Algorithms for LINGUISTIC PROCESSING

NWO PIONIER

Progress Report

August 2002

Algorithms for Linguistic Processing
NWO PIONIER
Progress Report

Leonoor van der Beek
Gosse Bouma
Jan Daciuk
Tanja Gaustad
Robert Malouf
Gertjan van Noord
Robbert Prins
Begoña Villada

bcn
Graduate School for Behavioral and Cognitive Neurosciences
Alfa-informatica
P.O. Box 716
NL 9700 AS Groningen

August, 2002

Contents

Part I

Introduction

Chapter 1

Introduction

1.1 Summary Project Goals

Algorithms for Linguistic Processing is a research proposal in the area of *computational linguistics*. The proposal focuses on problems of *ambiguity* and *processing efficiency* by investigating *grammar approximation* and *grammar specialization* techniques.

Theoretical linguistics has developed extensive and precise accounts of the grammatical knowledge implicit in our use of language. It has been able to adduce explanations of impressive generality and detail. These explanations account for speakers' discrimination between different linguistic structures, their ability to distinguish well-formed from ill-formed structures, and their ability to assign meaning to such well-formed structures. *Grammars* are hypothesised which model the well-formed utterances of a given natural language and the meaning representations which correspond with these utterances.

The smaller and younger field of computational linguistics has also been successful in obtaining results about the computational processing of language. These range from descriptions of dozens of concrete algorithms and architectures for understanding and producing language (parsers and generators), to careful theoretical analysis of the underlying algorithms. The theoretical analyses classify algorithms in terms of their applicability, and the time and space they require to operate correctly. The scientific success of this endeavor has opened the door to many new opportunities for applied linguistics.

However a number of important research problems have not been solved. An important challenge for computational accounts of language is the observed *efficiency* and *certainty* with which language is processed. The efficiency challenge is both theoretical and practical: grammars with transparent inspiration from linguistic theory cannot be processed efficiently. This can be demonstrated theoretically, and has been corroborated experimentally. In current practice, such grammars are recast into alternative formats, and are restricted in implementation. Effectively, large areas of language are then set aside.

The *certainty* with which language is processed is not appreciated generally. But careful implementation of wide-coverage grammars inevitably results in systems which regard even simple sentences as grammatically ambiguous, even to a high degree. The computational challenge is to incorporate disambiguation into processing.

There are two central leading hypotheses of the project. We shall explore *approximation* techniques which recast theoretically sound grammars automatically into forms which allow for efficient processing. The hypothesis is that processing models of an extremely simple type, namely finite automata, can be employed. The use of finite automata leads to interesting hypotheses about language processing, as we will argue below.

Second, we test the hypothesis that *certainty* can be accounted for—at least to some extent—by incorporating the results of language experience into processing. This will involve the application of machine learning techniques to grammars in combination with large samples of linguistic behavior, called corpora. Such techniques will ensure that a given utterance, which receives a number of competing analyses if considered in isolation, will receive a single analysis if the relevant context and situation are taken into account.

The project aims furthermore at significant partial results. In order to test its processing claims, large scale grammars of some theoretical ambition must be tested. While these exist now for English, the project will devote resources to extending existing Dutch grammars to further test the claims. An extensive Dutch grammar in the public domain would be a major contribution to Dutch computational linguistics and to the international community. Second, the processing techniques and concrete implementations are technology which directly enables a number of interesting applications in spoken language information systems, language instruction, linguistic research, grammar checking, and language aids to the disabled.

1.2 Adaptations in Project Goals

The general project goals have been kept quite stable over the first two years of the project. A number of changes have been made with respect to details of implementation of the original work plan.

In the original project proposal, the following sub-topics were identified:

1. Finite-state Language Processing
2. Grammar specialization (disambiguation)
3. Grammar development for Dutch
4. Linguistically-informed search tool

The first two topics were large and the last two topics were smaller, in the sense of manpower assigned to them.

It turned out that one of the reviewers for the proposed project was skeptical about the fourth item (the linguistically-informed search tool). In addition, a search tool of the type described in the original project proposal became available before the start of the project: *gsearch* (Corley et al., 2001). Both considerations led to the conclusion that it would be better to ignore this fourth theme, and instead focus on a different contribution to Dutch corpus linguistics. In particular, by providing a corpus of syntactically annotated Dutch sentences, we hope to provide a resource for theoretical linguists, corpus linguists and computational linguists that is at least as useful as the originally proposed tool. In any case, it has already turned out to be very useful for the development of the project itself.

The structure of the project can now be described as follows. Note that we have chosen to implement many of the algorithms in a single coherent framework: a natural language analysis system for Dutch, called Alpino. This system is shortly described in chapter 2, although many of the sub-components are described in more detail in later chapters. The topics of the project are as follows:

Computational Grammar Leonoor van der Beek, Gosse Bouma, Gertjan van Noord, Begoña Villada.

In section 3 we provide an overview of the efforts devoted to the construction of a computational grammar of Dutch. An important sub-task in the computational description of Dutch is related to the analysis of various types of fixed phrases. Progress in this area is described in section 4.

Annotation Efforts Leonoor van der Beek, Gosse Bouma, Robert Malouf, Gertjan van Noord.

A number of tools have been implemented to facilitate the construction of a corpus of syntactically annotated Dutch sentences. In section 5 we describe the various issues related to the construction of the *Alpino Dependency Treebank*.

Finite State Language Processing Gosse Bouma, Jan Daciuk, Gertjan van Noord, Robbert Prins.

We have worked on a variety of topics in the context of finite-state natural language processing. In section 6 a number of results in the area of efficient construction of compact finite state automata is described. These results are relevant for the construction and use of very large dictionaries and language models. In section 7 we describe a number of experiments that have been performed in the area of finite-state approximation. Chapter 8 presents a generalization of finite state automata in which labels represent predicates over symbols. This generalization is motivated by problems in natural language processing. A finite-state implementation of *Optimality Theory* is the topic of section 9. Chapter 10 describes finite-state solutions to the problem of hyphenation.

Disambiguation Tanja Gaustad, Robert Malouf, Tony Mullen, Gertjan van Noord.

In section 11 we describe a number of experiments we performed using log-linear modeling for parse selection. A number of different algorithms for parameter estimation of such log-linear models are compared in chapter 12. In section 13 we describe the progress we have made in the area of *word sense disambiguation*.

