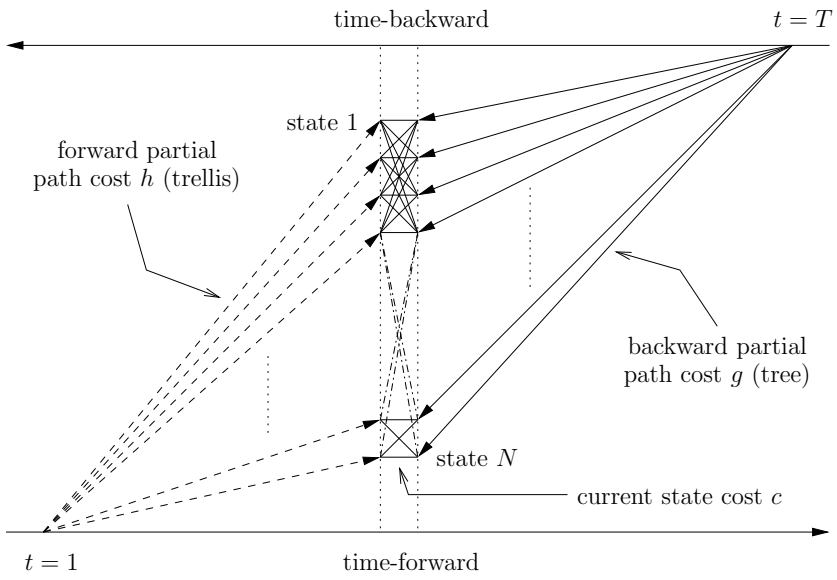

Figure 5.2: Merging forward and backward partial paths during the tree-trellis search. The total path cost is determined by the costs of the backward partial path, current node and forward partial path cost, i.e. $f = g + c + h$.

The overall backward tree search algorithm is given in Figure 5.3. The formal notation is based on the one used in (Rabiner, 1989) for the Viterbi search in order to maximize the level of abstraction, such that the algorithm itself and the data structures it requires are independent of a specific programming language. Unlike the presentation in (Soong and Huang, 1991), it is tried to minimize the use of natural language for expressing the different steps of the algorithm. At first glance, this decision may seem to make the algorithm rather difficult to read and understand. Nevertheless, it is presented in a way that allows for a direct implementation. Since the probabilities are multiplied during the search, the limitation of the representation of floating point numbers on computer hardware causes a serious problem. For long sequences, the computation of the probabilities will almost always end in numerical underflow because the values get so small that they cannot be realized any more. A common way of dealing with this problem is the use of *logprob* values (cf. (Manning and Schütze, 2000)). Instead of multiplying the probabilities, the logarithm of each probability is taken and the values are summed. This measure is also used in the implementation of the n-best HMM supertagger. The backward tree search algorithm is prefixed with line numbers and in the upcoming paragraphs, a detailed explanation to all steps will be given. Comments within the algorithm are written in a C-like

It is only shown for the sake of completeness.

manner between "/*" and "*/".

The algorithm uses a *stack* to manage the nodes of the search tree (cf. (Knuth, 1997)). Since the f-values are the exact path costs of the state sequences, it is possible to limit the size of the stack to n and nevertheless have the certainty to find the top n hypotheses. This approach guarantees a computationally feasible search in terms of memory requirements. In general, nodes are inserted according to their cost which is defined by the probability of the optimal state sequence passing through that specific state. So the node with highest priority is always extracted next.[3] This is a property required by the best-first search framework. A node ν represents a state of the HMM in the search tree and consists of seven elements:

- For each node, a rank-ordered list of predecessor states is stored in the variable $\gamma(\nu, i), 1 \leq i \leq N$. Basically, this is a means of associating the states from the ψ-table of the modified forward Viterbi step with a specific node ν of the search tree.
- The total path costs of the optimal state sequence passing through node ν and having one of the predecessor states from the γ-table are stored in $f(\nu, i), 1 \leq i \leq N$, and are used to sort the items of the stack such that the next best node is always expanded first (cf. best-first search).
- Each node holds its backward partial path cost g, i.e. the actual cost of the state sequence found so far.
- A special variable $\iota(\nu)$ keeps track of the index of the next best predecessor state not yet expanded.
- The time frame of the current node is stored in $\tau(\nu)$.
- The state that the current node is associated with is given by $\omega(\nu)$.
- Finally, the predecessor node[4] of the current node ν is stored in $\rho(\nu)$. When the search reaches a goal state, this information is used to backtrack through all nodes on the path of this solution in order to build up the next best state sequence.

[3] Therefore, the stack really shows more characteristics of a priority queue, but it is referred to as a stack in (Soong and Huang, 1991).

[4] Do not confuse the two terms "predecessor state" and "predecessor node". The former denotes a state whose time frame is smaller than the time frame of the current state, whereas the latter represents a node of the stack that has already been expanded in a previous step, i.e. whose time frame lies in the future. This mix-up is basically due to the fact that the tree search is *time-backward* while simultaneously using structures obtained in the *time-forward* trellis step. The "predecessor state" ought really be called a "successor state" since it is expanded next, but this would conflict with the terminology used in the Viterbi algorithm. Anyhow, you would probably be confused either way.

0: /* initialize stack σ with start node ν_{start} */
1: $\gamma(\nu_{\text{start}}, i) = \psi'_T(i) \qquad 1 \leq i \leq N$
2: $f(\nu_{\text{start}}, i) = \delta_T(\gamma(\nu_{\text{start}}, i)) \qquad 1 \leq i \leq N$
3: $g(\nu_{\text{start}}) = 0$
4: $\iota(\nu_{\text{start}}) = 1$
5: $\tau(\nu_{\text{start}}) = T + 1$
6: $\omega(\nu_{\text{start}}) = \varnothing$
7: $\rho(\nu_{\text{start}}) = \text{NULL}$
8: insert$(\sigma, \nu_{\text{start}}, f(\nu_{\text{start}}, \iota(\nu_{\text{start}})))$
9: $n_{\text{found}} = 0$
10: $\mathbb{H}(i) = \text{NULL} \qquad 1 \leq i \leq n \qquad$ /* create empty hypotheses vector */
11: /* search iteratively for hypotheses */
12: **while not** empty(σ) **do begin**
13: $\quad \nu_{\text{current}} =$ extract-top(σ)
14: $\quad$ create new node ν_{next}
15: $\quad s = \gamma(\nu_{\text{current}}, \iota(\nu_{\text{current}})) \qquad$ /* retrieve next best state for expansion */
16: $\quad \tau(\nu_{\text{next}}) = \tau(\nu_{\text{current}}) - 1$
17: $\quad$ **if** $\iota(\nu_{\text{current}}) \leq N$ **then begin** $\qquad$ /* reinsert rest of ν_{current} in stack */
18: $\quad\quad \iota(\nu_{\text{current}}) = \iota(\nu_{\text{current}}) + 1$
19: $\quad\quad$ insert$(\sigma, \nu_{\text{current}}, f(\nu_{\text{current}}, \iota(\nu_{\text{current}})))$
20: $\quad$ **end**
21: $\quad$ **if** s is a goal state, i.e. $\tau(\nu_{\text{next}}) \leq 1$ **then begin**
22: $\quad\quad$ /* obtain the hypothesis by backtracking through the stored pointers */
23: $\quad\quad$ create new hypothesis H and add s to H
24: $\quad\quad \nu = \rho(\nu_{\text{current}})$
25: $\quad\quad$ **while** $\nu \neq \text{NULL}$ **do**
26: $\quad\quad\quad$ add $\omega(\nu)$ to H and set $\nu = \rho(\nu)$
27: $\quad\quad n_{\text{found}} = n_{\text{found}} + 1$
28: $\quad\quad \mathbb{H}(n_{\text{found}}) = H$
29: $\quad\quad$ **if** $n_{\text{found}} \geq n$ **then** $\qquad$ /* found n best hypotheses */
30: $\quad\quad\quad$ **return** $\mathbb{H}$
31: $\quad$ **end else begin** $\qquad$ /* expand new successor node */
32: $\quad\quad \gamma(\nu_{\text{next}}, i) = \psi_t(s, i) \qquad 1 \leq i \leq N$
33: $\quad\quad$ let $j = \omega(\nu_{\text{current}})$, $t = \tau(\nu_{\text{current}})$ and $t' = \tau(\nu_{\text{next}})$
34: $\quad\quad g(\nu_{\text{next}}) = g(\nu_{\text{current}}) + a_{sj} b_j(o_t)$
35: $\quad\quad f(\nu_{\text{next}}, i) = g(\nu_{\text{next}}) + \phi_{t'}(s, \gamma(\nu_{\text{next}}, i)) \qquad 1 \leq i \leq N$
36: $\quad\quad \omega(\nu_{\text{next}}) = s$
37: $\quad\quad \iota(\nu_{\text{next}}) = 1$
38: $\quad\quad \rho(\nu_{\text{next}}) = \nu_{\text{current}}$
39: $\quad\quad$ insert$(\sigma, \nu_{\text{next}}, f(\nu_{\text{next}}, \iota(\nu_{\text{next}})))$
40: $\quad$ **end**
41: **end**
42: **return** $\mathbb{H}$ /* there are less than n hypotheses */

Figure 5.3: The backward A* tree search for finding the n best hypotheses.

Lines 1–10 in Figure 5.3 deal with the initialization of the data structures. All states of the last time frame T are added to the predecessor list γ and are sorted according to the probabilities stored in δ_T. The time frame of the starting node is $T+1$ and $\omega(\nu_{\text{start}}) = \varnothing$ denotes a pseudo-state which is added to the HMM since the "predecessor" states of the last time frame of an HMM have no successor. This means that in the graphical notation of an HMM, all final states need additional transitions to a single ending state where the search starts. The total path cost at this point is $f = h$, i.e. the backward path cost for ν_{start} is zero and the heuristic is the score obtained in the forward Viterbi step and stored in δ_T. The index of the next best state is set to 1. Thus, the first state that is going to be expanded is the best state q_T^* found by the Viterbi algorithm. The rest of the initialization sets the backpointer to NULL (since there are no predecessor nodes yet) and the counter n_{found} which keeps track of how many hypotheses have been found so far to zero. An empty hypotheses vector that will hold all hypotheses is created and the start node is put on the stack with priority $f(\nu_{\text{start}}, 1) = p^*$.

Now, the iterative search process starts (lines 12–41). In each step, the currently best node ν_{current} with highest score is extracted from the stack σ. A new node ν_{next} is created which will hold the next best state expansion s, whose index is given by $\iota(\nu_{\text{current}})$, from the list of the predecessor states of ν_{current}. In lines 17–20, the current node is reinserted into the stack according to the score $f(\nu_{\text{current}}, \iota(\nu_{\text{current}}))$ of the state which is coming after s, i.e. which is pointed to by $\iota(\nu_{\text{current}})$ after having increased it by one. The section in lines 21–30 handles the case if s, the state that is going to be associated with the node ν_{next}, is actually a goal node. This is the case if the time frame of the next node has reached the beginning of the observation sequence, i.e. $\tau(\nu_{\text{next}}) = 1$. The hypothesis is obtained by backtracking through the pointers stored in ρ. As soon as the search finds the n best hypotheses, it stops and returns the results (line 30). The final expansion of the next node is shown in lines 31–40. It gathers all predecessors of the best state expansion s. Then, the backward partial path cost g and the total path costs f are computed from the cached results of the previous nodes and the values stored in the forward trellis step. First, the backward partial path cost of ν_{next} is determined (line 35) by summing the backward partial path cost of the previous node ν_{current} and the current cost of the expansion from node $\omega(\nu_{\text{current}})$ to node s, the former denoting the state associated with the previous node. The backpointer to ν_{current}, the node where we come from, is remembered by $\rho(\nu_{\text{next}})$ and the index variable ι is initialized to 1. Finally, the new node is added to the stack according to its score of the next best expansion.

The search stops if σ is empty, i.e. it does not hold nodes for further expansion any more, or if it finds the n best hypotheses. In the first case, there probably exist less than n hypotheses which are returned in line 42. Note that the algorithm can be easily changed from A* to greedy search by adjusting the line where the computation of the total path score takes place. Instead of adding the backward partial path cost

g to the heuristic h (which is represented by ϕ_t), only the latter is used for f. So line 35 has to be changed as follows: $f(\nu_{\text{next}}, i) = \phi_{t'}(s, \gamma(\nu_{\text{next}}, i))$. As stated in Section 4.5.2, the resulting search is then neither optimal nor complete, but tends to find good solutions very fast.

5.2 System's components

In the previous section, the basic concepts that are going to be used in the overall n-best HMM supertagger for the special domain of ambiguous typing have been introduced. The starting point is the ambiguously coded word sequence typed with a reduced keyboard as introduced in Chapter 3. The simple approach orders the matching words of the candidate list for each code according to the words' frequencies that are obtained from a large corpus. The first step to improve this approach is to use information from the context, i.e. base the disambiguation on n-grams as described in Section 2.2. Since the sentence we want to type is "hidden" behind the sequence of codes, it is suitable to use HMMs as the primary modeling paradigm. The underlying statistical language model of the procedures in this thesis is a trigram Hidden Markov Model. So for each code in the sentence, the disambiguation process is based on the history of the last two items. The HMM that represents the language model is directly trained on an annotated training corpus. The trained HMM is used as a knowledge source while disambiguating the code sequences of the evaluation material, i.e. the test sentences. Every code generates a list of words and every word has several supertags associated with it. A general supertagger could be used to find the most likely supertag sequence for the sentence and use this information to reorder the candidate list such that the most likely words (which are the lexical anchors of the supertags) appear at the top. Due to the ambiguous coding, the number of supertags for a code (which corresponds to the supertags of all word expansions of a code) is so large that the best supertag sequence is not sufficient to improve the results significantly. Therefore, the supertagger is enhanced with the n-best tree-trellis algorithm from Section 5.1 in order to produce more than one hypothesis. At this point, the code sequence of each sentence is associated with a list of the n best supertag sequences found by the supertagger. In a final step, a lightweight dependency analysis is used as an additional knowledge source that once again reorders the candidates. The supertag hypotheses that span a large portion of the sentence and seem most "consistent" are moved to the top. The resulting hypotheses list is used to rearrange the list of matching words accordingly, since each supertag is associated with a lexical anchor.

The new components of the n-best supertagger are partly derived from existing classes and the connections between these classes are shown in Figure A.2 on page 103. The interface NBestSuperTagger extends the inherited SuperTagger by the

method tagNBest which applies the n-best tree-trellis algorithm to a sentence and returns a vector that holds all hypotheses. The main class that performs the evaluation of a test corpus is CodedNBestSuperTaggingEvaluator. It uses an instance of a CodedNBestTrigramSuperTagger where the overall n-best tree-trellis search is finally implemented. We will turn to this in Section 5.2.2. But first, the next section gives a detailed description of how the ambiguous codings are added to the supertagging framework that serves as a starting point.

5.2.1 Coping with ambiguity

The ability of coping with ambiguous codes is added in the class CodedTrigramDataManager which is a subclass of TrigramDataManager. The trigram data manager processes all requests to the language model and has to be "primed" with the symbol emission probabilities $P(w_i|t_i)$ according to the current observation sequence (i.e. the words of the sentence) and the associated supertags that are stored in the trained language model. Instead, when typing with an ambiguous keyboard, the final word sequence is not known and all words of the lexicon for a given code have to be considered. The class CodedLexicon provides a keyboard specification and the data structures that map the code sequences of the ambiguous keyboard to a list of possible words matching that particular code. For the German keyboard layout (cf. Section 3.4.1), the code "2133", for example, has 27 matches (the first five being *kann*, *habe*, *Hand*, *fand* and *frei*) which are sorted according to their frequency based on the CELEX lexical database (cf. Section 6.1.2). The CodedTrigramDataManager takes each of these words and primes the data manager with all possible supertags stored in the trained model for that particular word. Every word usually has several supertags, since the lexical items of an LTAG are almost always associated with several elementary structures that encode the various local dependencies of each word. And since every code expands to several matching words, the result is a set of supertag sets that form a trellis (cf. detailed view in Figure 5.4). This trellis is the basis for the tree-trellis search that finds the n best supertag hypotheses for a given sentence. Figure 5.4 shows the different expansion steps for the sentence *ich habe ein kleines Problem* ("*I have a slight problem*").

After typing the words of a sentence with the ambiguous keyboard, the code sequence is expanded and the candidate list is obtained according to the CELEX lexicon. After that, the possible supertags are looked up in the trained language model, i.e. all supertags that occurred in the training corpus with its corresponding lexical anchor are primed for the n-best tree-trellis search. The hypotheses that are returned by the search are then used to reorder the candidate lists. The effect is that likely words of the trained language model will move to the top of the match lists and improve the overall accuracy of the system. An additional LDA on the hypotheses determines the ones with maximum coverage and uses this information to make final adjustments

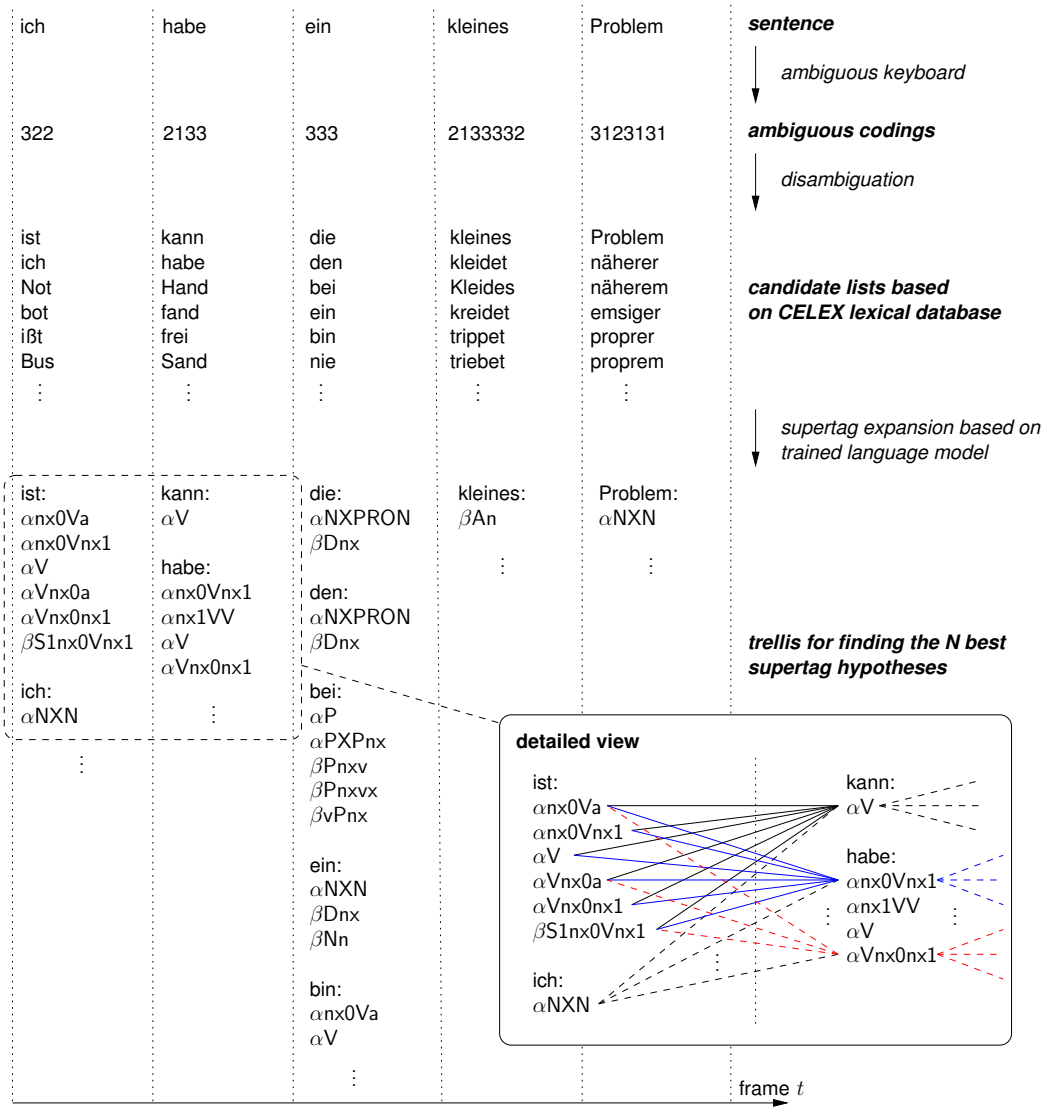

Figure 5.4: Coping with ambiguous words: disambiguation of coded words and the corresponding supertag expansion.

to the match lists. These two phases could be merged to one but they are carried out separately in order to measure how much impact each method has on the rank accuracy. A detailed view on this issue is given in Section 5.2.3.