1.3 Results

The project has been running for more than two years. All of the sub-projects are now well under way. A number of research results are already established. As can be seen from the list of publications at the end of this report, not only the post-docs and senior researchers have established and published these results, but in addition the Ph.D. students have already done very useful work, and all have published (refereed) papers already.

Below we summarize the most important results.

Alpino A natural language analysis system of Dutch has been constructed which integrates various components designed in many of the sub-projects. The system therefore is very useful as a test-bed for innovations. We also use the system to construct various training sets which are then used to improve some of the sub-components of the system (*bootstrapping*). Last year, the system was chosen to be the winner of the *Battle of the Parsers* at the LOT winterschool. The system constructs dependency structures as proposed by the NWO project *Corpus Gesproken Nederlands*. An overview of the system is presented in (Bouma, van Noord, and Malouf, 2001).

Computational Grammar A wide-coverage computational grammar of Dutch has been implemented, in combination with a large lexicon. The grammar is capable of analyzing a large collection of Dutch constructions, including all frequent ones. A Dutch description of the grammar will appear as an article in the *Nederlandse Taalkunde* journal. The grammar employs a state-of-the-art analysis of non-local dependencies (Bouma, Malouf, and Sag, 2001) and Dutch er-pronouns (Bouma, 2000). The lexicon was partly derived from existing electronic resources (Bouma, 2001a; Bouma, van Eynde, and Flickinger, 2000). Preliminary results for the sub-project on the analysis of Dutch fixed phrases, in particular focusing on collocational prepositions, are reported in (Bouma and Villada Moirón, 2002).

Annotation Efforts A large collection of Dutch sentences has been annotated (semi-automatically) by CGN dependency structures. At the time of writing, about 6400 sentences from the `cdbl` (newspaper) part of the Eindhoven corpus are annotated (Uit den Boogaart, 1975). In addition, various annotation tools are implemented and integrated in Alpino. This greatly facilitates the future annotation of corpus material. The treebank is freely accessible via Internet: http://www.let.rug.nl/ vannoord/trees/. We report on this work in (van der Beek et al., 2002). A number of search tools have been developed for the treebanks; a report is presented in (Bouma and Kloosterman, 2002).

Finite State Language Processing With Lauri Karttunen and Kimmo Koskenniemi, Gertjan van Noord chaired the ESSLLI workshop on *Finite State Methods in Natural Language Processing*. The Workshop included two talks by project members (Daciuk, 2001; Bouma, 2001b). In addition, a related special event was organized entitled *20 years of two-level morphology*. Currently, van

Noord is editing (with Lauri Karttunen and Kimmo Koskenniemi) a special issue of the *Journal of Natural Language Engineering* on *Finite State Methods in Natural Language Processing.*

In the area of finite state language processing a number of algorithms have been designed which efficiently construct very compact finite state automata for the representation of natural language *dictionaries* and *language models* (Daciuk, 2000b; Daciuk et al., 2000; Daciuk, 2000a; Daciuk, 2002; Daciuk and van Noord, 2001; van Noord, 2000b).

Regular expressions are a powerful means for the construction of finite state automata. A number of extensions that are particularly useful for natural language processing were reported (van Noord and Gerdemann, 2000). We designed a more powerful variant of finite state automata in which symbols are represented by *predicates* (collaboration with Dale Gerdemann from the university of Tübingen). The properties of the new model were investigated and have been reported in an article in *Grammars* (van Noord and Gerdemann, 2001).

In the area of finite state approximation, a lexical analysis filter using a Hidden Markov Model, trained on an automatically generated annotated corpus, has been designed. It has been shown that the use of this filter is extremely effective: average parse times are up to twenty times shorter (!), whereas in addition a small increase in accuracy is observed (Prins and van Noord, 2001).

In addition, we have worked on a number of other applications of finite-state methods. With Dale Gerdemann (University of Tübingen), Gertjan van Noord gave a keynote lecture on the SIGPHON Finite State Phonology workshop, on a finite-state method for the implementation of *Optimality Theory* phonology (Gerdemann and van Noord, 2000).

We also established that finite-state techniques are applicable to the problem of hyphenation, and we investigated means of constructing finite state automata for hyphenation automatically (Bouma, 2001b).

Disambiguation In the area of word-sense disambiguation we established that the use of so-called *pseudo-words* for evaluation purposes is inappropriate (Gaustad, 2001).

In the area of parse selection for disambiguation we have made very good progress. A number of *log-linear models* have been implemented, and their success carefully evaluated. The most successful models solve about 70% of the disambiguation problem for the Alpino system. A collection of algorithms for training such log-linear models have been implemented. We showed that certain more general methods that are being employed in other areas of computing perform much better than the traditional IIS algorithm. A comparison appears in (Malouf, 2002). In log-linear models, the selection of features is an important issue. In (Mullen, 2002) various methods for *feature merging* were investigated; the experiments include an experiment with the Alpino system. This experiment was presented at the NLPRS conference (Mullen, Malouf, and van Noord, 2001).

Quantitative Evaluation

The availability of annotated corpus material enables us to monitor the progress of the Alpino system in a quantitative way. In the first published result of the Alpino system (Bouma, van Noord, and Malouf, 2001), f-scores were reported over all sentences of up to twenty words of the corpus (in as far as these were annotated at the time). The best reported score for this sub-corpus was 75%. These experiments were performed in the spring of 2001. In the mean-time we decided that *concept-accuracy* is a somewhat more reliable metric (the metric is defined in chapter 5). Concept-accuracy scores are always a little bit lower than f-scores. Even so, the latest version of the Alpino system obtains an accuracy score of 82.5% (using the default settings) on a random sample of (annotated) sentences of up to twenty words.

Chapter 2

The Alpino System

Alpino is a wide-coverage computational analyzer of Dutch which aims at accurate, full, parsing of unrestricted text. For English, tremendous progress has been made in the area of wide-coverage parsing of unrestricted text. Many of the proposed systems are statistical parsers, but systems based on a hand-written grammar exist as well. The aim of Alpino is to provide computational analysis of Dutch with coverage and accuracy comparable to state-of-the-art parsers for English.

The construction of a dependency structure on the basis of some input proceeds in a number of steps, described below. The first step consists of lexical analysis (section 2.1). An important component of lexical analysis is a Hidden Markov Model which filters unlikely lexical categories. This component is described in detail in chapter 7. The lexicon constitutes the linguistic knowledge source of this processing phase. The lexicon is described in more detail in chapter 3.

In the second step a parse forest is constructed (section 2.2), on the basis of the Alpino grammar. The grammar is described in more detail in chapter 3.