Now, coping with unknown words in ambiguous typing is a more complicated problem. If the word is not in the dictionary, it has to be disambiguated letter by letter for all the keys of the code (cf. Section 3.3.2). Since the primary goal was not to simulate a specific keyboard but to evaluate whole sentences with the n-best supertagging framework, the dictionary was patched by adding the unknown words with a zero-frequency (cf. Section 6.1.2) and thus contained all words of the corpus.

5.2.2 Implementation of the tree-trellis search

The tree-trellis search is applied in the class CodedNBestTrigramSuperTagger. It comprises two steps, namely the modified forward Viterbi algorithm (cf. Figure 5.1) and the n-best backward tree search (cf. Figure 5.3). The stack is realized in the class NBestStack which is shown in Figure A.3 on page 104. The nodes of the stack are stored in a rank ordered list since the nodes with highest priority, i.e. the states of the HMM with highest logprob, are always expanded next. As stated in Section 2.1.2, the runtime of the forward Viterbi step is in $O(N^2T)$, where N is the number of states and T is the length of the input sequence. The modification that stores all predecessors instead of the best one increases the runtime slightly since it sorts the predecessor lists according to the scores. The initial Viterbi algorithm has to look for the maximum (best) predecessor, which is in $O(n)$ for n items, whereas building the whole rank-ordered predecessor list needs $O(n \log n)$ time. The backward tree search is very efficient since it uses the optimal predecessor lists provided by the enhanced Viterbi algorithm. These lists guarantee that the next node for expansion also is the current overall best one. Since the search is limited to n best hypotheses, the main iterative loop of the algorithm (cf. Figure 5.3) has a time complexity of $O(n \log nN^T)$. The loop itself has $O(nN^T)$ iterations and within the loop, all actions have constant time except the insertion of nodes into the stack which needs $O(\log n)$ time.[5] So the runtime of the overall algorithm is exponential because of the A* search (N is the branching factor, T is the depth of solution). Nevertheless, since the heuristic is *optimal* and the performance of A* heavily relies on its heuristic, solutions are found much faster in general. The larger drawback is the amount of memory needed. Since we made the algorithm faster by pre-computing the predecessor lists during the Viterbi step, it is clear that the gain in runtime is bought by "sacrificing" additional memory. The initial Viterbi algorithm needs $O(N^2 + NM + NT)$ space for the state transition probabilities, observation symbol probabilities and the backpointers for

[5] Since the nodes are already ordered according to their probability, we only have to find the correct insertion point for a new node which can be found by binary search (see e.g. in (Gonnet and Baeza-Yates, 1991)), thus $O(\log n)$ for n being the maximum stack size.

	Viterbi	modified Viterbi	backward A* search
time	N^2T	$(N \log n)^2 T$	nN^T
space	$N(N+M+T)$	$N(N+M+nT)$	$N^2+NM+nNT$

Table 5.1: Worst-case space and time complexity of all components of the n-best supertagger. N is the number of states (for S supertags, $N = S^2$ because we consider pairs of supertags as states in the trigram model), M is the number of output symbols (words), T is the length of the input sequence (sentence) and n is the maximum number of supertag hypotheses that are determined.

each position in the trellis, respectively. With introducing the n-best predecessor lists instead of the single best predecessor, the complexity of the last term increases to $O(nNT)$. But this worst-case is not likely to happen because only the relevant states are considered, i.e. those with a non-zero probability. The memory usage of the A* search only increases by a constant factor since it uses the data structures from the modified Viterbi algorithm. The stack of the n-best search can be limited to n nodes due to the optimality of the heuristic. So the overall space complexity does not increase significantly when compared to the one of the modified Viterbi. All time and space complexities are summarized in Table 5.1.

5.2.3 Adjusting the candidate lists

After the n-best tree-trellis search, the algorithm provides a list of supertag hypotheses. These sequences of supertags are the top n readings of the ambiguous code sequence according to the trained language model. They serve as a basis to rearrange the candidate lists obtained from the CELEX ordering. We will refer to the basic approach of reordering the unigram-frequency candidate lists by using the n best supertag hypotheses as *match list boosting*. So let $\mathbb{H}(i)$, $1 \leq i \leq n$, hold the n best hypotheses returned by the tree-trellis search (cf. Figure 5.3) and $\mathbb{M}(j)$, $1 \leq j \leq T$, be the initial candidate list for the j^{th} word of the sentence. Now, all lexical anchors of the supertags of a single hypothesis are boosted to the top of the match list in reverse order, i.e. starting with the last hypothesis n and ending with the best hypothesis stored in $\mathbb{H}(1)$.[6] The algorithm for adjusting the candidate lists by boosting is given in Figure 5.5. It can be optimized if the list of all supertags for a word position is reduced by discarding all duplicates and move only the highest ranked occurrence to the top of the candidate list.

So far, the improvement in accuracy of the whole system is solely based on the

[6] Alternatively, one could take the hypotheses as they are, reduce multiple lexical anchors per word position (frame) to one (since a word can have several supertags) and add the rest of the words from the CELEX lexicon that are not yet covered by the list.

```
 0: i = n        /* n is the last hypothesis */
 1: while i > 0 do begin
 2:         H = ℍ(i)        /* retrieve current hypothesis */
 3:         j = 1
 4:         /* boost all supertags of H in the corresponding match list */
 5:         while j ≤ T do begin
 6:                 boost(H(j), 𝕄(j))
 7:                 j = j + 1
 8:         end
 9:         i = i − 1
10: end
```

Figure 5.5: The algorithm for *match list boosting*. $\mathbb{H}(i)$ holds the i^{th} of the top n supertag hypotheses, $\mathbb{M}(j)$ is the match list with the candidates that are expanded from the j^{th} code in the sentence of length T. $H(j)$ denotes the lexical anchor (word) of the current supertag that is moved to the top in match list $\mathbb{M}(j)$. The words are boosted in reverse order such that the best hypothesis is applied last, resulting in the overall best word appearing at the first position.

trigram language model. Although the n-best supertag sequences have the highest scores out of all possible hypotheses, this does not mean that *all* of these hypotheses make perfect sense. And it is quite likely that some lower ranked hypothesis actually has more correct lexical anchors when compared to the target sentence than the overall best one. So it is not known exactly which of the n-best hypotheses gives the best result if applied last. The effect is that a correct word that is boosted to the top might be displaced by a wrong word that comes from a higher ranked hypothesis. This is where a *lightweight dependency analyzer* (LDA) helps. After boosting all hypotheses returned by the n-best search, the ones that pass an additional LDA filter step are used again by the algorithm in Figure 5.5. Figure 5.6 shows this step for the setting from Figure 5.4. Section 4.3.1 introduced the LDA and showed how it can provide a shallow parse of a sentence. This information can be used to further determine the quality of the hypotheses. The more "consistent" the shallow parse, the more agreement it might have with the original sentence. By applying an LDA, we also have to connect to the LTAG since it yields the necessary dependency slots for each supertag. The components to accomplish this task originate from the work done within the project INTEGENINE (Harbusch et al., 1998; Woch and Widmann, 1999; Harbusch and Woch, 2000) and have been enhanced for the initial supertagger in (Bäcker, 2002). The class LightweightDependencyAnalysis connects to components

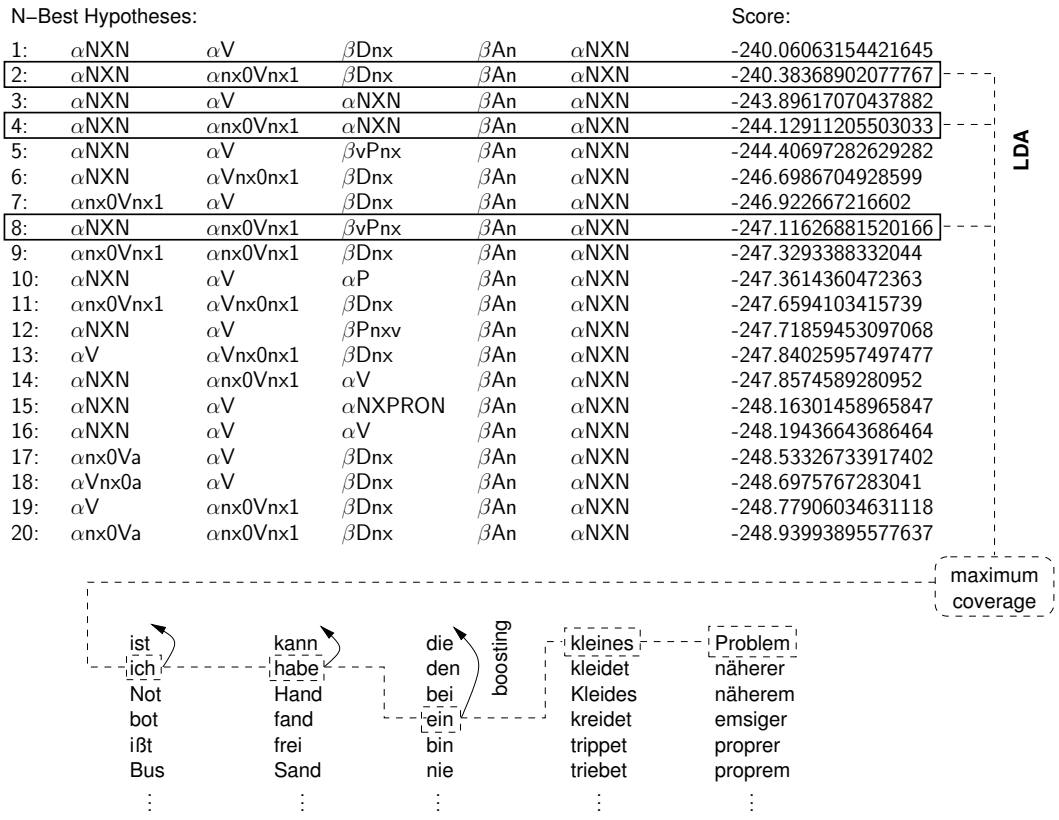

Figure 5.6: Boosting LDA matches with maximum coverage. In this (idealistic) case, the lexical anchors of the second hypothesis, which is the overall correct one, are boosted last, thus yielding that the right words of the target sentence *Ich habe ein kleines Problem* are being moved to the first position within the candidate lists.

of the class STAGSchemaTag and XTAGTree which provide the required dependency slots. The overall collaboration is shown in Figure A.4 on page 105. The result of the LDA is an LDASentence which holds the necessary information needed to "measure the quality" of the original hypothesis. Basically, it provides the codings introduced in Section 4.8 ($+$, $-$, $\bullet$ and $*$). Two criteria are used to determine the usefulness of an analyzed hypothesis:

- *dependency coverage*: only the hypotheses are boosted that have the maximum number of covered elements, i.e. where as many of the dependency slots are filled to the left or right as possible.
- *top level supertags*: only the hypotheses with a certain number of elements on the uppermost level are considered for boosting. This value can be seen as the number of "consistent islands" that has been found by the algorithm.

In Figure 5.6, the three marked hypotheses have a dependency coverage of 5, i.e. all supertags have fulfilled dependency slots, whereas the other hypotheses have coverages less than 5. The experiments with the test corpus have shown that a combination of both criteria, namely boosting the hypotheses with maximum coverage *and* maximum top level, results in a continuous small improvement to the overall accuracy of the system. The dependency coverage criterion has the largest impact on system accuracy. The best hypotheses, i.e. those that match the target sentence as close as possible, almost always have the highest dependency coverages. The number of top level supertags knows to vary. Experiments were only conducted on minimum and maximum criteria, but it seems that a medium number of top level supertags looks more promising since a manual comparison of the hypotheses with the annotated target sentence showed that most agreement lied within hypotheses with maximum dependency coverage and a medium number of top level supertags. This observation should be further investigated in future work. Appendix C lists the output of the system when applied to the setting from Figures 5.4 and 5.6 when using the top 5 supertag hypotheses.

The next chapter gives a detailed overview on the LTAG and its lexicon, the evaluation corpus and the results achieved with the n-best supertagger as presented in the previous sections.

6 Results and Discussion

For an evaluation of the techniques presented in the previous chapter, the ambiguous typing of a sample text is simulated and processed with the n-best supertagger. As performance criteria, the *accuracy* and the *average rank* of the correct word are compared to the values obtained from the simple unigram-based approach that was presented in Section 3.4.2. For this purpose, a lexicalized tree adjoining grammar is needed because of the lightweight dependency analysis performed in the last step of the n-best approach. The trigram HMM is directly trained on a corpus that is annotated with supertags. For training and testing, the corpus developed in (Bäcker, 2002) is used. It contains sentences from German news group articles in the domain of general support for hardware problems concerning monitors and hard disks. The advantage of the LTAG which was also developed in (Bäcker, 2002) for that domain is its computational efficiency. Since it is rather small, containing only 127 elementary trees, this directly impacts on the size of the trained HMM and the runtime of the LDA. Therefore, it was possible to run the n-best supertagger for up to 2,000 hypotheses in an acceptable amount of time. The basic idea behind this kind of corpus is its domain-specificity. As mentioned in Section 3.3.2, the performance of the ambiguous keyboard can be improved when combining the common dictionary with a domain-specific or even topic-specific knowledge source.

Section 6.1 introduces the evaluation corpus and the lexicalized tree adjoining grammar that are used within this work. The results obtained on the baseline, i.e. when using the initial frequency-ordered CELEX candidates, are presented in Section 6.2. Section 6.3 deals with the improved results from the n-best supertagger when tested on the evaluation corpus. The last section discusses the results.

6.1 Evaluation corpus

The evaluation of the n-best supertagger is based on a collection of sentences that were extracted from German news groups and discussion forums[1] in the domain of general support for computer hardware, especially dealing with monitor and hard disk problems in this particular case. At first glance, this might be an inappropriate domain for testing AAC techniques since the corpus was primarily developed for an user-initiative dialog system (Harbusch et al., 2001). Nevertheless, some of the

[1] `de.comp.hardware.{misc|graphik|laufwerke.festplatten}`, `http://www.heise.de/foren`

Topic-specific problem descriptions	
Jetzt habe ich mit fdisk zwei Partitionen eingerichtet.	Now I have created two partitions with fdisk.
Meine neue Festplatte wird vom CMOS Setup nicht automatisch erkannt.	My new hard drive is not being recognized automatically by the CMOS setup.
Mein Monitor flackert plötzlich.	My monitor flickers suddenly.
Wenn ich meine Boxen einschalte, flimmert mein Monitor.	If I turn on my speakers, the monitor flickers.
General expressions	
Was kann ich machen.	What can I do.
Nichts hat geholfen.	Nothing helped.
Ich habe ein Problem.	I have a problem.
Klappt wunderbar.	Works perfectly.

Table 6.1: Some example sentences of the corpus used for evaluation in this thesis. It consists of topic-specific problem descriptions as well as general (i.e. unspecific) expressions.

sentences represent colloquial expressions as they often arise in spontaneous communication situations where the user of the ambiguous keyboard wants to express simple ideas as fast as possible. The second advantage is the specific domain that is dealt with (cf. Section 3.3.2). The exploration of different domains or topics is left out due to the time expenditure that would be involved in parsing and annotating other text corpora.

The corpus which is used in this thesis comprises 250 sentences with a total of 1,964 word tokens that mainly deal with monitor and hard disk problems. Table 6.1 shows some detailed problem descriptions as well as general expressions of the text collection. The corpus is manually annotated with supertags from the lexicalized tree adjoining grammar (cf. Section 6.1.1) that was developed in (Bäcker, 2002). For example, each word of the sentence

```
das             betaDnx
bild            alphaNXN
meines          betaDnx
monitors        betanxN
flimmert        alphanx0V
%%%END_OF_SENTENCE
```

has the corresponding supertag on the right that is obtained by parsing the sentence with the XTAG parser. The corpus is divided into 90% for training (225 sentences)

and 10% for testing (25 sentences). Initially, the text material consisted of separate topic-specific sets for hard disks and monitors, respectively. Therefore, each topic is divided separately and merged afterwards in order to guarantee that the sentences of the two problem classes are equally distributed on the training and test set. This prevents the possibility of primarily training on the hard disk domain while testing on the monitor class or vice versa if a random division had been made. In addition, a cross-validation test run on the whole corpus is applied. Since the test sets usually comprise 10% of the available data (250 sentences), the cross-validation results in 10 distinct divisions of the training and test set, each containing 225 sentences for training and 25 for testing. So, in every run, $1/10^{\text{th}}$ of the data is held out for testing, while the rest is used for training. This procedure is repeated until all possible blocks of 25 sentences have been evaluated.

6.1.1 German LTAG

The lexicalized tree adjoining grammar presented in (Bäcker, 2002) was developed for the special domain of a user help-desk in a dialog system. Therefore, it is not representing a wide coverage grammar for the German language but, since it was used to parse and annotate the evaluation corpus, its expressibility is sufficient for this task. The grammar contains 127 elementary trees, divided into 58 initial and 69 auxiliary trees. In comparison, the English general purpose grammar developed by (XTAG Research Group, 2001) has over 1,000 trees. A grammar of that size also needs a larger training corpus in order to achieve reliable results when estimating trigram probabilities based on supertags (sparseness of data). Since this would have required a large amount of additional time, it was decided to use the already available German KoHDaS-ST grammar for the evaluation. The lexicon contains 907 lexical entries and covers the vocabulary from the evaluation corpus. The format of the grammar is that of (Doran et al., 1994; XTAG Research Group, 2001) and can hence be used with the available XTAG parser to annotate the words with the corresponding supertags.

6.1.2 Lexicon

The primary lexicon for the ambiguous keyboard is the CELEX lexical database (Baayen et al., 1995). It consists of separate lexicons for the Dutch, English and German language. As basis, a modified version of the German lexicon is used. The modification is due to the German spelling reform that was decided on in 1996 and came into effect in 1998. In this new German orthography (IDS, 1996), changes in rules for spelling, hyphenation, capitalization and punctuation have been made. Therefore, the lexicon contains all new entries and additionally the old alternatives, since the old orthography is valid until the end of 2005. The LTAG lexicon and the

total frequency of tokens/types	4,013,854/317,198
cumulative accuracy for rank $\leq 1, \ldots, 5$ in %	74.14/88.33/93.34/95.82/97.16
expectation/standard deviation of rank	1.59/1.60
total number of ambiguous codes	167,430
number of match lists with length $= 1, \ldots, 5$	114,325/31,644/6,875/4,707/2,242
maximum match list length	64
expectation/standard deviation of list lengths	1.89/2.81

Table 6.2: The modified German lexicon used for building the candidate lists.

supertagger by (Bäcker, 2002) do not distinguish capitalized from non-capitalized word forms. So the modified CELEX lexicon was adapted to this circumstance such that all word types with capitals are written in lower case. If the word form with lower case letter already exists, the frequencies are added in order to maintain the overall frequency mass. In general, the lexical entries of the CELEX lexicon comprise all derived and inflected word forms of the corpus that the frequency estimation is based on. For the German part, the frequencies are obtained from the Mannheim corpus (6 million words) provided by the IDS (Institute for German Language). A problem with ambiguous keyboards is typing unknown words, i.e. words that are not contained in the lexicon. Usually, an additional disambiguation step has to be carried out. For all codes, the right letter has to be determined separately. Since the main attention of this work is turned on the sentence-based evaluation of ambiguous typing, the additional disambiguation step is not simulated and all unknown words from the test set are added to the main lexicon with a zero frequency instead.[2] In addition, unknown words would be stored in a special user dictionary after being disambiguated for the first time, so the cost of this step can be neglected.

Table 6.2 gives a summary on the modified German lexicon based on the CELEX database and the LTAG lexicon. The statistics show the total number of word tokens and types, i.e. the individual occurrences of all words and the distinguishable word forms, respectively. The cumulative accuracy of a rank i denotes the percentage of tokens that appear at i^{th} or less position in the candidate list. So for the whole lexicon, approx. 74% of all words appear at the first place of the match list and there are less than 3% of words that appear after the 5^{th} position. The high value for rank 1 is due to the fact that over 68% of the candidate lists actually have only one entry and the most frequent words appear at the top of the match lists.

If a closer look is taken on the distribution of ranks and their corresponding frequencies, one notices that the distribution is not a normal (Gaussian) distribution, but rather a hyperbolic distribution. This is due to the fact that the frequency f of a word is inversely proportional to its rank r in the whole list of lexical entries if

[2] In CELEX, all word forms that do not occur in the Mannheim corpus have a frequency of zero.

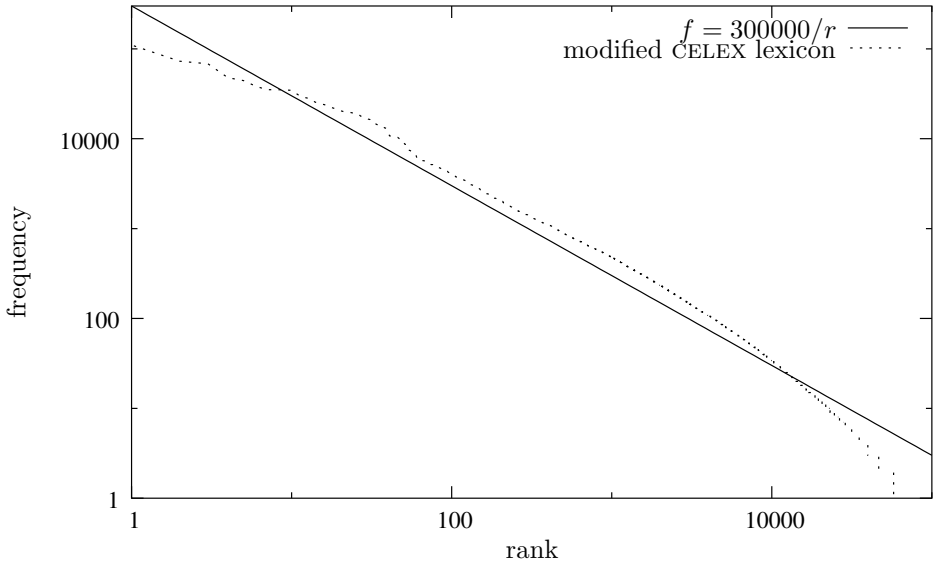

Figure 6.1: Zipf's law and the modified lexicon. The straight line is the graph from Equation 6.1 for $k = 300000$, the dashed graph denotes the values from the lexicon. The x- and y-axis (rank vs. frequency) use logarithmic scales.

they are sorted according to their frequency:

$$f \propto \frac{1}{r} \qquad \text{or} \qquad \exists k : f \cdot r = k. \tag{6.1}$$

This is also known as Zipf's law (see e.g. in (Manning and Schütze, 2000)). Therefore, natural languages tend to have few words which are used most commonly, some mid-frequent words and a large amount of words with low frequency. The consequence of the latter point is that we often have to work with sparse data. For the modified lexicon, the graph delineating this law is shown in Figure 6.1. Fortunately, the low-frequent words are also the longer words in a language, or to put it another way: the high-frequent words are usually very short.[3] And the longer the word, the higher the probability that its coding is unique and the candidate list therefore has only one entry. For the modified German lexicon, the average word length of words with a non-zero frequency is 5.9 characters, whereas the words with zero frequency have an average length of 11.5 letters.

[3] consider the top ten words of the English CELEX database: *the, a, of, and, to, in, that, it, I, is*

6.2 Baseline results

The baseline results are achieved with the simple unigram approach mentioned in Section 3.4.2. Here, the frequencies of the words that are stored in the lexicon order the candidate list in descending order, i.e. with highest frequency first. As evaluation criteria, the *accuracy of rank r* and the *average match position* is chosen. More formally, let

$$f_r(w|c) = \begin{cases} 1 & \text{if } w \in \text{matches}(c) \text{ and } \text{rank}(w) = r \\ 0 & \text{else} \end{cases} \tag{6.2}$$

be a binary function that returns 1 if a disambiguated target word w correctly occurs on the r^{th} position of the candidate list of its code c, which is given by matches(c). For a test corpus containing a total of N words, the accuracy of rank r for the given corpus can be computed as

$$\text{acc}(r) = \frac{\sum_w f_r(w|c)}{N}. \tag{6.3}$$

For a *cumulative accuracy*, i.e. where the target words appear within the first r ranks of the candidate lists, the single accuracy values are summed:

$$\text{cumacc}(r) = \sum_{i=1}^{r} \text{acc}(i). \tag{6.4}$$

The second evaluation measure is the average rank of words of the test corpus. It is simply computed by

$$\bar{r} = \frac{\sum_w \text{rank}(w)}{N}. \tag{6.5}$$

Several runs are applied to the test data. The initial evaluation is based on the training and test division that was used in (Bäcker, 2002) which will be further referred to as the *reference test set*. The supertagger that is introduced there is reported to achieve an accuracy of 78.3%. In this case, the accuracy denotes the percentage of correctly assigned supertags to the corresponding words, i.e. no ambiguous typing takes place. The test set contains sentences from all topics, i.e. monitor and hard disk problems as well as general expressions. If the original supertagger, which is only based on the Viterbi forward search (i.e. 1-best), is applied to the ambiguously typed corpus, the percentage of correct supertag to word assignments drops down to 17.5%. This is due to the large amount of possible supertag hypotheses that are expanded for every code. One possible way of improving the performance of the supertagger is to examine more than one hypothesis, namely the n best hypotheses in general (cf. Section 5.1).

reference test set evaluation, $\bar{r} = 3.02$					
	$r = 1$	$r = 2$	$r = 3$	$r = 4$	$r = 5$
acc(r) in %	50.26	28.04	5.29	7.41	1.59
cumacc(r) in %	50.26	78.30	83.59	91.00	92.59
cross-validation, $\bar{r} = 3.23$					
acc(r) in %	54.25	24.10	4.42	5.40	2.04
cumacc(r) in %	54.25	78.35	82.77	88.17	90.21

Table 6.3: The baseline results of ambiguously typing the corpus. The candidate lists are solely based on the unigram frequencies from the modified German lexicon. The results of the cross-validation represent the average of 10 runs.

The results for the baseline are shown in Table 6.3. As can be seen, the unigram approach places approx. 50% of the target words on the first position of the candidate lists. 92.6% of the words appear within the first 5 ranks. The rank expectation for the reference test set is 3, i.e. the user has to scroll two times on average before selecting the desired word. The values for the cross-validated results are similar. The average accuracy for rank 1 is slightly higher, but nevertheless nearly 10% of the words do not appear within the first 5 ranks which results in high selection costs. The next section presents the evaluation of the test corpus when processed with the n-best supertagger that was presented in the previous chapter.

6.3 N-best supertagging results

So far, the baseline results show that the simple approach based on unigrams places approx. 50% of the words on the first position, so the disambiguation costs do not accrue except for the selection step itself for half the words that are typed. On average, the user's target word can be found on the third position in the candidate list. Intuitively, a better language model which uses trigrams instead of unigrams should clearly yield an improvement. As with the baseline, the results presented in this section mainly constitute two parts, namely testing the reference data and running a cross-validated version that tests the randomized corpus in chunks of 25 sentences such that all 10 possible combinations of training and test divisions are evaluated. For the reference test set, the number of hypotheses during the evaluation reaches from 1 to 2000, whereas for the cross-validation, the maximum number of hypotheses was limited to 500 due to the time expenditure of the computation. Additionally, an upper bound evaluation is reported in Section 6.3.2 that shows what accuracy the n-best supertagger can achieve theoretically. Lastly (Sections 6.3.3 and 6.3.4), a small evaluation is performed that compares the different kinds of informed search

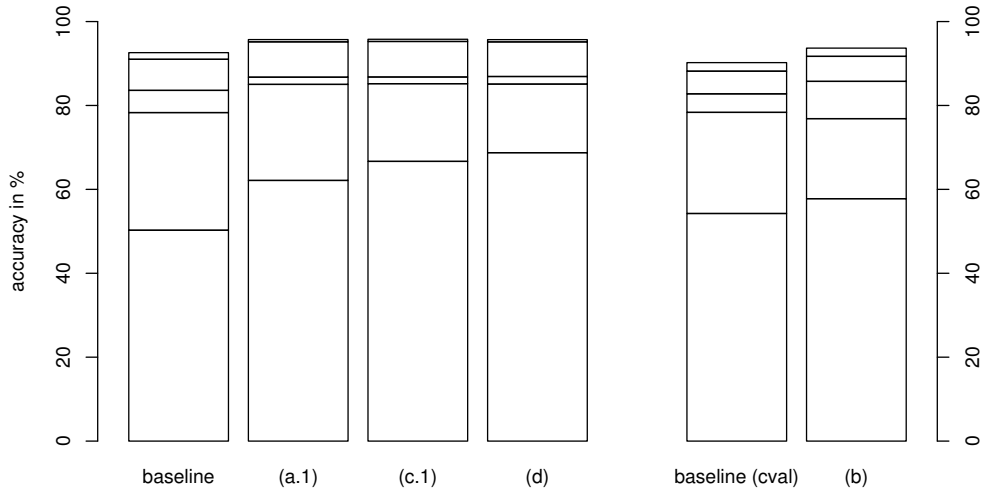

Figure 6.2: Graphical summary of the n-best supertagging results from Table 6.4. The bars show the accuracy of rank r for the reference test set (4 bars on the left) and the cross-validation (on the right). Each bar comprises 5 boxes, the lowest denoting acc(1), whereas the uppermost shows acc(5).

presented within the best-first framework (A* vs. greedy search) and an experimental setting is reported which examines the use of a word-based trigram language model.

6.3.1 Reference test set and cross-validation

The overall results are given in Table 6.4 and Figure 6.2. The table is divided in 4 parts and summarizes the results obtained with the n-best supertagger and LDA. The first part (a) shows the values computed for the reference test set. In (a.1), the average is listed for the full evaluation runs with hypothesis sizes ranging from 1 to 2000, whereas (a.2) shows the same for the first 500 hypotheses (for comparison to the cross-validated runs in (b)). When comparing the values to those in Table 6.3, a significant improvement for the reference test set is visible. The cumulative accuracy of rank 1 raises by approx. 12%, i.e. 62% of the target words are now placed on the top of the candidate lists. For the other ranks, the improvement is not as big as for rank 1, but there is still a significant increase. The average rank drops down to 2.16. The overall best run of this evaluation session is given in (c.1). The maximum occurred for the hypothesis size $n = 592$, i.e. the 592 best supertag sequence hypotheses for the ambiguously coded sentences are used for adjusting the candidate lists. This result also shows that the biggest variation takes place for rank 1. The changes in cumulative accuracy for ranks ≥ 2 are very small for larger values of n. The smoothed

reference test set evaluation					(a)
average for $n = 1, \ldots, 2000$, $\bar{r} = 2.16$					(a.1)
	$r = 1$	$r = 2$	$r = 3$	$r = 4$	$r = 5$
acc(r) in %	62.13	22.89	1.72	8.42	0.54
cumacc(r) in %	62.13	85.02	86.74	95.16	95.70
average for $n = 1, \ldots, 500$, $\bar{r} = 2.19$					(a.2)
acc(r) in %	61.54	23.05	2.02	8.29	0.58
cumacc(r) in %	61.54	84.59	86.61	94.90	95.48
cross-validation					(b)
average for $n = 1, \ldots, 500$ (10 runs), $\bar{r} = 2.64$					
acc(r) in %	57.78	19.08	8.88	5.96	1.97
cumacc(r) in %	57.78	76.86	85.74	91.70	93.67
best results					(c)
overall best for $n = 592$, $\bar{r} = 2.11$					(c.1)
acc(r) in %	66.67	18.52	1.59	8.47	0.53
cumacc(r) in %	66.67	85.19	86.78	95.25	95.78
best accuracy/time trade-off for $n = 250$, $\bar{r} = 2.16$					(c.2)
acc(r) in %	61.90	22.75	2.12	8.47	0.53
cumacc(r) in %	61.90	84.65	86.77	95.24	95.77
upper bound experiment					(d)
average for $n = 1, \ldots, 2000$, $\bar{r} = 2.09$					
acc(r) in %	68.74	16.35	1.81	8.25	0.54
cumacc(r) in %	68.74	85.09	86.90	95.15	95.69

Table 6.4: The improved results of ambiguously typing the corpus. All candidate lists except for the last part (upper bound experiment) are based on the trigram n-best supertagger with lightweight dependency analysis. The values of (a) represent the average of evaluating hypothesis sizes up to 2000 and 500 (the latter for comparison with the cross-validated results). The second table (b) delineates the cross-validation runs for $1 \leq n \leq 500$. Additionally, the best single run and the optimal run in terms of accuracy vs. time trade-off for the reference test set are given in (c). Lastly, the results of the upper bound experiment are summarized in (d) (cf. 6.3.2).

graphs in Figure 6.3 give an overview on the differences between the n-best approach and the baseline. Since the improvements with increasing size of the hypotheses lists lie within a few percent only, the course of the graphs is not as visible as with a more fine grained observation scale for the y-axis. Appendix B on pages 107*ff* lists all separate graphs for the various test runs.

When comparing the results of the reference test set to the cross-validated runs, the essential problem of the whole evaluation becomes clear. Due to the small size of the corpus that is used for this evaluation (250 sentences/1,964 words), there is an impeding influence of data sparseness. In fact, this behavior can be observed directly in the second graph of Figure 6.3 which shows the results for the cross-validation. The accuracy of rank 1 and 2 actually decreases with increasing values for n because the higher the number of hypotheses that are taken into consideration, the higher the overall number of unknown trigrams which are encountered in the hypotheses. For rank = 2, the n-best approach even performs worse than the simple unigram baseline for $n > 25$. Nevertheless, the total number of words placed within the first 5 ranks is still slightly higher than for the baseline. The overall improvement is not as good as for the reference test set. As can be seen in Table 6.3, the average rank for the cross-validated test set only decreases from 3.23 to 2.64. The graphs showing the average ranks are given in Figures 6.4 (reference test set) and 6.5 (cross-validation).

As can be seen in all graphs, enhancing the search from 1-best (Viterbi) to n-best has the largest effect for values of $n < 50$. After approx. 50 hypotheses, the results do not improve significantly, although there is some minimal increase in overall performance as can be seen in the more detailed graphs shown in Appendix B. In general, a hypothesis size of $n = 250$ (cf. Table 6.4 (c.2)) shows good results since the value for cumacc(5) does not increase any more for $n \geq 250$ (cf. Figure B.5 on page 110) and the computation is faster than the best case which occurred for $n = 592$. The evaluation of the reference test set needs approx. 3.37s, 10.58s and 21.62s for $n = 1$, 250 and 600, respectively.[4] So for $n = 250$, this gives an average of 423ms for a sentence. Even for $n = 600$, the computation lasts less than a second per sentence, still resulting in an acceptable delay for the user after she would have entered the last word. The adjustments of the match lists can therefore be performed in real-time for smaller values of n.