The third step consists of the selection of the best parse from the parse forest (section 2.3). The last step employs various disambiguation models. Our efforts are described in more detail in chapter 11.

To evaluate the coverage and disambiguation component of the system, a testbench of syntactically annotated material is absolutely crucial. Given the current lack of such material for Dutch, we have started to annotate corpora with dependency structures. Dependency structures provide a convenient level of representation for annotation, and a fairly neutral representation for further processing. The annotation format is taken from the project *Corpus Gesproken Nederlands* (*Corpus of Spoken Dutch*) (Oostdijk, 2000). The construction of dependency structures in the grammar and our tree-banking efforts are described in section 5. A variety of tools to help in constructing the treebank are integrated in Alpino.

The various components of Alpino are integrated in a single system which can be used in a variety of ways. Typical uses include interactive usage for grammar development or tree-banking. The graphical user interface built with Hdrug is useful here (van Noord and Bouma, 1997b). The interface provides various visualization formats for parse trees, dependency structures, feature structures, type hierarchies and derivation trees.

For evaluation purposes, or for the construction of automatically generated annotated training material, the system is employed in a non-interactive mode. Finally, a simple experimental web-interface is supported: `http://gudrun.let.rug.nl/vannoord_bin/alpino`.

An important practical problem is the huge size of the various knowledge sources that are employed (dictionary, language models). In section 6.3 a technique is presented which solves this problem.

2.1 Lexical Analysis

The lexicon associates a word or a sequence of words with one or more lexical *categories*. Such categories contain information such as part-of-speech, inflection as well as a subcategorization frame. For verbs, the lexicon typically hypothesizes many different categories, differing mainly in the subcategorization frame. For sentence (1), the lexicon produces 83 categories.

(1)	Mercedes	zou	haar	nieuwe	model	gisteren	hebben	aangekondigd
	Mercedes	should	her	new	model	yesterday	have	announced

Mercedes should have announced her new model yesterday

Some of those categories are obviously wrong. For example, one of the categories for the word `hebben` is `verb(hebben,pl,part_sbar_transitive(door))`. The category indicates a finite plural verb which requires a separable prefix `door`, and which subcategorizes for an SBAR complement. Since `door` does not occur anywhere in sentence (1), this category will not be useful for this sentence. A filter containing a number of hand-written rules has been implemented which checks that such simple conditions hold. For sentence (1), the filter removes 56 categories. After the filter has applied, feature structures are associated with each of these categories. Often, a single category is mapped to multiple feature structures. The remaining 27 filtered categories give rise to 89 feature structures.

An important aspect of lexical analysis is the treatment of unknown words. The system applies a number of heuristics for unknown words. Currently, these heuristics attempt to deal with numbers and number-like expressions, capitalized words, words with missing diacritics, words with 'too many' diacritics, compounds, and proper names.

If such heuristics still fail to provide an analysis, then the system guesses a category by inspecting the suffix of the word. A list of suffixes is maintained which predict the category of a given word. If this still does not provide an analysis, then it is assumed that the word is a noun.

In addition to the treatment of unknown words, the robustness of the system is enhanced by the possibility to skip tokens of the input. Currently this possibility is employed only for certain punctuation marks. Even though punctuation is treated both in the lexicon and the grammar, the syntax of punctuation is irregular enough to warrant the possibility to ignore punctuation. For instance, quotation marks may appear almost anywhere in the input. The corpus contains:

(2)	De	z.g.	"	speelstraat	,	die	hier	en	daar	al	bestaat	?
	The	so-called	"	play-street	,	that	here	and	there	already	exists	?

Apparently, the author intended to place `speelstraat` within quotes, but the second quote is not present. During lexical analysis, categories are optionally extended to include neighboring words which are classified as 'skip-able'.

After the system has found all possible lexical categories, a further filter is applied which removes unlikely lexical assignments. This filter is described in full detail in section 7. This filter removes many unlikely lexical categories. This speeds up the parser a lot, whereas no reduction in accuracy is observed.

2.2 Parsing

The initial design and implementation of the Alpino parser is inherited from the system described in (van Noord, 1997a), van Noord et al. (1999) and van Noord (2001).

The Alpino parser takes the result of lexical analysis as its input, and produces a *parse forest*: a compact representation of all parse trees. The Alpino parser is a left-corner parser with selective memoization and goal-weakening. It is a variant of the parsers described in van Noord (1997a). We generalized some of the techniques described there to take into account relational constraints, which are delayed until sufficiently instantiated (van Noord and Bouma, 1994).

As described in van Noord et al. (1999) and van Noord (2001), the parser can be instructed to find all occurrences of the start category *anywhere in the input*. This feature is added to enhance robustness as well. In case the parser cannot find an instance of the start category from the beginning of the sentence to the end, then the parser produces parse trees for large chunks of the input. A best-first search procedure then picks out the best sequence of such chunks. Depending on the application, such chunks might be very useful. In the past, we successfully employed this strategy in a spoken dialogue system (van Zanten et al., 1999).

2.3 Parse Selection

The motivation to construct a parse forest is efficiency: the number of parse trees for a given sentence can be enormous. In addition to this, in most applications the objective will not be to obtain *all* parse trees, but rather the *best* parse tree. Thus, the final component of the parser consists of a procedure to select these best parse trees from the parse forest.

In order to select the best parse tree from a parse forest, we assume a parse evaluation function which assigns a score to each parse. In section 11 we describe experiments with a variety of parse evaluation functions based on log-linear models.

A naive algorithm constructs all possible parse trees, assigns each one a score, and then selects the best one. Since it is too inefficient to construct all parse trees, we have implemented the algorithm which computes parse trees from the parse forest as a best-first search. This requires that the parse evaluation function is extended to partial parse trees. In order to be able to *guarantee* that this search procedure indeed finds the best parse tree, a certain monotonicity requirement should apply to this evaluation function: if a (partial) tree s is better than s', then

beam	≤ 10			≤ 20			≤ 30		
	acc (%)	msec	out	acc (%)	msec	out	acc (%)	msec	out
1	83.46	233		82.18	1004		77.66	2263	
2	83.76	243		82.59	1195		78.24	2813	
4	84.06	253		82.64	1439		78.52	3626	
8	84.02	262		82.75	1765		78.71	4976	
16	84.01	273		82.38	1718	1	78.20	6525	2
32	84.02	285		82.43	2180	1	77.34	7678	7
64	84.02	295		82.43	2992	1			
128	84.02	310		82.50	4480	1			
∞	84.02	270							