6.3.2 Upper bound

Another method of evaluating the n-best supertagger is the possibility to look at the target words of the sentences that are typed ambiguously and use only the hypotheses that match closest for adjusting the candidate lists (cf. results in Table 6.4 (d)). Clearly, this procedure is illegal for an objective evaluation since we are already look-

[4] running on an AMD Athlon XP 1600+ (1.4GHz)

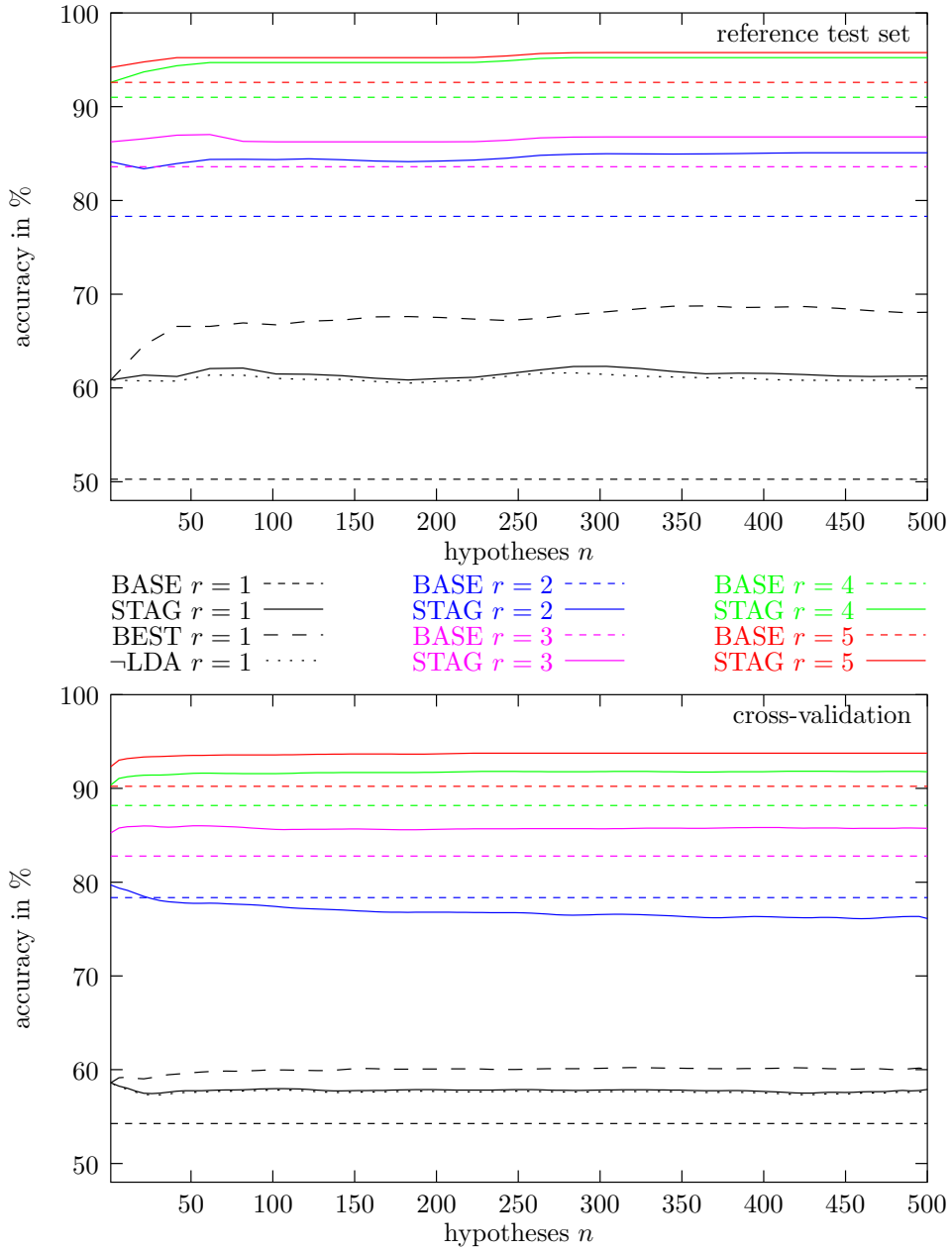

Figure 6.3: The cumulative accuracy of the reference test set and the averaged cross-validation (n-best supertagging results compared to the baseline). Dashed lines ("BASE") represent the cumulative accuracy of the baseline, whereas solid lines ("STAG") denote the improved results obtained with the n-best supertagger.

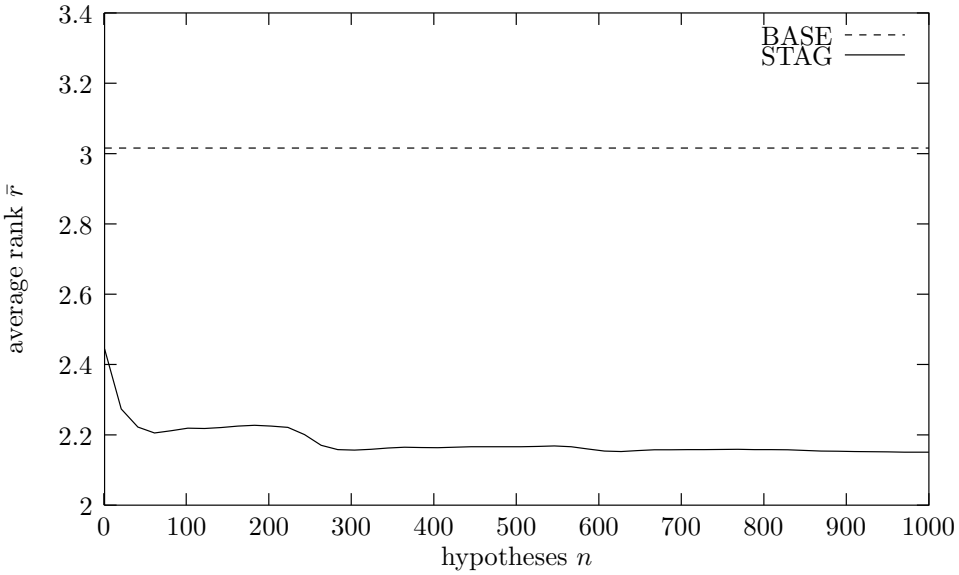

Figure 6.4: The average rank of the test run with the reference test set (n-best supertagger compared to the baseline). The dashed line is $\bar{r}$ for the baseline, the solid line shows the results of the n-best approach.

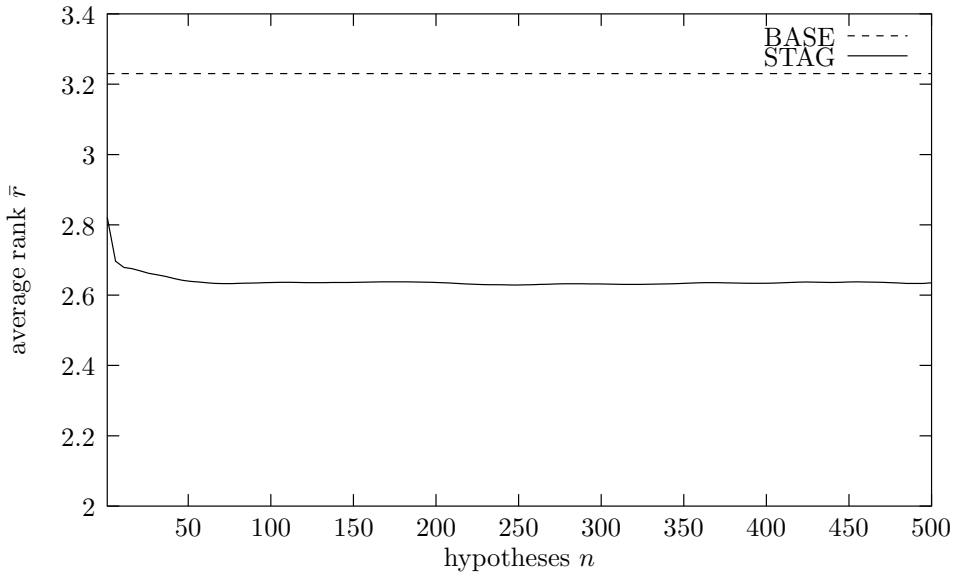

Figure 6.5: The average rank of the cross-validation runs. For better comparison, the scale of the y-axis is the same as for the graph in Figure 6.4.

ing at the desired result we want to achieve, but nevertheless it gives an upper bound of what accuracy the n-best supertagger can theoretically reach by just picking the most promising hypotheses. The detailed evaluation graphs are given in Section B.1. As can be seen, the accuracy between the two approaches differs only for lower ranks (Figures B.1–B.3, while for higher ranks (Figures B.4 and B.5), the graphs are identical. This means that for the higher rank accuracy, the n-best supertagger already performs in an optimal way for the reference test set and it actually cannot get any better with this kind of training material. It is obvious that cumacc(5) will probably never reach 100%, i.e. all target words would appear within the top 5 ranks and the cognitive load of the user would be reduced immensely because the search via scrolling through the candidates no longer applies. The words that appear after the fifth position in the candidate lists (approx. 4.5% for the reference test set) are probably again the result of the small corpus size. If a larger training corpus had been available, it is very likely that the accuracy would be slightly higher since more training samples of those words that appear after the fifth position would have been encountered. A cumulative accuracy of 98–99% for rank 5 is very desirable and should be the goal of further examination (see also Section 7.1).

6.3.3 A* versus greedy search

The comparison of the n-best supertagger when using A* or greedy search for the evaluation is summarized in Figures B.7–B.12. As can be seen, the greedy approach tends to find slightly better solutions in early stages of the search, i.e. for small values of n. This behavior is visible in Figures B.10, B.11 and B.12. Nevertheless, the overall performance of the A* search is better. An experiment for several hypothesis sizes showed no significant differences between the two approaches in terms of execution time. The greedy search was always slightly faster (approx. 2s for $n = 1000$), but the gain was in the vicinity of only a few hundred milliseconds for smaller values of n, thus being negligible.

6.3.4 Experiment with word trigrams

Finally, a small experiment was carried out that compares the n-best supertagger to a language model that is based on word trigrams instead of unigrams. The idea is the following: instead of having a training corpus where the words are annotated with their corresponding supertags, it is annotated with the words themselves. So, the supertags of the annotated sentence *das Bild meines Monitors flimmert* ("*the picture on my monitor flickers*") on the left of the following table, e.g., are replaced by the target words (see right column):

das	betaDnx		das	das
bild	alphaNXN		bild	bild
meines	betaDnx	$\Longrightarrow$	meines	meines
monitors	betanxN		monitors	monitors
flimmert	alphanx0V		flimmert	flimmert
%%%END_OF_SENTENCE			%%%END_OF_SENTENCE	

The resulting corpus can be instantly used with the implemented n-best supertagger presented in Chapter 5 by skipping the LDA step that connects to the LTAG and its lexicon. The result is a trigram language model based on words instead of supertags. The graphs are shown in Figures B.13–B.18. The most interesting property of the word trigram model is that it outperforms the supertagging based approach for rank $= 1$ if the simple Viterbi search is used (1-best search). In fact, all results have their maximum for $n = 1$. This indicates that a word-based trigram model tends to boost only single entries that occur within the language model whereas the supertag-based approach uses word-class information which results in boosting all words of a certain class if it has been encountered in the training data, thus increasing the probability of a whole set of words.

6.4 Discussion

This chapter presented the evaluation of the n-best supertagger that is based on the theoretical background introduced in Chapters 2 and 4 and whose implementational details were described in Chapter 5. The domain-specific corpus is no typical evaluation corpus for use in AAC systems. Nevertheless, it shows some features of colloquial speech that emerge in everyday conversations since it contains sentences from newsgroup postings, and the language in newsgroups and forums has often very similar aspects of spoken language. The obtained results have to be seen with some reservations, though. The overall system is an early prototype and can be further improved. Due to the small corpus, the n-best supertagger uses only supertag information without the lexical anchor (e.g. `alphaNXN` instead of `alphaNXN//ich`) within the hypotheses of this evaluation in order to reduce the problem with data sparseness. Bäcker's supertagger showed good results for the corpus because of state-of-the-art discounting and back-off methods. The accuracy of the system is high since the words in the sentence are unambiguous and the Viterbi search can easily find the best supertag annotation. In this approach, the level of ambiguity rises enormously. The codes expand to words, and these words again expand to possible supertags. So, in order to achieve reliable results through expressive language models, the training

size has to be much larger than the available 225 sentences. Usually, statistical approaches use big corpora with millions of words or sentences, which was not the case in this study. The poor quality of the hypotheses can also be observed when leaving out the additional lightweight dependency analysis. The graph "¬LDA" in Figure 6.3 shows the evaluation based only on the match list boosting for hypotheses returned by the n-best search. As can be seen, the accuracy decreases minimally for rank 1 (the difference is approx. 1%). This is due to default supertags that the back-off mechanism chooses for unknown trigrams at the end of the back-off chain. At these positions, the LDA cannot find consistent readings any more, and the maximum dependency coverage criterion is not fulfilled.

Improving the Viterbi search (1-best) to an A* n-best search has a significant impact on the results. Nonetheless, a better estimation of the trigram language model on a larger corpus would certainly yield better results. One important goal is to get the cumulative accuracy of rank 5 as high as possible. The famous article of (Miller, 1956) is widely accepted and argues that people have problems coping with "information chunks" larger than 7 ± 2 units. Thus, the candidate lists in word prediction usually contain 5 words. Larger values significantly increase the cognitive load by scanning the lists and hence slow down the overall typing process. The evaluation in this chapter shows that the basic approach presented in the course of this study has the potential to improve a sentence-wise text entry for the communication aid UKO-II. Clearly, the primary goal is to further increase the accuracy of rank 1 such that as many words as possible come out at the top of the suggestion lists. Section 7.1 elaborates on possible areas of future investigations.

7 Conclusion

The main motivation of this study was to present the framework of a communication aid that uses a highly reduced keyboard and allows for a sentence-wise text entry. The argument against a system that solely relies on word prediction is supported by the cognitive load of scanning the candidate lists after each new keypress for a possible completion to the current word. In a sentence-wise approach, the user can focus on the text entry and afterwards make the final adjustments of the candidate lists by selecting the correct word. Ideally, the system finds the correct sentence hypothesis and thus, the user has no additional selection costs beside one action that accepts the top reading.

In Chapter 3, an introduction was given to the area of Augmentative and Alternative Communication. The text entry framework for reduced keyboards was presented as well as some existing approaches in the field of word completion. The only sentence-wise approach known to us so far is the work of (Rau and Skiena, 1996). The results obtained in this study cannot compete with the ones presented in the above citation. On the one hand, this is due to a much higher ambiguity of the reduced keyboard in our case (we use only 3 letter keys instead of 9) and, on the other, the small evaluation corpus has not the power to produce an expressive language model based on trigrams. Nevertheless, the basic idea seems promising and should be elaborated in further investigations. The theoretical background for the n-best supertagger that is able to find the most promising supertag hypotheses of a coded sentence was presented in Chapters 2 and 4. The main aspects are concerned with the use of n-grams and Hidden Markov Models to find likely supertags for the corresponding words and combine the Viterbi search with the A* search paradigm from the field of Artificial Intelligence to yield the n best hypotheses. This list of hypotheses is used to adjust the candidate lists such that likely interpretations of the code sequence appear at the top. The process, which was named "match list boosting", was presented in Section 5.2. It was argued that an additional lightweight dependency analysis filters the n-best lists to pick the hypotheses that have the best dependency coverage for the supertags. This is equivalent to a shallow parse of the sentence hypotheses and discards inconsistent readings. In the final evaluation, this analysis only gave a very small improvement in terms of accuracy. In a separate small experiment with a sentence containing no unknown trigrams (see Figure 5.6 and Appendix C), the performance was much better and the LDA significantly improved the final result. The bad performance of the LDA can be tracked down to

low-quality supertag hypotheses. Due to the small size of the training corpus, many trigrams were unknown in the test set, so the system backed off to default supertags for the words that were not covered in the trained language model.

The evaluation showed an increase in the accuracy of rank 1 from 50% up to approx. 67% for the best run with $n = 592$ and resulted in a decrease of the average rank by one position. For the test corpus, the words could be located on the second instead of the third position within the suggestion lists on average. At this point, the obtained results may not look so good to the one or another, but we think that for this size of the training material the achievements are promising. A larger drawback is probably the strong assumption of a perfect typist who makes no errors. The problems with wrongly entered words, i.e. where an incorrect key was pressed or the user forgot a letter, were not addressed in this study since an evaluation of these features needs real user trials. It is obvious that these kind of errors are very frustrating if the basic approach is a sentence-wise text entry since the error is not noticed at the end of a word but rather at the end of the whole sentence. The other problem was already noted in (Rau and Skiena, 1996). Typing comfort decreases with increasing length of the sentences. The conclusion is that a sentence-wise approach will probably only be feasible for shorter sentences, as they arise e.g. in conversational communication. For writing longer texts, the use of traditional word completion might give better results.

7.1 Future work

The list of future improvements is quite long. This study only introduced the basic concepts of a sentence-wise text entry system. Further evaluations should be undertaken with much larger corpora. Since the current implementation works on annotated corpora, a possible improvement would certainly be the application of Baum-Welch reestimation to automatically annotate the evaluation corpus. The use of a more general LTAG with an extended lexicon is needed for this task. An extensive LTAG exists e.g. for the English language (XTAG Research Group, 2001). The LTAG used in this work was tailored for the use with the domain-specific evaluation corpus.

The notion of entropy and perplexity was shortly mentioned in Section 2.2.3 as a measure for the quality of a language model. Since the overall framework of the procedures is only partly based on statistical language models (the use of n-grams and HMMs) and yields more of a hybrid approach for the candidate list orderings (cf. match list boosting), a computation of the entropy is not possible due the lack of final probability estimates of the words. If a more mathematical way of characterizing the process of boosting entries is derived, the use of entropy could be considered as a quality measure of the overall performance of the system.

The number of hypotheses in the n-best search, i.e. the parameter n, stayed constant during a simulation run on the test corpus. However, experiments should be carried out with various values of n, depending on the length of the sentence. For long sentences, n should be considerably large, whereas for short sentences, the accuracy might benefit more from a small number of hypotheses that are considered for boosting. So further investigations are necessary to experiment with a dynamic choice of the parameter n.

As already noted, one of the primary goals is to increase the cumulative accuracy of rank 5 such that the user has not to scroll the list of suggestions up and down in search of the target word. One possibility is to use topic- or user-specific lexicons to "prime" the general dictionary (e.g. CELEX) by linear interpolation as presented in (Harbusch et al., 2003). Beside lower OOV rates, the topic- and user-specific words also appear at a higher rank in the candidate lists. As was shown in Section 6.3, approx. 95.5% of the words appeared within the first five ranks. The use of a better language model based on a larger training corpus might have a positive effect on this percentage since the hypotheses will cover more words through the corresponding supertags. But this value can also be increased if more words initially appear at higher ranks, which is achieved by linear interpolation of the CELEX dictionary and a corpus-specific dictionary that is based on the training data.

A UML class diagrams

This chapter presents the main UML class diagrams of the components reused from (Bäcker, 2002) and newly modeled and implemented ones for the n-best supertagger and the evaluation of the test corpus. The implementation is made in JAVA (see e.g. (Arnold and Gosling, 1997)).

A.1 Package supertagging

Figure A.1 (page 102) shows the initial components of the supertagger from (Bäcker, 2002). The class TrigramSuperTaggingTrainer estimates the probabilities of the trigram HMM directly on an annotated training corpus and is used in this thesis to create the language model on the 225 sentences of the evaluation set.

A.2 Package supertagging.nbest

Figures A.2 and A.3 (pages 103–104) present the main components of the package supertagging.nbest which implements the modifications to Bäcker's supertagger.

A.3 Package LDA

Figure A.4 (page 105) is basically taken from (Bäcker, 2002) and shows the connection of the class CodedNBestSuperTaggingEvaluator to the Lightweight Dependency Analyzer.