Table 2.1: Effect of beam-size on accuracy and efficiency of parse selection

a tree t which contains s should be better than t' which is just like t except it has s' instead of s. However, instead of relying on such a requirement, we implemented a variant of a best-first search algorithm in such a way that for each state in the search space, we maintain the b best candidates, where b is a small integer (the *beam*). If the beam is decreased, then we run a larger risk of missing the best parse (but the result will typically still be a relatively 'good' parse); if the beam is increased, then the amount of computation increases too. Currently, we find that a value of $b = 4$ is a good compromise between accuracy and efficiency. In table 2.1 the effect of various values for b is presented for a number of different treebanks. In the first columns, we have listed the result on all sentences of the treebank with up to ten words (1450 sentences). In the second column, we have listed the result on a random sample of sentences from the treebank of up to twenty words (415 sentences). In the third column, we have listed the result on a random sample of sentences from the treebank of up to thirty words (588 sentences). Per column, we list the concept accuracy number (this metric is explained in chapter 5) as well as the mean amount of milliseconds CPU-time per sentence, and the number of sentences for which the parser could not find an analysis due to memory limitations (in such cases the accuracy obviously is dropped too, since no correct result is constructed).

Part II

Computational Grammar

Chapter 3

Computational Grammar

3.1 Introduction

The wide-coverage computational grammar for Dutch developed within the project, known as the Alpino grammar, consists of a collection of syntactic rules, defined in terms of general principles, and a dictionary. Together, they cover a large and non-trivial part of Dutch syntax. Below, we describe the grammar formalism, the organization of the rule component, and the main principles used in the definition of grammatical rules. We also describe how the construction of the dictionary, which was to a large extent derived from existing resources.

3.2 Head-driven Phrase Structure Grammar

Head-driven Phrase Structure Grammar (Pollard and Sag, 1994; Sag and Wasow, 1999) is a linguistic theory which aims at formally explicit descriptions of natural language phenomena. As computational considerations have played an important role in the design of the formalism, the formalism is also widely in use in computational linguistics. Wide-coverage computational grammars based on HPSG exist at least for English, German and Japanese. Analyses of various aspects of Dutch grammar can be found in van Noord and Bouma (1994), Bouma and van Noord (1998), Van Eynde (1996), van Eynde (1999), and Bouma (2000).

Linguistic knowledge is represented in HPSG in terms of *attribute-value matrices* (AVM's). The dictionary consists of words which are assigned AVMs expressing the relevant linguistic properties. Examples, for the words *vrienden* (*friends*) and *vraag* (*question*) are given in figure 3.1. The AVMs are of type *noun*. This implies, among other things, that an attribute such as NFORM is defined. The value [NFORM *norm*] distinguishes between 'ordinary' nouns and expletives such as *het* (*it*) and *er* (*there*). The type *verb* licenses the attribute VFORM. The value of the attribute SC (for *subcategorization*) is a list, specifying the linguistic properties of the phrases which may occur as complements of the word. The infinitival form of the verb *achtervolgen* (*chase*), for instance, selects a direct object. The noun *vraag* may select a subordinate clause (which has to be either an indirect question introduced by a WH-constituent or by the complementizer *of* (*whether*)). As the complement clause may be extraposed, it is realized on EXTRA, instead of on SC. The attribute DT rep-

a. *vrienden*:

$$\begin{bmatrix} \textit{noun} & \\ \text{NFORM} & \text{norm} \\ \text{SC} & \langle\,\rangle \\ \text{EXTRA} & \langle\,\rangle \\ \text{DT} & \begin{bmatrix} \text{CAT} & \text{n} \\ \text{HD} & \text{vriend} \\ \text{MODS} & \langle\,\rangle \end{bmatrix} \end{bmatrix}$$

b. *vraag*:

$$\begin{bmatrix} \textit{noun} & \\ \text{NFORM} & \text{norm} \\ \text{SC} & \langle\,\rangle \\ \text{EXTRA} & \left\langle \begin{bmatrix} \textit{sbar} & \\ \text{C\_TYPE} & \text{ind-q} \vee \text{of} \\ \text{DT} & \boxed{1} \end{bmatrix} \right\rangle \\ \text{DT} & \begin{bmatrix} \text{CAT} & \text{n} \\ \text{HD} & \text{vraag} \\ \text{VC} & \boxed{1} \\ \text{MODS} & \langle\,\rangle \end{bmatrix} \end{bmatrix}$$

c. *achtervolgen*:

$$\begin{bmatrix} \textit{verb} & \\ \text{VFORM} & \text{inf} \\ \text{SC} & \left\langle \begin{bmatrix} \textit{noun} & \\ \text{CASE} & \text{acc} \\ \text{DT} & \boxed{2} \end{bmatrix} \right\rangle \\ \text{EXTRA} & \langle\,\rangle \\ \text{DT} & \begin{bmatrix} \text{CAT} & \text{v} \\ \text{HD} & \text{achtervolg} \\ \text{OBJ1} & \boxed{2} \\ \text{MODS} & \langle\,\rangle \end{bmatrix} \end{bmatrix}$$

Figure 3.1: Words and their linguistic properties, encoded in *attribute-value matrices*.

resents a *dependency tree*, which is a representation of the grammatical relations in the phrase headed by this word. The construction of dependency trees in HPSG is explained at the end of this section.

HPSG is usually seen as a (radical) lexicalist theory, i.e. as a theory which combines general rule schemata with rich and detailed lexical information. In Sag (1997), a variant of HPSG is proposed in which rules are much more construction specific. Construction specific rules are useful especially in the description of constructions which are not determined exclusively by lexical material. The grammar of relative clauses, for instance, and the analysis of headless relatives in particular, requires syntactic structures which cannot be attributed to specific lexical heads. By defining rules in terms of more general structures, and by defining structures in terms of general principles, a rule component can be defined which contains a potentially large number of specific rules, while at the same time the relevant generalizations about these rules are still expressed only once. The Alpino grammar is implemented along the lines of the proposal in Sag (1997). Apart from linguistic considerations, and important argument in favor of such an implementation is the fact that parsing using a grammar with specific rules appears to be more efficient that parsing on the basis of general rule schemata.

Almost all rules in the grammar are instances of a so-called *headed structure*. A *headed structure* consists of a mother node, a head daughter, and zero or more non-head daughters. Every *headed structure* satisfies the following principles:

- **Head-feature principle**: The HEAD features of the mother and head-daughter are unified.
- **Valence principle**: The AVM of a complement daughter must unify with the first element on SC of the head. The SC value of the mother is the SC value of the head daughter minus any selected complement daughter.
- **Filler Principle**: The AVM of a filler daughter must unify with the first element on SLASH of the head. The SLASH value of the mother is the SLASH value of the head daughter, minus any selected filler daughter. [1]
- **Extraposition Principle** : The AVM of an extraposed daughter must unify with the first element on EXTRA of the head. The EXTRA value of the mother is the EXTRA value of the head daughter, minus any selected extraposed daughter.
- **Adjunct and Dependency Principle**: The value of MODS on the mother equals the value of MODS on the head daughter concatenated with the DT value of any *modifier* daughter. The value of all other DT attributes is unified on mother and daughter.

The *head feature principle* presupposes a distinction between head features and non-head features. In standard HPSG, the distinction is implemented by grouping the head features under a single attribute HEAD. In Alpino, the head features are explicitly listed in the definition of the *head feature principle*.

A *head-complement structure* is a specialization of *headed structure*, which admits, apart from the head, exactly one dependent which acts as complement. In that case, the *valence principle* states that the value of SC on the mother equals that of SC on the head, minus the first element (which is unified with the complement daughter. As there are no filler, extraposed, or modifier daughters, the values SLASH, EXTRA and DT|MODS will be identical on the mother and head daughter.

The *head-filler*, *head-extra*, and *head-adjunct structures* are specializations of *headed structure*, which consist of a head and resp. a *filler*, extraposed or adjunct daughter. The value of resp. SLASH, EXTRA, of DT|MODS will differ between mother and head daughter, while the value of all other attributes is shared.

Most rules in the grammar are defined as instances of one of the structures just mentioned. In most rules, only the category of the mother and the daughters and their relative order needs to be specified. In the examples below, the head has been underlined:

[1]See Bouma, Malouf, and Sag (2001) for a motivation of this *head-driven* approach to extraction.

(1) a. *head-complement-structure*: $v \rightarrow np\ \underline{v}$
b. *head-adjunct-structure*: $n \rightarrow ap\ \underline{n}$

Note that the rules in (1) are instances of a *head-complement-structure* and a *head-adjunct-structure*, respectively. During compilation of the grammar, all constraints which apply, in order to satisfy the definition of these structures, are added to the rule. This leads to the rules as given in (2). The attributes NFORM and VFORM are head features. Some other head features defined for nominal and verbal categories are left out. $\langle H|T \rangle$ represents a list consisting of a head H and a tail T, $L \oplus M$ represents the concatenation of two lists L and M.

(2) a.
$$\begin{bmatrix} \textit{verb} & \\ \text{VFORM} & \boxed{1} \\ \text{SC} & \boxed{2} \\ \text{SLASH} & \boxed{3} \\ \text{EXTRA} & \boxed{4}\oplus\boxed{5} \\ \text{DT} & \boxed{6} \end{bmatrix} \rightarrow \boxed{7}\begin{bmatrix} \textit{noun} & \\ \text{EXTRA} & \boxed{4} \end{bmatrix} \begin{bmatrix} \text{VFORM} & \boxed{1} \\ \text{SC} & \langle \boxed{7} | \boxed{2} \rangle \\ \text{SLASH} & \boxed{3} \\ \text{EXTRA} & \boxed{5} \\ \text{DT} & \boxed{6} \end{bmatrix}$$

b.
$$\begin{bmatrix} \textit{noun} & \\ \text{NFORM} & \boxed{1} \\ \text{SC} & \boxed{2} \\ \text{SLASH} & \boxed{3} \\ \text{EXTRA} & \boxed{4}\oplus\boxed{5} \\ \text{DT} & \begin{bmatrix} \text{CAT} & \boxed{6} \\ \text{HD} & \boxed{7} \\ \text{MODS} & \langle \boxed{9} | \boxed{8} \rangle \end{bmatrix} \end{bmatrix} \rightarrow \begin{bmatrix} \textit{adj} & \\ \text{EXTRA} & \boxed{4} \\ \text{DT} & \boxed{9} \end{bmatrix} \begin{bmatrix} \textit{noun} & \\ \text{NFORM} & \boxed{1} \\ \text{SC} & \boxed{2} \\ \text{SLASH} & \boxed{3} \\ \text{EXTRA} & \boxed{5} \\ \text{DT} & \begin{bmatrix} \text{CAT} & \boxed{6} \\ \text{HD} & \boxed{7} \\ \text{MODS} & \boxed{8} \end{bmatrix} \end{bmatrix}$$

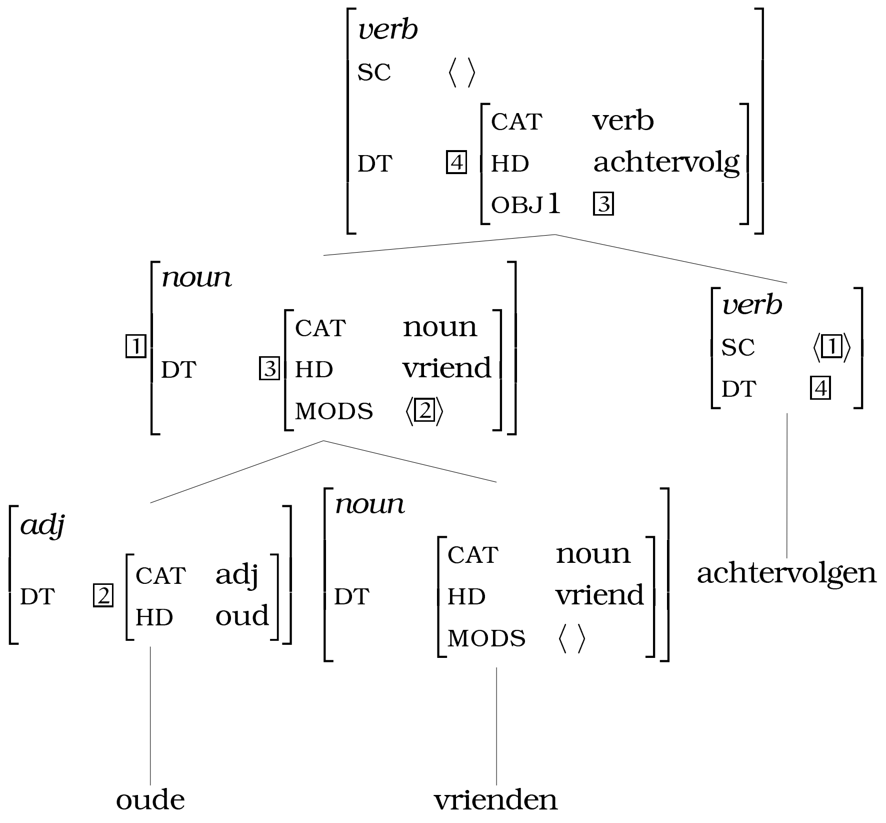
The rules license the VP in (3-a), as shown in (3-b) (irrelevant features have been suppressed). The constituent *oude vrienden* is an instance of rule (2-b), while the VP is an instance of rule (2-a).