A.4 Package evaluate

Figure A.5 (page 105) shows the main classes for running the n-best supertagger. NBestSuperTagger runs the evaluation of a test corpus on a trained model in various modes and with variable hypothesis sizes.

```
[yonker@kazzbajjah ~]$ java evaluate.NBestSuperTagger
usage: java evaluate.NBestSuperTagger <trigram-model-file>
<tagged-corpus-file> [<N-best> <mode> <greedy>]
```

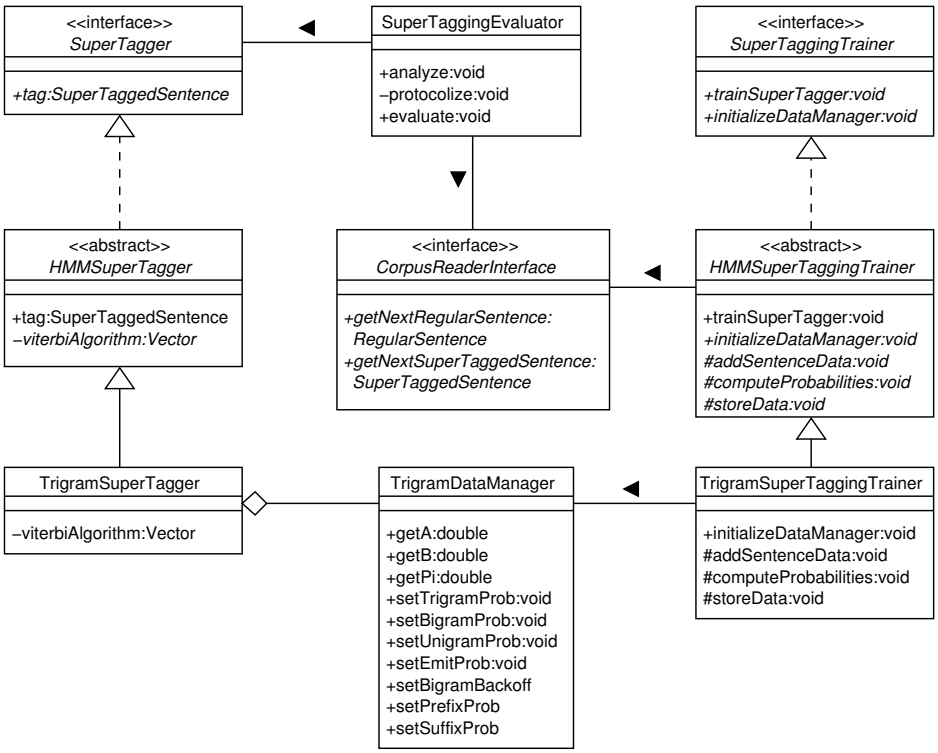

Figure A.1: UML class diagram of the initial supertagger (from (Bäcker, 2002)).

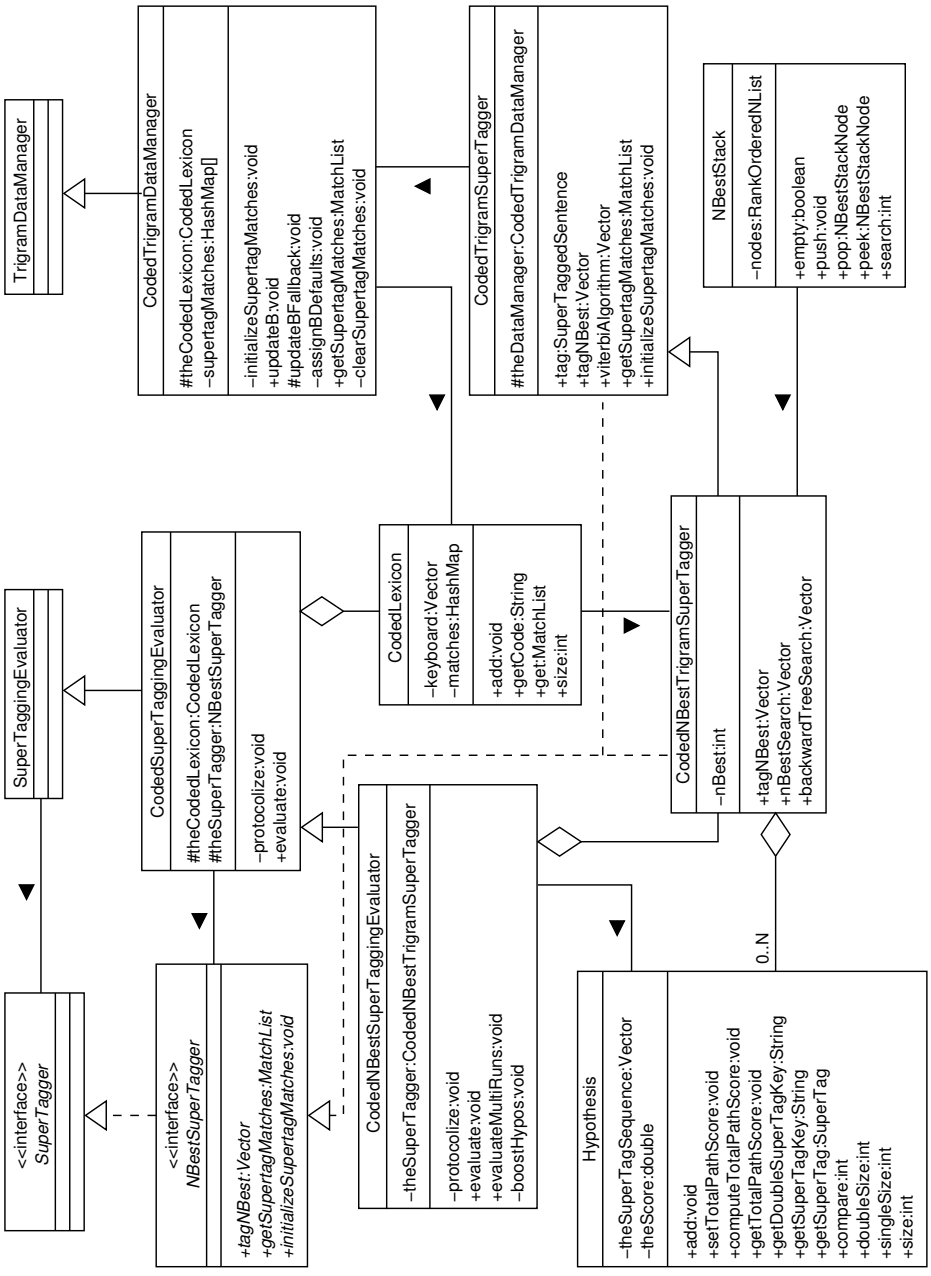

Figure A.2: UML class diagram of the n-best supertagger.

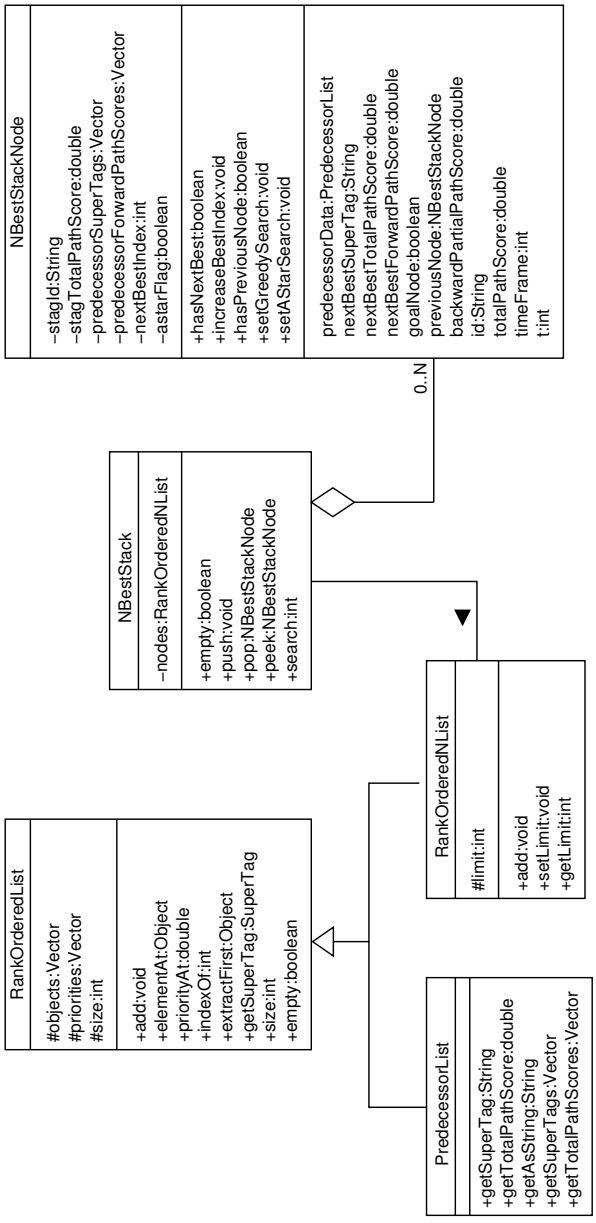

Figure A.3: UML class diagram of the n-best stack.

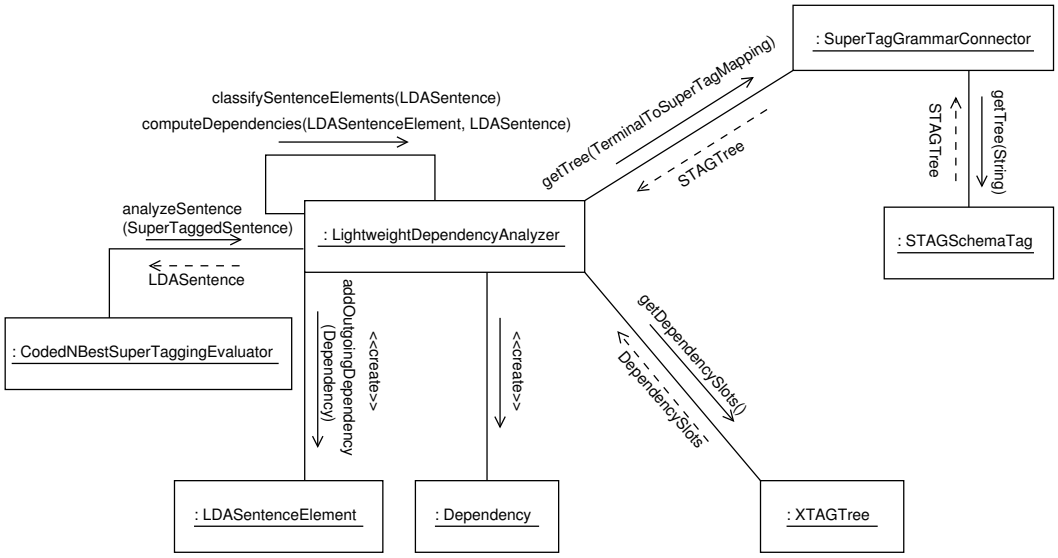

Figure A.4: UML collaboration diagram showing how the LDA connects to the LTAG for additional analysis of supertag hypotheses from CodedNBest-SuperTaggingEvaluator (mostly from (Bäcker, 2002)).

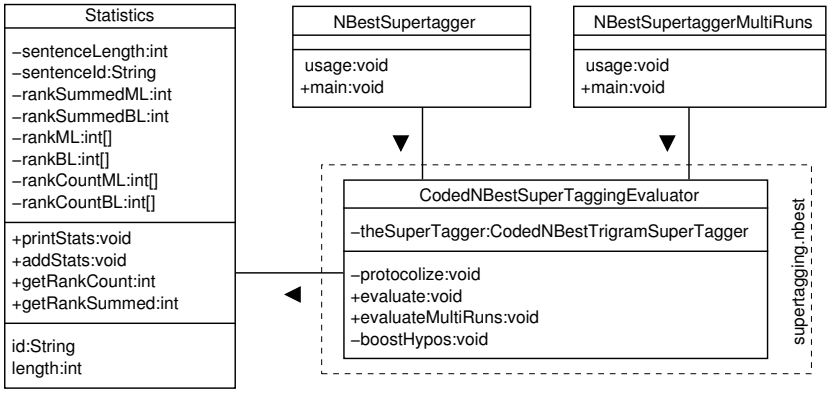

Figure A.5: UML class diagram of the classes used for evaluation.

B Evaluation graphs

This chapter gives a detailed overview on the graphs obtained from the evaluation of the test corpus. The following labels are used throughout the thesis:

- BASE: baseline using unigrams for the language model (candidate lists sorted by CELEX frequency)
- STAG: n-best trigram supertagging with lightweight dependency analysis
- BEST: upper bound experiment, boosting best hypotheses according to target words
- GREEDY: same as STAG except using greedy search instead of A*
- TRIGRAM: lanugage model based on word trigrams

B.1 STAG vs. BEST

Figures B.1–B.6 (pages 108–110) present the detailed graphs for the n-best supertagger compared to the upper bound and baseline.

B.2 STAG A* vs. greedy search

Figures B.7–B.12 (pages 111–113) present the detailed graphs for the n-best supertagger (A* search) compared to a different search strategy (greedy search).

B.3 STAG vs. TRIGRAM

Figures B.13–B.18 (pages 114–116) present the detailed graphs for the n-best supertagger compared to a language model based on word trigrams (instead of supertags).

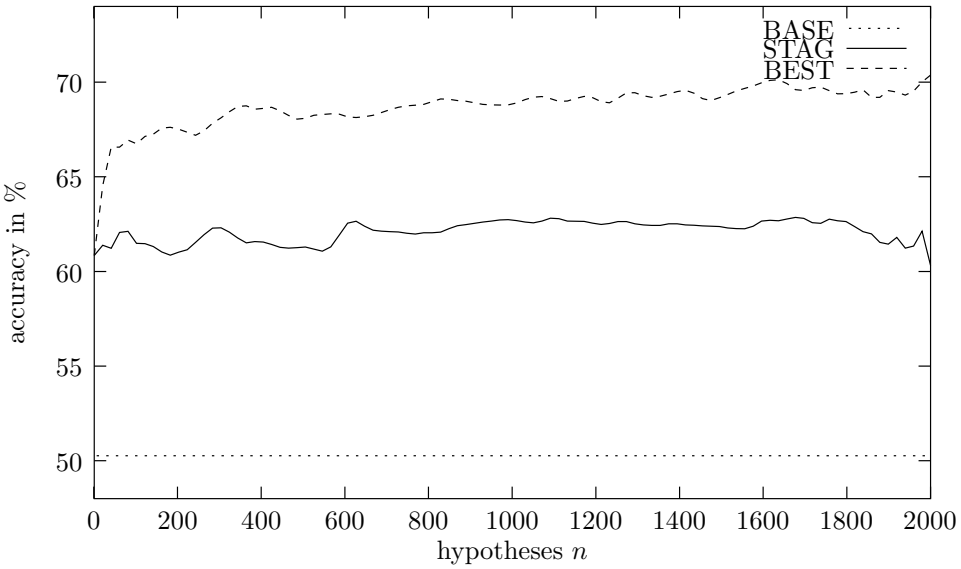

Figure B.1: STAG vs. upper bound, rank = 1.

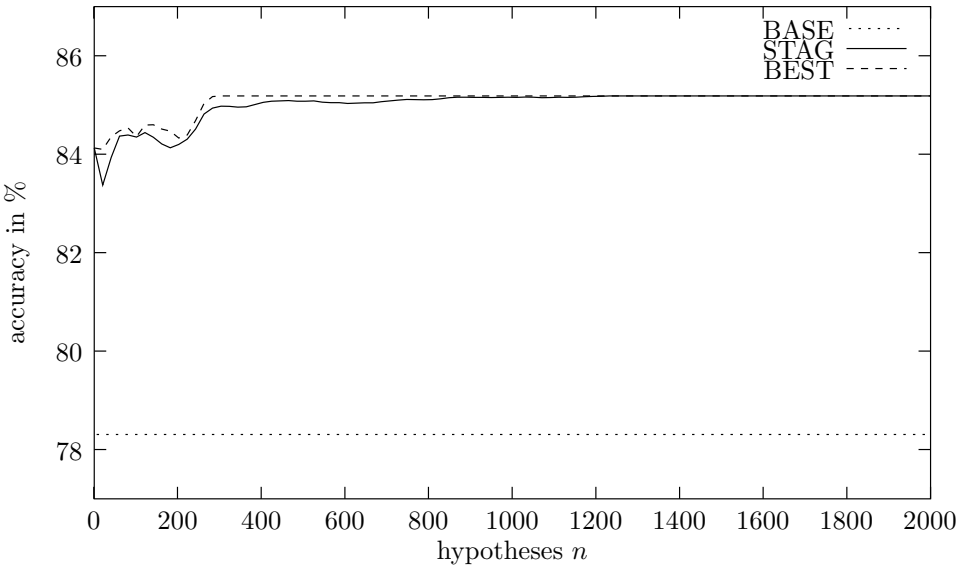

Figure B.2: STAG vs. upper bound, rank $\leq$ 2.

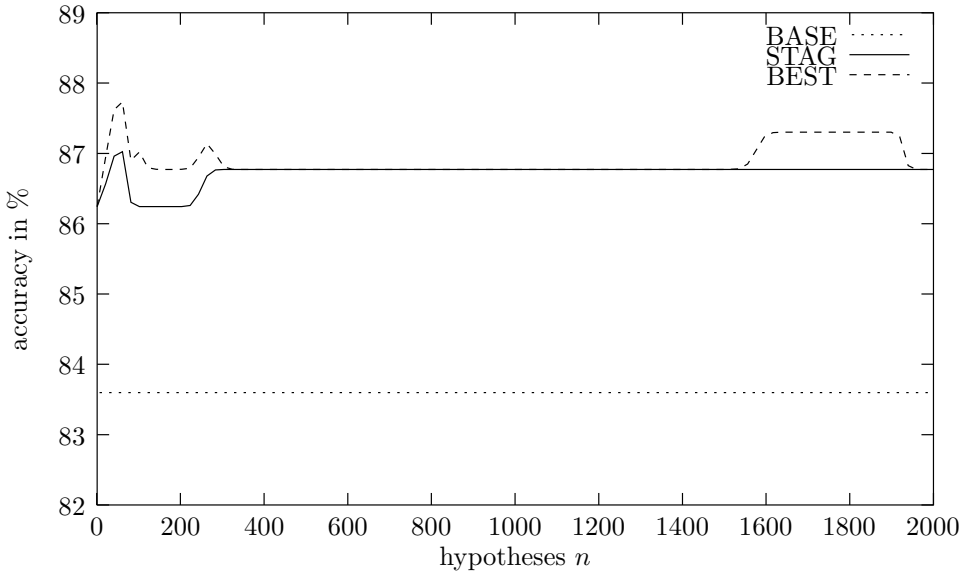

Figure B.3: STAG vs. upper bound, rank ≤ 3.

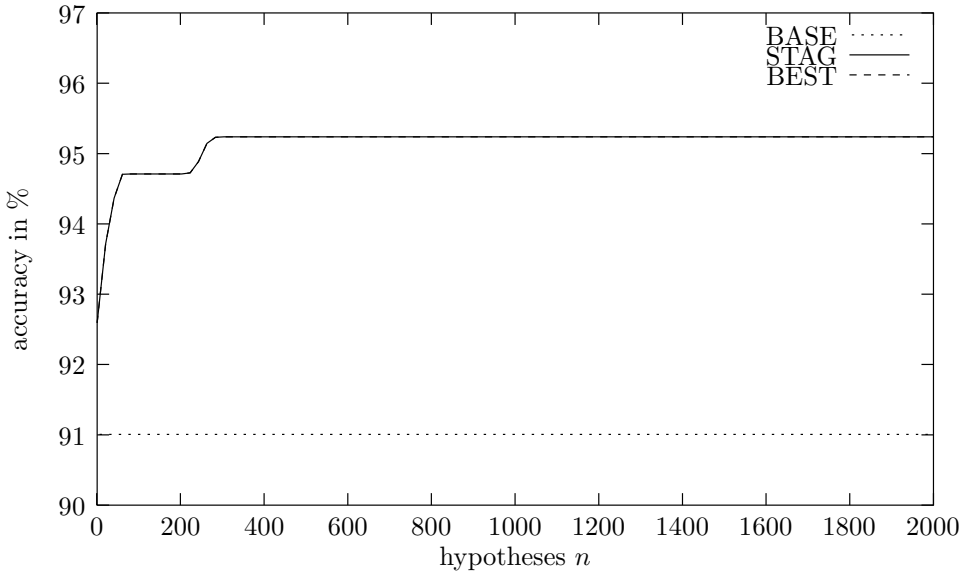

Figure B.4: STAG vs. upper bound, rank ≤ 4.

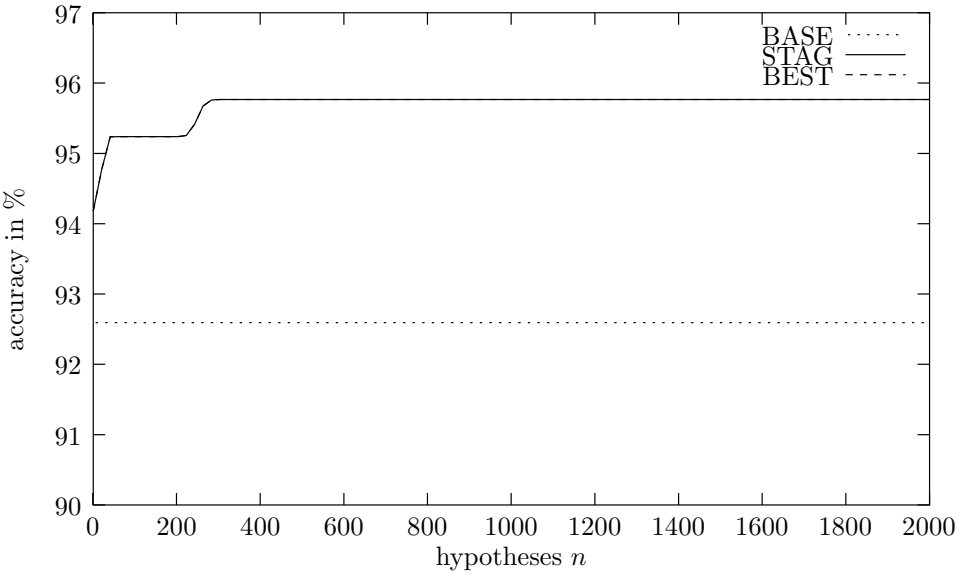

Figure B.5: STAG vs. upper bound, rank ≤ 5.

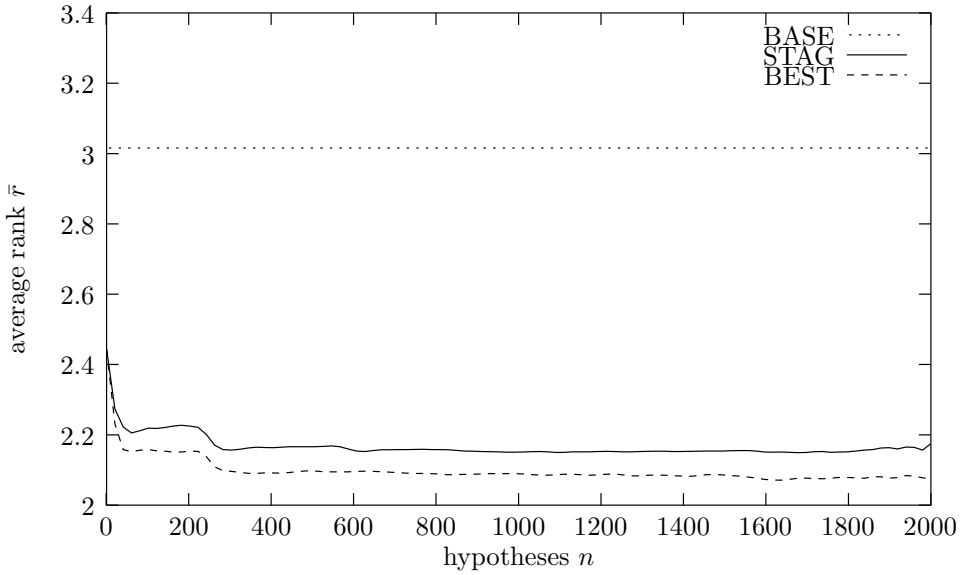

Figure B.6: STAG vs. upper bound, average rank.

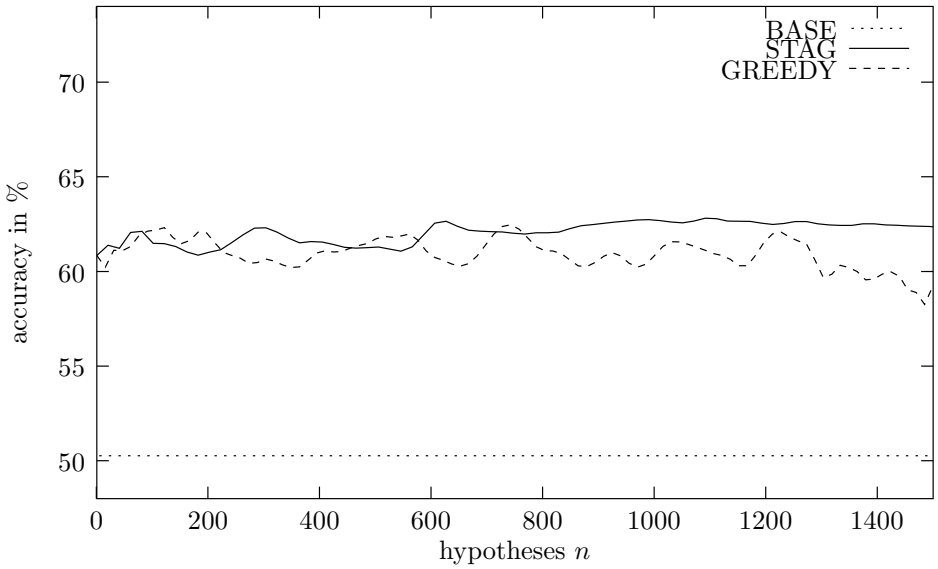

Figure B.7: STAG A* vs. Greedy, rank = 1.

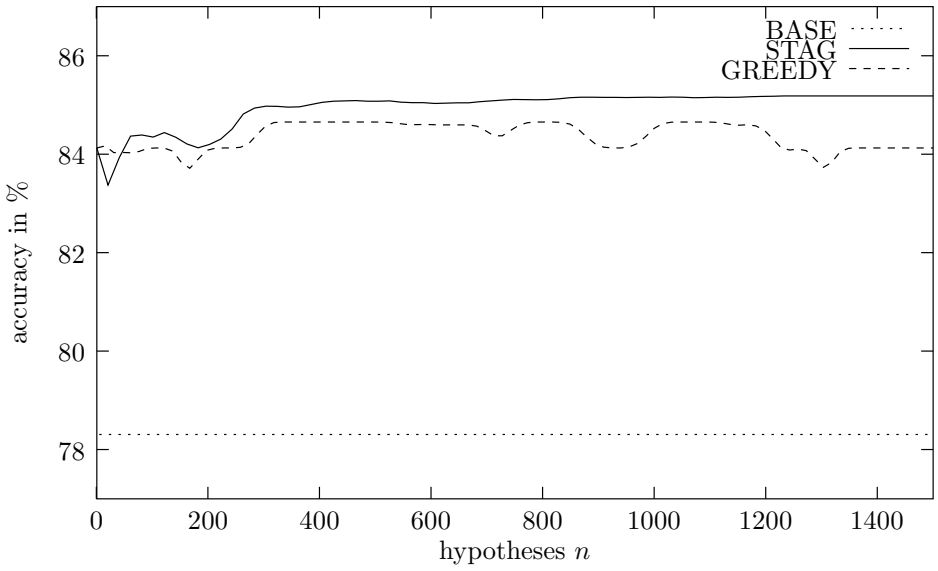

Figure B.8: STAG A* vs. Greedy, rank $\leq$ 2.

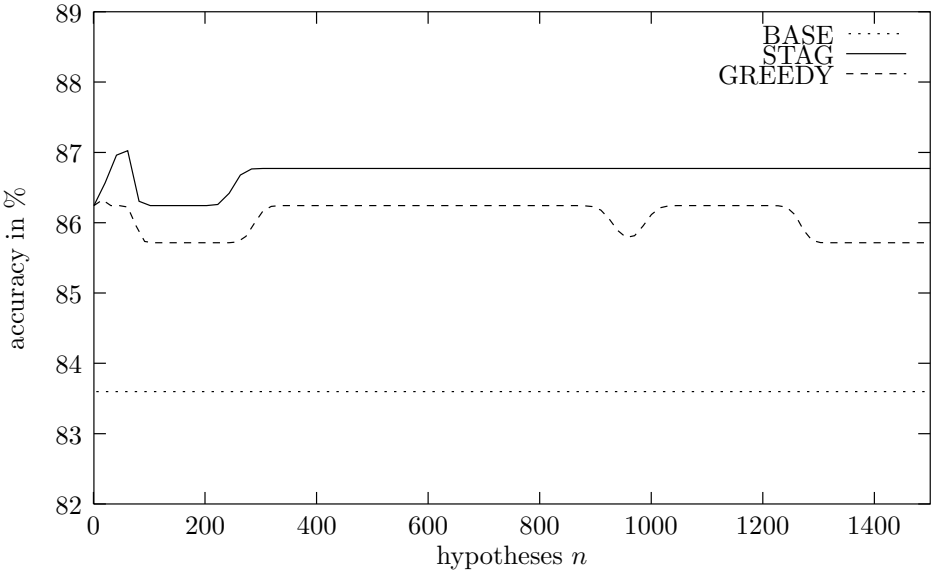

Figure B.9: STAG A* vs. Greedy, rank ≤ 3.

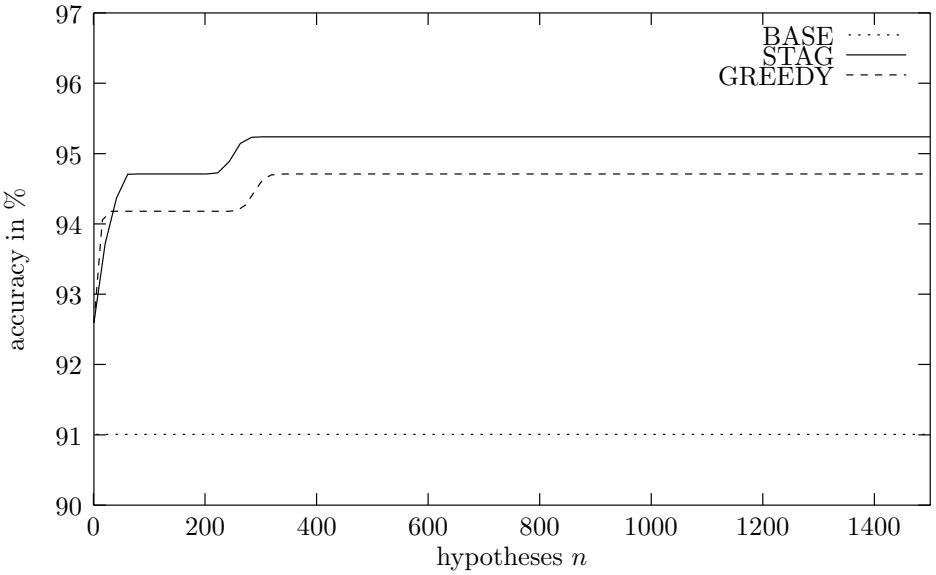

Figure B.10: STAG A* vs. Greedy, rank ≤ 4.

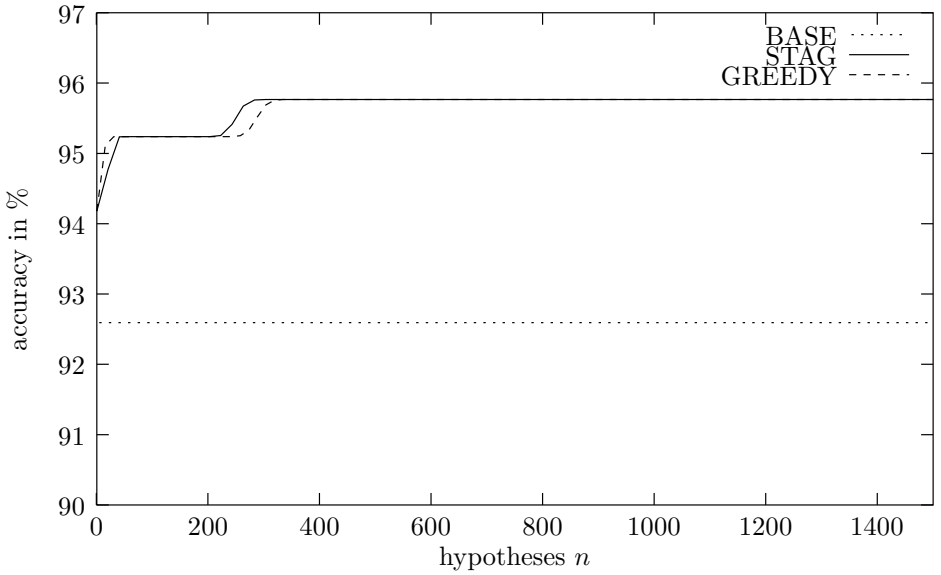

Figure B.11: STAG A* vs. Greedy, rank ≤ 5.

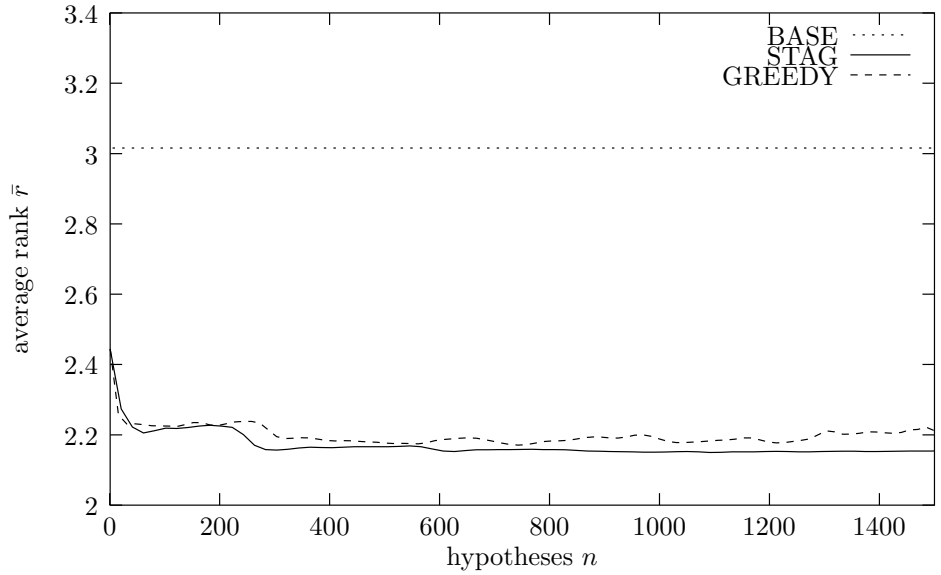

Figure B.12: STAG vs. Greedy, average rank.

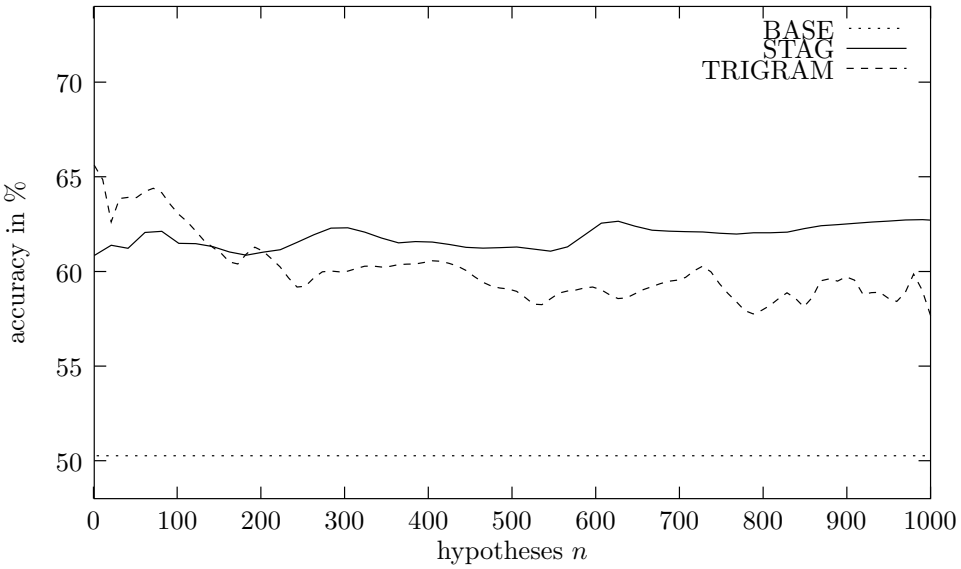

Figure B.13: STAG vs. word trigrams, rank = 1.

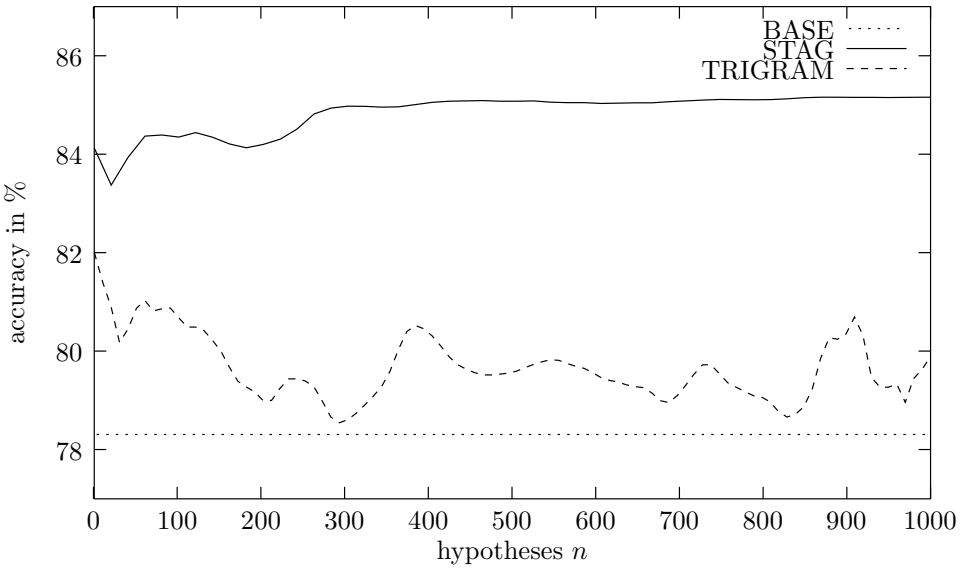

Figure B.14: STAG vs. word trigrams, rank $\leq$ 2.