(3) a.

(Kim blijft)	oude	vrienden	achtervolgen
(Kim continues)	old	friends	chase

Kim continues to chase old friends

b.



The rules in (4) license structures in which a verbal constituent combines with an extraposed subordinate clause.

(4) a. *head-extra-structure*: $n \rightarrow \underline{n}\ sbar$

b. $\begin{bmatrix} \textit{noun} & \\ \text{AGR} & \boxed{1} \\ \text{SC} & \boxed{2} \\ \text{SLASH} & \boxed{3} \\ \text{EXTRA} & \boxed{4} \\ \text{DT} & \boxed{5} \end{bmatrix} \rightarrow \begin{bmatrix} \textit{noun} & \\ \text{AGR} & \boxed{1} \\ \text{SC} & \boxed{2} \\ \text{SLASH} & \boxed{3} \\ \text{EXTRA} & \langle \boxed{6} \mid \boxed{4} \rangle \\ \text{DT} & \boxed{5} \end{bmatrix} \quad \boxed{6}\begin{bmatrix} \textit{sbar} \end{bmatrix}$

c. *head-extra-structure*: $v \rightarrow \underline{v}\ sbar$

d. $\begin{bmatrix} \textit{verb} & \\ \text{AGR} & \boxed{1} \\ \text{SC} & \boxed{2} \\ \text{SLASH} & \boxed{3} \\ \text{EXTRA} & \boxed{4} \\ \text{DT} & \boxed{5} \end{bmatrix} \rightarrow \begin{bmatrix} \textit{verb} & \\ \text{AGR} & \boxed{1} \\ \text{SC} & \boxed{2} \\ \text{SLASH} & \boxed{3} \\ \text{EXTRA} & \langle \boxed{6} \mid \boxed{4} \rangle \\ \text{DT} & \boxed{5} \end{bmatrix} \quad \boxed{6}\begin{bmatrix} \textit{sbar} \end{bmatrix}$

For the examples in (5-a) and (5-b), the rules above give rise to the structures in (5-c) and (5-d).

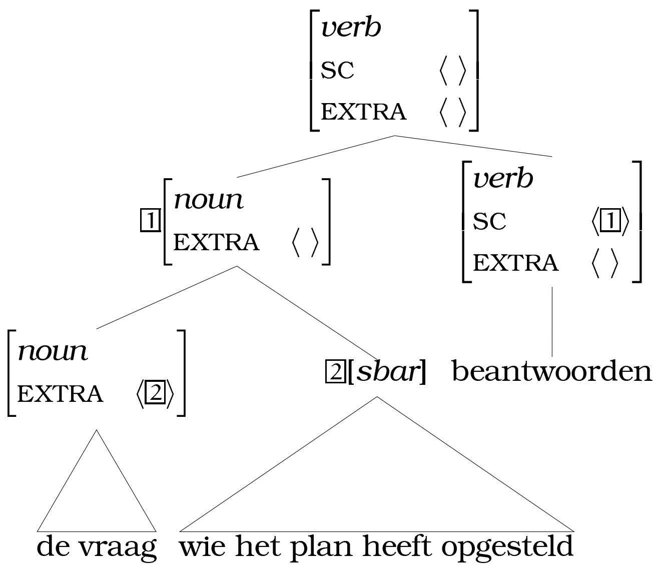
(5) a.

(Kim moet)	de	vraag	wie	het	plan	heeft	opgesteld	beantwoorden
(Kim must)	the	question	who	the	plan	has	formulated	answer

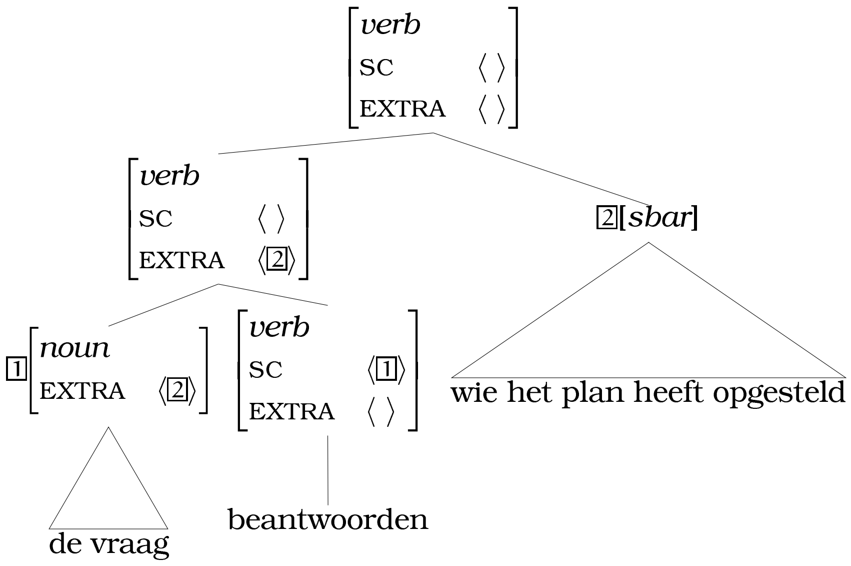
Kim must answer the question who has formulated the plan

b.	(Kim moet)	de	vraag	beantwoorden	wie	het	plan	heeft	opgesteld
	(Kim must)	the	question	answer	who	the	plan	has	formulated

c.



d.