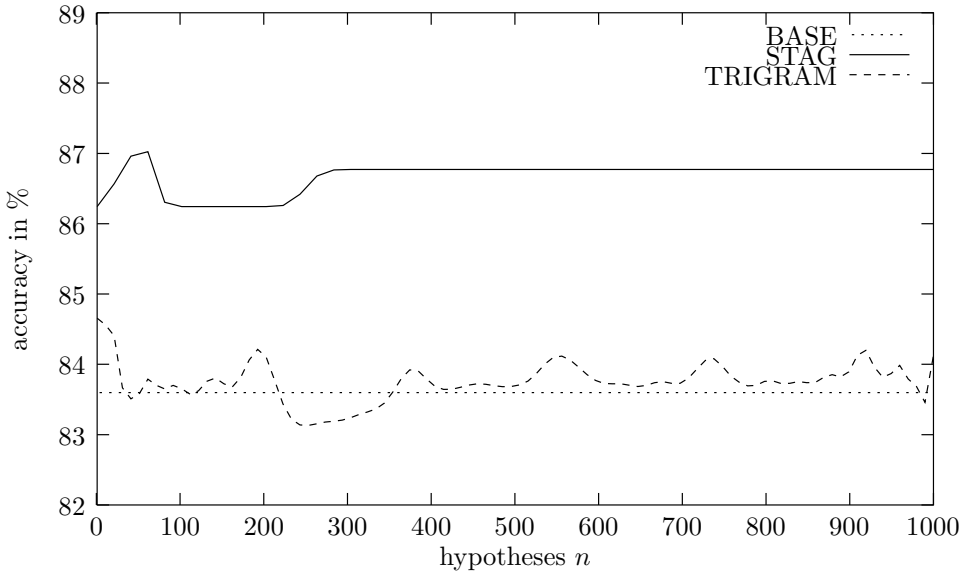

Figure B.15: STAG vs. word trigrams, rank ≤ 3.

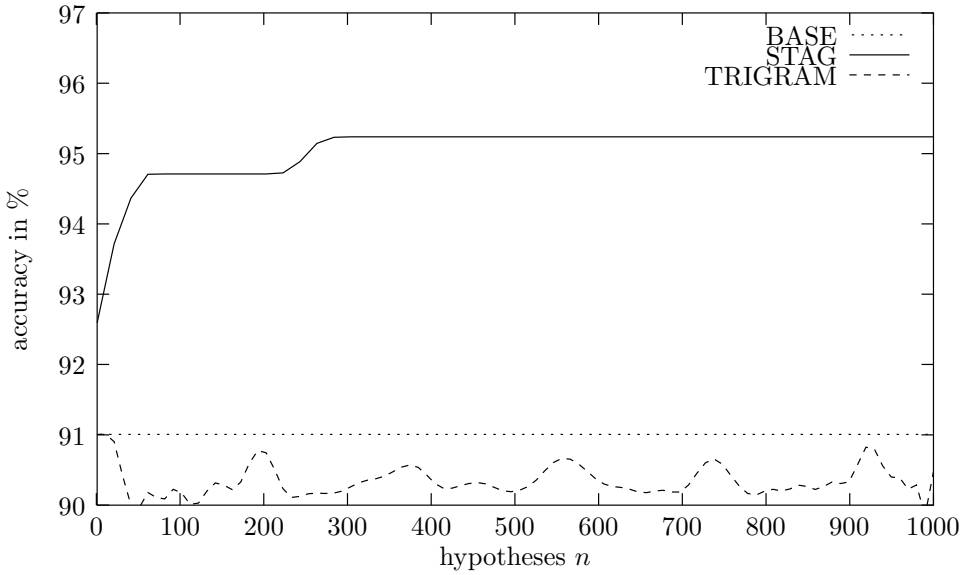

Figure B.16: STAG vs. word trigrams, rank ≤ 4.

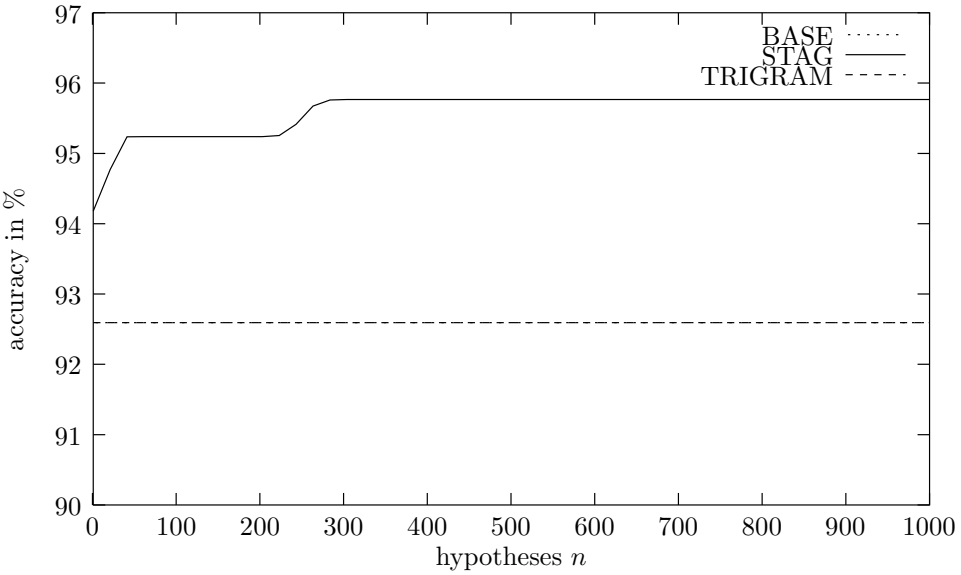

Figure B.17: STAG vs. word trigrams, rank ≤ 5.

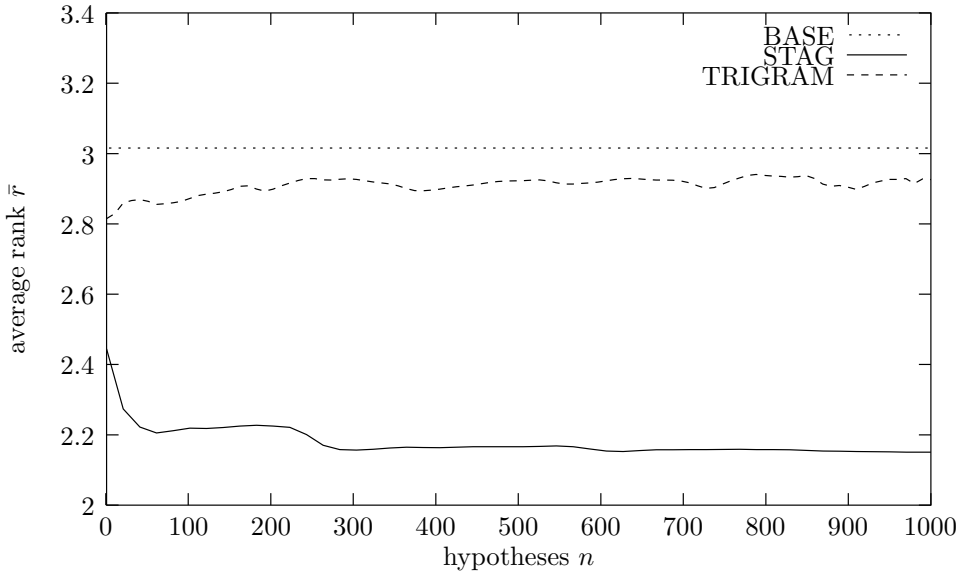

Figure B.18: STAG vs. word trigrams, average rank.

C Test run

```
[yonker@kazzbajjah Testing]$ java evaluate.NBestSuperTagger train
ich-habe-ein-kleines-problem.tagged 5 LDA

Reading coded lexicon 'train.codlex'... done.
3 keys: < agjlmqrwzä> < cfhkostuvxyüß> < bdeinpö>
460 entries in hash map.
Running N-best version (N = 5)
Reading supertagger model... done.
Loading XTAG...
Using: XTAG German grammar for KoHDaS
Load template <KoHDaS-grammar/templates.lex>
Load syntax <KoHDaS-grammar/syntax/syntax-coded.flat>
Load tree files from <KoHDaS-grammar/grammar/TnxOV.trees><TnxOVnx1.trees><TnxOVpnx1.trees>
<TnxOVPnx1.trees><TnxOVnx1nx2.trees><TnxOVpl.trees><TnxOVplnx1.trees><TnxOVplnx1nx2.trees>
<TnxOVs1.trees><TnxOVa.trees><Trest.trees><lex_STAG.trees><determiners.trees>
<advs-adjs.trees><modifiers.trees><prepositions.trees><conjunctions.trees><neg.trees>
Running n-best supertagger in mode 'lda'...
Tagging: [...EOS..., ...EOS..., ich, habe, ein, kleines, problem, ...EOS..., ...EOS...]
TERMINAL:       ich//alphaNXN//[211]: ich
TERMINAL:       habe//alphaV//[1022]: habe kann
TERMINAL:       ein//betavPnx//[222]: bei
TERMINAL:       kleines//betaAn//[1022221]: kleines
TERMINAL:       problem//alphaNXN//[2012020]: problem
LDA analyzed sentence: 0:alphaNXN[][?]; 1:alphaV[][-]; 2:betavPnx[][1*, 4.];
                       3:betaAn[][4*]; 4:alphaNXN[][-]
LDA toplevel: [0:alphaNXN[][?], 2:betavPnx[][1*, 4.], 3:betaAn[][4*]]
TERMINAL:       ich//alphaNXN//[211]: ich
TERMINAL:       habe//alphanxOVnx1//[1022]: habe
TERMINAL:       ein//alphaNXN//[222]: ein
TERMINAL:       kleines//betaAn//[1022221]: kleines
TERMINAL:       problem//alphaNXN//[2012020]: problem
LDA analyzed sentence: 0:alphaNXN[][-]; 1:alphanxOVnx1[][0., 2.]; 2:alphaNXN[][-];
                       3:betaAn[][4*]; 4:alphaNXN[][-]
LDA toplevel: [1:alphanxOVnx1[][0., 2.], 3:betaAn[][4*]]
TERMINAL:       ich//alphaNXN//[211]: ich
TERMINAL:       habe//alphaV//[1022]: habe kann
TERMINAL:       ein//alphaNXN//[222]: ein
TERMINAL:       kleines//betaAn//[1022221]: kleines
TERMINAL:       problem//alphaNXN//[2012020]: problem
LDA analyzed sentence: 0:alphaNXN[][?]; 1:alphaV[][?]; 2:alphaNXN[][?];
                       3:betaAn[][4*]; 4:alphaNXN[][-]
LDA toplevel: [0:alphaNXN[][?], 1:alphaV[][?], 2:alphaNXN[][?], 3:betaAn[][4*]]
TERMINAL:       ich//alphaNXN//[211]: ich
TERMINAL:       habe//alphanxOVnx1//[1022]: habe
TERMINAL:       ein//betaDnx//[222]: ein den die
TERMINAL:       kleines//betaAn//[1022221]: kleines
TERMINAL:       problem//alphaNXN//[2012020]: problem
LDA analyzed sentence: 0:alphaNXN[][-]; 1:alphanxOVnx1[][0., 4.]; 2:betaDnx[][4*];
                       3:betaAn[][4*]; 4:alphaNXN[][-]
```

```
LDA toplevel: [1:alphanx0Vnx1[][0., 4.], 2:betaDnx[][4*], 3:betaAn[][4*]]
TERMINAL:       ich//alphaNXN//[211]: ich
TERMINAL:       habe//alphaV//[1022]: habe kann
TERMINAL:       ein//betaDnx//[222]: ein den die
TERMINAL:       kleines//betaAn//[1022221]: kleines
TERMINAL:       problem//alphaNXN//[2012020]: problem
LDA analyzed sentence: 0:alphaNXN[][?]; 1:alphaV[][?]; 2:betaDnx[][4*];
                       3:betaAn[][4*]; 4:alphaNXN[][-]
LDA toplevel: [0:alphaNXN[][?], 1:alphaV[][?], 2:betaDnx[][4*], 3:betaAn[][4*]]
BHyps: 1 3
1. "ich":
Original match list: [211]: ist ich not bot ißt bus boß ... (11 more entries)
 Boosted match list: [211]: ich ist not bot ißt bus boß ... (11 more entries)
2. "habe":
Original match list: [1022]: kann habe hand fand frei sand ... (20 more entries)
 Boosted match list: [1022]: habe kann hand fand frei sand ... (20 more entries)
3. "ein":
Original match list: [222]: die den bei ein bin nie eid öde eie böe
 Boosted match list: [222]: ein die den bei bin nie eid öde eie böe
4. "kleines":
Original match list: [1022221]: kleines kleidet kleides kreidet ... (12 more entries)
 Boosted match list: [1022221]: kleines kleidet kleides kreidet ... (12 more entries)
5. "problem":
Original match list: [2012020]: problem näherer näherem emsiger ... (4 more entries)
Boosted match list: [2012020]: problem näherer näherem emsiger ... (4 more entries)
HYP: alphaNXN[ich]; alphanx0Vnx1[habe]; betaDnx[ein]; betaAn[kleines]; alphaNXN[problem]
HYPBEST 1
HYPMATC 4       5       3       4       3
HYPFAIL 1       0       2       1       2
HYPLDAC 3       5       2       5       4
HYPLDAT 4       3       4       2       3
HYPLDAMAX       5       5
HYPLDAMAXindices        1       3
HYPLDABESTcov   5
HYPLDABESTindices       1
HYPLDABESTtop   3
Hypothesis 1: alphaNXN alphaV betaDnx betaAn alphaNXN (-240.06063154421645)
Hypothesis 2: alphaNXN alphanx0Vnx1 betaDnx betaAn alphaNXN (-240.38368902077767)
Hypothesis 3: alphaNXN alphaV alphaNXN betaAn alphaNXN (-243.89617070437882)
Hypothesis 4: alphaNXN alphanx0Vnx1 alphaNXN betaAn alphaNXN (-244.12911205503033)
Hypothesis 5: alphaNXN alphaV betavPnx betaAn alphaNXN (-244.40697282629282)
==============================
number of test sentences: 1
number of test tokens: 5
number of correctly tagged sentences: 1
number of correctly tagged tokens: 4
percentage of correct sentences: 100.0%
percentage of correct tokens (avg): 80.0%
total time consumed: 68
tagged sentences per minute: 882.3529411764705
==============================
TotalRank1      40.0    100.0
TotalRank2      80.0    100.0
TotalRank3      80.0    100.0
TotalRank4      100.0   100.0
TotalRank5      100.0   100.0
TotalAvg        2.0     1.0
done.
```

Bibliography

Abney, S. (1991). Parsing by chunks. In (Berwick et al., 1991), pages 257–278.

Abney, S. (1997). Part-of-speech tagging and partial parsing. In (Young and Bloothooft, 1997), pages 118–136.

ACL97-WS (1997). *Proceedings of the Workshop on Natural Language Processing for Communication Aids*, Madrid, Spain. Association for Computational Linguistics.

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA.

Allen, J. F. (1995). *Natural Language Understanding*. Benjamin/Cummings, Redwood City, CA, USA, 2nd edition.

Armstrong, S., Church, K., Isabelle, P., Manzi, S., Tzoukermann, E., and Yarowsky, D., editors (1999). *Natural Language Processing Using Very Large Corpora*, volume 11 of *Text, Speech and Language Technology*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Arnold, K. and Gosling, J. (1997). *The Java Programming Language*. Addison-Wesley, Reading, MA, USA, 2nd edition.

Atkins, B. T. S. and Zampolli, A., editors (1994). *Computational Approaches to the Lexicon*. Oxford University Press.

Baayen, R. H., Piepenbrock, R., and L., G. (1995). The CELEX lexical database (release 2). CD-ROM, Linguistic Data Consortium, University of Pennsylvania, PA, USA.

Bäcker, J. (2001). *Entwicklung eines Supertaggers für das Deutsche*. Studienarbeit, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, Germany.

Bäcker, J. (2002). *KoHDaS-ST — Supertagging in dem automatischen Dialogsystem KoHDaS*. Diplomarbeit, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, Germany.

Bäcker, J. and Harbusch, K. (2002). Hidden Markov model-based supertagging in a user initiative dialogue system. In *Proceedings of the Sixth International Workshop on Tree Adjoining Grammar and Related Frameworks (TAG+6)*, pages 269–278, Venice, Italy.

Bahl, L. R., Jelinek, F., and Mercer, R. L. (1983). A maximum likelihood approach to continuous speech recognition. In (Waibel and Lee, 1990), pages 308–319.

Baum, L. E. (1972). An inequality and associated maximization technique in statistical estimation for probabilistic functions of Markov processes. In Shisha, O., editor, *Inequalities*, volume 3, pages 1–8, University of California, Los Angeles, CA, USA. Academic Press.

Beck, A. R. and Fritz, H. (1998). Can people who have aphasia learn iconic codes? *Augmentative and Alternative Communication*, 14(3):184–196.

Berwick, R. C. (1991). Principles of principle-based parsing. In (Berwick et al., 1991), pages 1–37.

Berwick, R. C., Abney, S. P., and Tenny, C., editors (1991). *Principle-based parsing: Computation and Psycholinguistics*, volume 44 of *Studies in Linguistics and Philosophy*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Boguraev, B. K., Garigliano, R., and Langer, S., editors (1998). *Natural Language Engineering*, volume 4(1). Cambridge University Press.

Brants, T. (2000). TnT — A statistical part-of-speech tagger. In *Proceedings of the 6th Conference on Applied Natural Language Processing (ANLP '00)*, pages 224–231, Seattle, WA, USA.

Brill, E. (1992). A simple rule-based part-of-speech tagger. In *Proceedings of the 3rd Conference on Applied Natural Language Processing (ANLP '92)*, pages 152–155, Trento, Italy.

Brill, E. (1994). Some advances in transformation-based part of speech tagging. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI '94)*, pages 722–727, Seattle, WA, USA.

Brill, E. (1995). Transformation-based error-driven learning and natural language processing: A case study in part of speech tagging. *Computational Linguistics*, 21(4):543–566.