For purposes such as parser and grammar evaluation and comparison, we have extended the grammar with a level of representation called *dependency structure*. The idea is that a dependency structure provides relatively detailed information about the syntactic structure of a sentence, while at the same time it abstracts away from all aspects of the syntactic analysis which are relevant for grammar internal purposes only. A dependency structure encodes the grammatical relations within a sentence or phrase. An example is given in 3.2. The nodes in the tree are labeled with a grammatical relation (*hd* for *head*, *su* for *subject*, *mod* for *modifier*, etc.), an optional index (the index **1** encodes that the subject of *moet* is identical to the subject of *beantwoord*), and a syntactic category. Leaves consist of a grammatical relation, a syntactic category, and the root of the word with a subscript indicating the string position, or of a grammatical relation and an index only. An important property of dependency structures is the fact that they may contain phrases which correspond to a discontinuous string at the level of word order. The dependency constituent *de vraag of Anne komt*, for instance, corresponds to an NP and an extraposed subordinate clause at the level of word order.

By being fairly general and easy to construct, dependency structures are provide an attractive level of representation for (semi-) manual annotation of test and training material. In fact, a number of syntactic treebank projects have adopted dependency relations. For us, the most important of these is the Dutch project Corpus Gesproken Nederlands. The dependency structures produced by Alpino are compatible with their standards. This has the advantage that we can build on the annotation guidelines developed within that project, while at the same time material annotated in the CGN project can be used as testing and training material for the Alpino grammar. Our own annotation efforts are discussed in more detail in chapter 5. Below, we will briefly describe how dependency structures are constructed in the grammar.

Dependency structures are constructed by adding a feature DT (for *dependency tree*) to lexical entries and rules. Verbs The DT value of a verb, for instance, specifies the grammatical relations of the complements which the verb selects for. The infinitival form *beantwoorden*, for instance, selects for a direct object (OBJ1). The lexical entry for *beantwoorden*, including its DT attribute is given in figure 3.1. The values for DT constructed by the grammar can be mapped deterministically onto proper tree-like representations as given in figure 3.2.

3.3 The rule component

The current version of the grammar contains over 270 rules. Almost half of them are *head-modifier* or *head-complement structures*. The other half consists of *head-filler-structures* (voor topicalization, WH-questions, and relatives), *head-extra-structures* (for extraposed relatives, complement clauses VPs, PPs, and comparative clauses), rules for coordination, appositions, verbal constituents introduced by a *complementizer*, and some rule to cover typical spoken language constructs.

The examples below provide a global overview of the coverage and level of detail of the rules.

Complements. The rules in (6) license verbal complements in positions left or right of the verbal head. The label *v-arg(left)* denotes the disjunction of NP, PP, and AP. The label *v-arg(right)* denotes the disjunction of $\overline{\text{S}}$, PP, en VP.

(6) a. *head-complement-structure* : $v \rightarrow$ *v-arg(left)* $\underline{v}$
   b. een boek *kopen* (*buy a book*), in Sinterklaas *geloven* (*believe in Sinterklaas*), aardig *vinden* (*consider nice, like*).
   c. *head-complement-structure* : $v \rightarrow \underline{v}$ *v-arg(right)*
   d. *geloven* dat Sinterklaas bestaat (*believe that Sinterklaas exists*), *geloven* in Sinterklaas (*believe in Sinterklaas*), *proberen* om te komen (*try to come*).

The rules in (7) define potential complements of prepositions, where *p-arg* is a disjunction over NP[NFORM *norm*], PP, AP, $\overline{\text{S}}$, and VP. Rule (7-c) cover PPs containing a preposition as well as a particle.

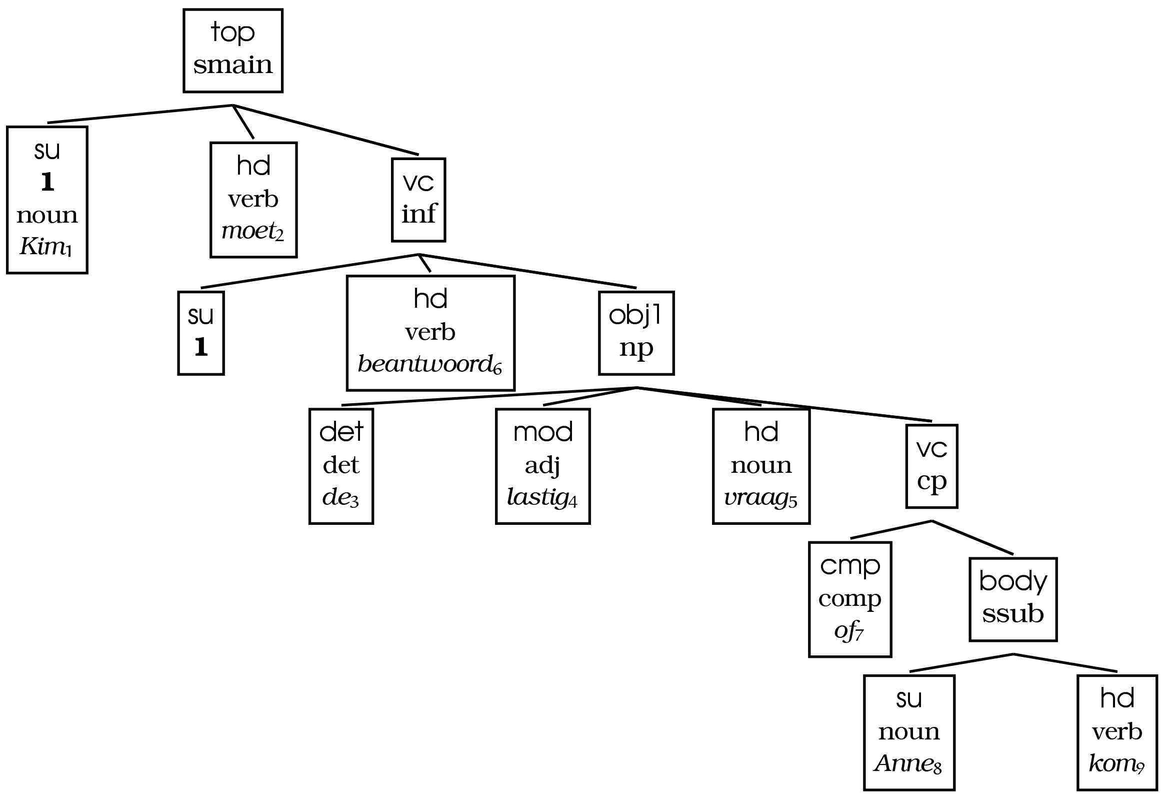


Figure 3.2: Dependency tree for the sentence*Kim moet de lastige vraag beantwoorden of Anne komt (Kim must answer the difficult question whether Anne is coming).*

(7) a. *head-complement-structure :* $v \rightarrow \underline{p}$ *p-arg*
b. *in* Groningen (*in Groningen*), *tot* aan de rand (*till at the edge*), *op* rood (*on red*), *zonder* dat het opvalt (*without that it attracts attention, i.e. without attracting attention*), *zonder* te twijfelen (*without to doubt, i.e. without doubt*)
c. *head-complement-structure :* $v \rightarrow \underline{p}$ *p-arg part*
d. *naar* Groningen toe (*to Groningen to, i.e. towards Groningen*).