Brown, P. F., Della Pietra, S. A., Della Pietra, V. J., Lai, J. C., and Mercer, R. L. (1992). An estimate of an upper bound for the entropy of English. *Computational Linguistics*, 18(1):31–40.

Charniak, E. (1993). *Statistical Language Learning*. MIT Press, Cambridge, MA, USA.

Chow, Y.-L. and Schwartz, R. (1989). The N-best algorithm: An efficient procedure for finding top N sentence hypotheses. In *Proceedings of the DARPA Speech and Natural Language Workshop*, pages 199–202, Cape Cod, MA, USA.

Church, K. W. (1988). A stochastic parts program and noun phrase parser for unrestricted text. In *Proceedings of the 2nd Conference on Applied Natural Language Processing*, pages 136–143, Morristown, NJ, USA. Association for Computational Linguistics.

Church, K. W. and Mercer, R. L. (1993). Introduction to the special issue on computational linguistics using large corpora. *Computational Linguistics*, 19(1):1–24.

Clarkson, P. and Rosenfeld, R. (1997). Statistical language modeling using the CMU-Cambridge toolkit. In *Proceedings of Eurospeech '97*, volume 5, pages 2707–2710, Rhodes, Greece.

Cleary, J. G. and Witten, I. H. (1984). Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402.

Copestake, A. (1996). Applying natural language processing techniques to speech prostheses. In *Proceedings of the AAAI Fall Symposium on developing assistive technology for people with disabilities*, MIT, Cambridge, MA, USA.

Copestake, A. (1997). Augmented and alternative NLP techniques for Augmentative and Alternative Communication. In (ACL97-WS, 1997), pages 37–42.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990). *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA.

Covington, M. A. (1994). *Natural Language Processing for Prolog Programmers*. Prentice Hall, Englewood Cliffs, NJ, USA.

Cristianini, N. and Shawe-Taylor, J. (2000). *An introduction to Support Vector Machines and other kernel-based learning methods*. Cambridge University Press, Cambridge, UK.

Darragh, J. J. and Witten, I. H. (1992). *The Reactive Keyboard*. Cambridge Series on Human-Computer Interaction. Cambridge University Press.

Demasco, P. W. and McCoy, K. F. (1992). Generating text from compressed input: An intelligent interface for people with severe motor impairments. *Communications of the ACM*, 35(5):68–78.

Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society*, 39(1):1–21.

Doran, C., Egedi, D., Hockey, B. A., Srinivas, B., and Zaidel, M. (1994). XTAG system — A wide coverage grammar for English. In *Proceedings of the 17th International Conference on Computational Linguistics (COLING '94)*, pages 922–928, Kyoto, Japan.

EACL03-WS (2003). *Proceedings of the Workshop on Language Modeling for Text Entry Methods, EACL 2003*, Budapest, Hungary. Association for Computational Linguistics.

Fazly, A. (2002). The use of syntax in word completion utilities. Master's thesis, Graduate Department of Computer Science, University of Toronto, Canada.

Fazly, A. and Hirst, G. (2003). Testing the efficacy of part-of-speech information in word completion. In (EACL03-WS, 2003), pages 9–16.

Feller, W. (1968). *An Introduction to Probability Theory and its Applications*, volume 1, chapter Laws of Large Numbers, pages 228–247. John Wiley & Sons, New York, NY, USA, 3rd edition.

Francis, W. N. and Kučera, H. (1964). *Manual of Information to accompany A Standard Corpus of Present-Day Edited American English, for use with Digital Computers*. Providence, RI, USA. Online document available at `http://nora.hd.uib.no/icame/brown/bcm.html`.

Garbe, J., Kühn, M., and Lemler, K. (2001). Unbekanntes Kommunikationsobjekt im Gespräch. *Unterstützte Kommunikation*, 4/2001.

Garbe, J. U. (2001). Optimizing the layout of an ambiguous keyboard using a genetic algorithm. Diplomarbeit, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, Germany.

Garside, R. (1987). The CLAWS word-tagging system. In (Garside et al., 1987), pages 30–41.

Garside, R., Leech, G., and McEnery, A., editors (1997). *Corpus Annotation: Linguistic Information from Computer Text Corpora*. Longman, London, UK.

Garside, R., Leech, G., and Sampson, G., editors (1987). *The Computational Analysis of English: A corpus-based approach*. Longman, London, UK.

Garside, R. and Smith, N. (1997). A hybrid grammatical tagger: CLAWS4. In (Garside et al., 1997), pages 102–121.

Gazdar, G., Klein, E., Pullum, G. K., and Sag, I. A. (1985). *Generalized Phrase Structure Grammar*. Basil Blackwell, Oxford, UK.

Gleitman, L. R. and Liberman, M., editors (1995). *Language*, volume 1 of *An Invitation to Cognitive Science*. MIT Press, Cambridge, MA, USA, 2nd edition.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, USA.

Gonnet, G. H. and Baeza-Yates, R. (1991). *Handbook of Algorithms and Data Structures*. Addison-Wesley, Wokingham, England, 2nd edition.

Good, I. J. (1953). The population frequencies of species and the estimation of population parameters. *Biometrika*, 40:237–264.

Greene, B. B. and Rubin, G. M. (1971). *Automatic Grammatical Tagging of English*. Brown University, Providence, RI, USA.

Guenthner, F., Langer, S., Krüger-Thielmann, K., Richardet, N., Sabatier, P., and Pasero, R. (1993). KOMBE: Communication Aids for the Handicapped. Report 92-55, Centrum für Informations- und Sprachverarbeitung (CIS), Munich, Germany.

Harbusch, K., Knapp, M., and Laumann, C. (2001). Modelling user-initiative in an automatic help desk system. In Isahara, H. and Ma, Q., editors, *Proceedings of the 2nd Workshop on Natural Language Processing and Neural Networks (NLPNN2001)*, pages 69–76, Tokyo, Japan.

Harbusch, K. and Kühn, M. (2003a). An evaluation study of two-button scanning with ambiguous keyboards. In *Proceedings of the 7th Conference of the Association for the Advancement of Assistive Technology in Europe (AAATE 2003)*, Dublin, Ireland. To appear.

Harbusch, K. and Kühn, M. (2003b). Towards an adaptive communication aid with text input from ambiguous keyboards. In *Proceedings of the 10th Conference of the European Chapter of the Association for Computational Linguistics (EACL03), Conference Companion*, pages 207–210, Budapest, Hungary.

Harbusch, K., Kühn, M., Hasan, S., Hoffmann, H., and Schüler, B. (2003). Domain-specific disambiguation for typing with ambiguous keyboards. In (EACL03-WS, 2003), pages 67–74.

Harbusch, K., Widmann, F., and Woch, J. (1998). Towards a workbench for Schema-TAGs. In *Proceedings of the 4th International Workshop on Tree Adjoining Grammars (TAG+4)*, pages 56–61, Philadelphia, PA, USA.

Harbusch, K. and Woch, J. (2000). Direct parsing of Schema–TAGs. In *Proceedings of the 6th International Workshop on Parsing Technologies (IWPT)*, pages 305–306, Trento, Italy.

Higginbotham, D. J. (1992). Evaluation of keystroke savings across five assistive communication technologies. *Augmentative and Alternative Communication*, 8(4):258–272.

Hindle, D. (1994). A parser for text corpora. In (Atkins and Zampolli, 1994), pages 103–151.

Hobbs, J. R., Appelt, D., Bear, J., Israel, D., Kameyama, M., Stickel, M., and Tyson, M. (1997). FASTUS: A cascaded finite-state transducer for extracting information from natural-language text. In (Roche and Schabes, 1997), pages 383–406.

Horn, E. M. and Jones, H. A. (1996). Comparison of two selection techniques used in augmentative and alternative communication. *Augmentative and Alternative Communication*, 12(1):23–31.

Horstmann Koester, H. and Levine, S. P. (1994). Modeling the speed of text entry with a word prediction interface. *IEEE Transactions on Rehabilitation Engineering*, 2(3):177–187.

Horstmann Koester, H. and Levine, S. P. (1996). Effect of a word prediction feature on user performance. *Augmentative and Alternative Communication*, 12(3):155–168.

ICASSP91 (1991). *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '91)*, Toronto, Canada.

IDS, editor (1996). *Deutsche Rechtschreibung: Regeln und Wörterverzeichnis*. Text der amtlichen Regelung. Gunter Narr Verlag, Tübingen, Germany.

Johansson, S. (1986). *The Tagged LOB Corpus: User's Manual*. Norwegian Computing Centre for the Humanities, Bergen, Norway.

Joshi, A. and Hopely, P. (1996). A parser from antiquity. *Natural Language Engineering*, 2(4):291–294.

Joshi, A. K. and Schabes, Y. (1997). Tree Adjoining Grammars. In Rozenberg, G. and Salomaa, A., editors, *Handbook of Formal Languages*, volume 3, pages 69–214. Springer, Berlin, Germany.

Joshi, A. K. and Srinivas, B. (1994). Disambiguation of super parts of speech (or supertags): Almost parsing. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING '94)*, pages 154–160, Kyoto, Japan.

Jurafsky, D. and Martin, J. H. (2000). *Speech and Language Processing*. Prentice Hall, Upper Saddle River, NJ, USA.

Kaplan, R. M. and Bresnan, J. (1982). *The Mental Representation of Grammatical Relations*, chapter Lexical-Functional Grammar: A Formal System for Grammatical Representation, pages 173–281. MIT Press, Cambridge, MA, USA.

Katz, S. M. (1987). Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35(3):400–401.

Klund, J. and Novak, M. (2001). If word prediction can help, which program do you choose? Online document available at `http://trace.wisc.edu/docs/wordprediction2001/`.

Knuth, D. E. (1997). *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, 3rd edition.

Knuth, D. E. (1998). *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, 2nd edition.

Kühn, M. and Garbe, J. (2001). Predictive and highly ambiguous typing for a severely speech and motion impaired user. In Stephanidis, C., editor, *Universal Access in Human-Computer Interaction (UAHCI 2001)*, pages 933–937. Lawrence Erlbaum, Mahwah, NJ, USA.

Kushler, C. (1998). AAC using a reduced keyboard. In *Proceedings of the Technology and Persons with Disabilities Conference 1998*. Online document available at `http://www.csun.edu/cod/conf/1998/proceedings/csun98_140.htm`.

Langer, S. and Hickey, M. (1999). Augmentative and alternative communication and natural language processing: Current research activities and prospects. *Augmentative and Alternative Communication*, 15(4):260–268.

Laver, J. (1994). *Principles of Phonetics*. Cambridge Textbooks in Linguistics. Cambridge University Press, Cambridge, UK.

Lee, C.-H., Soong, F. K., and Paliwal, K. K., editors (1996). *Automatic Speech and Speaker Recognition: Advanced Topics*. Kluwer Academic Publishers, Boston, MA, USA.

Leech, G. (1997). Introducing corpus annotation. In (Garside et al., 1997), pages 1–18.

Lesher, G. W., Moulton, B. J., and Higginbotham, D. J. (1998). Techniques for augmenting scanning communication. *Augmentative and Alternative Communication*, 14(2):81–101.

Lieberman, P. and Blumstein, S. E. (1988). *Speech physiology, speech perception and acoustic phonetics*. Cambridge Studies in Speech Science and Communication. Cambridge University Press, Cambridge, UK.

Loncke, F. T., Clibbens, J., Arvindson, H. H., and Lloyd, L. L., editors (1999). *Augmentative and Alternative Communication: New Directions in Research and Practice*. Whurr Publishers, London, UK.

MacKenzie, I. S., Kober, H., Smith, D., Jones, T., and Skepner, E. (2001). LetterWise: Prefix-based disabiguation for mobile text input. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology (UIST 2001)*, pages 111–120, Orlando, FL, USA.

Manning, C. D. and Schütze, H. (2000). *Foundations of Statistical Language Processing*. MIT Press, Cambridge, MA, USA.

Marcus, M. P., Santorini, B., and Marcinkiewicz, M. A. (1993). Building a large annotated corpus of English: The Penn treebank. *Computational Linguistics*, 19(2):313–330.

Matiasek, J., Baroni, M., and Trost, H. (2002). FASTY — A multilingual approach to text prediction. In Miesenberger, K., Klaus, J., and Zagler, W., editors, *Computers Helping People with Special Needs — Proceedings of the 8th International Conference ICCHP 2002, Linz, Austria*, volume 2398 of *Lecture Notes in Computer Science*, pages 243–250, Berlin, Germany. Springer.

McCoy, K. F., Pennington, C. A., and Badman, A. L. (1998). Compansion: From research prototype to practical integration. *Natural Language Engineering*, 4(1):73–95.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63:81–97. Online document available at `http://www.well.com/user/smalin/miller.html`.

Newell, A., Langer, S., and Hickey, M. (1998). The rôle of natural language processing in alternative and augmentative communication. *Natural Language Engineering*, 4(1):1–16.

Ney, H. (1999). The use of the maximum likelihood criterion in language modelling. In Ponting, K. M., editor, *Computational Models of Speech Pattern Processing*, volume 169 of *NATO ASI Series F: Computer and Systems Sciences*, pages 259–279. Springer, Berling, Germany.

Ney, H., Martin, S., and Wessel, F. (1997). Statistical language modeling using leaving-one-out. In (Young and Bloothooft, 1997), pages 174–207.

Nietzio, A. (2002). Support vector machines for part-of-speech tagging. In Busemann, S., editor, *6. Konferenz zur Verarbeitung natürlicher Sprache (KONVENS 2002)*, pages 223–226. Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Saarbrücken, Germany.

Nilsson, N. J. (1998). *Artificial Intelligence: A New Synthesis*, chapter Heuristic Search. Morgan Kaufmann, San Francisco, CA, USA.

Pollard, C. and Sag, I. A. (1994). *Head-Driven Phrase Structure Grammar*. Studies in Contemporary Linguistics. Center for the Study of Language and Information (CSLI), Stanford. The University of Chicago Press, Chicago, IL, USA.

Purnhagen, H. (1994). *N-Best Search Methods Applied to Speech Recognition*. Diploma thesis, Universitetet i Trondheim, Norges Tekniske Høgskole, Institutt for Teleteknikk, Norway.

Rabiner, L. R. (1989). A tutorial on Hidden Markov Models and selected applications in speech recognition. In (Waibel and Lee, 1990), pages 267–296.

Rabiner, L. R., Juang, B.-H., and Lee, C.-H. (1996). An overview of automatic speech recognition. In (Lee et al., 1996), pages 1–30.

Rau, H. and Skiena, S. S. (1996). Dialing for documents: An experiment in information theory. *Journal of Visual Languages and Computing*, 7:79–95.

Roche, E. and Schabes, Y., editors (1997). *Finite-State Language Processing*. MIT Press, Cambridge, MA, USA.

Russell, S. J. and Norvig, P., editors (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, USA.

Schabes, Y., Abeillé, A., and Joshi, A. K. (1988). Parsing strategies with 'lexicalized' grammars: Application to Tree Adjoining Grammars. In *Proceedings of the 12th International Conference on Computational Linguistics (COLING '88)*, volume 2, pages 578–583, Budapest, Hungary.

Schabes, Y., Paroubek, P., and XTAG Research Group (1993). *XTAG User Manual: An X Window Graphical Interface Tool for Manipulation of Tree-Adjoining Grammars*. University of Pennsylvania, Department of Computer and Information Science, Philadelphia, PA, USA.

Schwartz, R. and Austin, S. (1991). A comparison of several approximate algorithms for finding multiple (N-best) sentence hypotheses. In (ICASSP91, 1991), pages 701–704.

Schwartz, R., Nguyen, L., and Makhoul, J. (1996). Multiple-pass search strategies. In (Lee et al., 1996), pages 429–456.

Soong, F. K. and Huang, E.-F. (1991). A tree-trellis based fast search for finding the N best sentence hypotheses in continuous speech recognition. In (ICASSP91, 1991), pages 705–708.

Sparck Jones, K. and Willett, P., editors (1997). *Readings in Information Retrieval*. Morgan Kaufmann, San Francisco, CA, USA.

Srinivas, B. (1997a). *Complexity of Lexical Descriptions and its Relevance to Partial Parsing*. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA.

Srinivas, B. (1997b). Performance evaluation of supertagging for partial parsing. In *Proceedings of the Fifth International Workshop on Parsing Technology (IWPT-97)*, Boston, MA, USA.

Srinivas, B. (2000). A lightweight dependency analyzer for partial parsing. *Computational Linguistics*, 6(2):113–138.

Srinivas, B. and Joshi, A. K. (1999). Supertagging: An approach to almost parsing. *Computational Linguistics*, 25(2):237–265.

Stolcke, A. (2002). SRILM — An extensible language modeling toolkit. In *Proceedings of the International Conference on Spoken Language Processing (ICSLP-2002)*, volume 2, pages 901–904, Denver, CO, USA.

Tanaka-Ishii, K., Inutsuka, Y., and Takeichi, M. (2002). Entering text with a four-button device. In *Proceedings of the 19th International Conference on Computational Linguistics (COLING '02)*, pages 988–994, Taipei, Taiwan.

Tapanainen, P. and Voutilainen, A. (1994). Tagging accurately — Don't guess if you know. In *Proceedings of the 4th Conference on Applied Natural Language Processing (ANLP '94)*, pages 47–52, Stuttgart, Germany.

Tech Connections (2002). Communication devices. Online document available at `http://www.techconnections.org/resources/guides/CommDevices.pdf`. Assistive Technology Quick Reference Series.

Thede, S. M. and Harper, M. P. (1999). A second-order Hidden Markov Model for part-of-speech tagging. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics*, pages 175–182, College Park, MD, USA. Association for Computational Linguistics.

van Halteren, H., editor (1999). *Syntactic Wordclass Tagging*, volume 9 of *Text, Speech and Language Technology*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Venkatagiri, H. S. and Ramabadran, T. V. (1995). Digital speech synthesis: Tutorial. *Augmentative and Alternative Communication*, 11(1):14–25.

Viterbi, A. J. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):1260–1269.

Waibel, A. and Lee, K.-F., editors (1990). *Readings in Speech Recognition*. Morgan Kaufmann, San Mateo, CA, USA.

Weischedel, R., Meteer, M., Schwartz, R., Ramshaw, L. A., and Palmucci, J. (1993). Coping with ambiguity and unknown words through probabilistic models. *Computational Linguistics*, 19(2):359–382.

Witten, I. H. (1982). *Principles of Computer Speech*. Academic Press, London, UK.

Witten, I. H., Radford, M. N., and Cleary, J. G. (1987). Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540.

Woch, J. and Widmann, F. (1999). Implementation of a Schema-TAG-Parser. Technical Report 8–99, Universität Koblenz–Landau, Computer Science Department, Koblenz, Germany.

XTAG Research Group (2001). A Lexicalized Tree Adjoining Grammar for English. Technical Report IRCS-01-03, IRCS, University of Pennsylvania, Philadelphia, PA, USA.

Young, S. and Bloothooft, G., editors (1997). *Corpus-Based Methods in Language and Speech Processing*, volume 2 of *Text, Speech and Language Technology*. Kluwer Academic Publishers, Dordrecht, The Netherlands.