Comparatives. Comparative clauses introduced by *dan* or *als*) are complements which are often not adjacent to the head selecting for such a complement. Heads which license a comparative clause introduced by *dan* are adjectives in the comparative form (8-a)-(8-c), the adjectives *meer* en *minder*(8-d)-(8-g) (which can also be used adverbially), and the nouns *(n)iets, (n)iemand), (n)ergens anders, niets/niks* (8-h)-(8-i). Heads which license an *als* comparative clause are combinations of *even + adjective* (8-j)-(8-l), *(net) zo + adjective* (8-m)-(8-o), the adverb *evenveel* (8-p), and NPs containing the determiner *hetzelfde* or *dezelfde* (8-q)-(8-s).

(8) a. deze prijs ligt *dichter* bij het bod van Bayer *dan* bij dat van Petrofina
this price is closer to the bid by Bayer than to that of Petrofina
b. de internationale fondsen waren aan het slot alle *lager dan*
the international funds were at the end all lower than
bij opening
at the opening

c. zeker nu hij tien kilo *lichter* is *dan* op de dag dat hij bij de
definitely now he ten kilos lighter is than at the day that he in the
Deventerkernploeg werd ingelijfd
Deventer-selection was incorporated

d. de gemeenten verkopen *meer* grond *dan* zij kunnen aankopen
the cities sell more land than they can buy

e. Leo is *minder* ijverig dan zijn broertje
Leo is less hard-working that his kid brother

f. hoeden worden *meer* gedragen *dan* vroeger
hats are more worn that in the past

g. een programma waarmee hij zich als artiest *minder* kon afficheren
a program with which he himself as artist less could promote
dan als vakman
than as professional

h. *Niks anders* doen *dan* almaar ruw materiaal verzamelen
Nothing else do but continuously raw material collect

i. over van Gaal *niets* *dan* lof
about van Gaal nothing but good words

j. deze koersen zijn *even* zeldzaam *als* witte raven
these races are as scarce as white ravens

k. *Even* duister en ondoorgrondelijk *als* het feit dat het kennelijk het
As dark and opaque as the fact that it apparently the
verkeerde resumé was
wrong resume was

l. *even* belangrijk *als* een goed in elkaar getimmerd partijprogramma
as important as a good together constructed party programme

m. tenminste driemaal *zo* groot *als* tien jaar geleden
at least three times as big as ten yeas ago

n. niet meer *zo* goed *als* vroeger
not anymore as good as in the past

o. maar wij zijn *net zo goed* machteloos *als* de regering en de
but we are equally without power as the government and the
Verenigde Naties
United Nations

p. Uitgeschakeld worden voor het Jaarbeursstedentoernooi zou
Eliminated being for the Jaarbeurs-cities-tournament would be
evenveel betekenen *als* niet meer meetellen in het internationale voetbal
equal meaning as no longer count in the international soccer

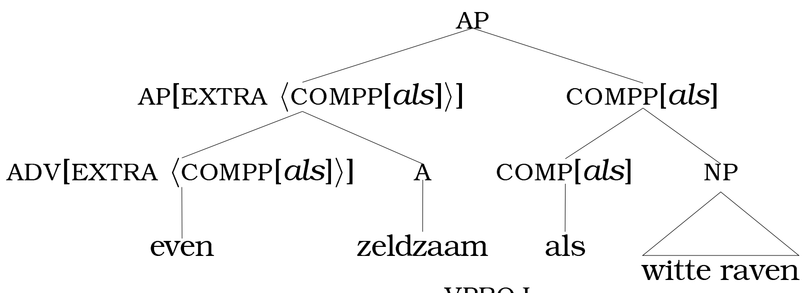
q. jonge mensen uit *dezelfde* leeftijdscategorie *als* de werkende jongeren
young people from every age category as the working youngsters

r. Dat was *dezelfde* *als* gisteren
that was the same as yesterday

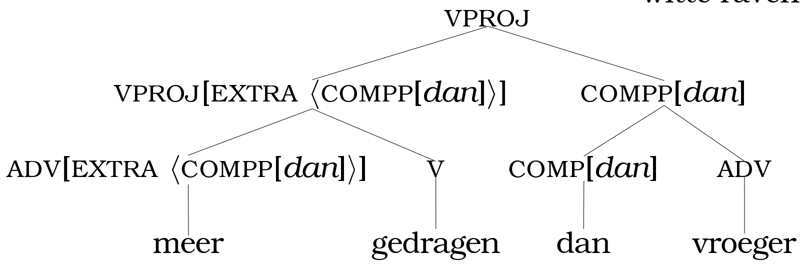
s. Daar zat *hetzelfde* idee achter *als* bij 't aderlaten
there was the same idea behind as with the bleeding

In the grammar it is assumed that lexical elements which may license a comparative clause contain an EXTRA value which contains a comparative phrase (COMPP). As extraposed comparative clauses may be adjoined to AP, NP, and VP nodes, we predict that extraposition is possible within an (predicative) adjectival constituent (9-a) and within a VP (9-b).

(9) a.



b.



The comparative clause itself consists of the complementizer *als* or *dan* followed by an NP, adverb, subordinate clause, VP, A (*witter dan wit*) or PP.

Note that this analysis uses the fact that all rules are subject to the extraposition principle, and thuis instantiate the feature EXTRA. To implement our analysis of extraposition of comparative phrases, we only have to assume that adjectives in their comparative form as well as a number of specific lexical items select for a comparative clause via their EXTRA attribute. The extraposition analysis subsumes cases where the complement is actually adjacent to the licensing head (*lager dan bij de opening, niets dan lof*) while at the same time it predicts that (non-vacuous) extraposition is obligatory with heads such as *even* and *zo* (**even als witte raven zeldzaam*, **zo als vroeger goed*).

Modification of nouns. In (10) we present rules for modification of nouns. We distinguish between modification by an adjective, PP, or relative clause on the one hand, and appositions on the other. The latter introduce a dependency relation APP, and are therefore licensed by a *head-app-structure*. The category *app_n* defines which nouns may occur as heads in constructions such as *een zak aardappelen* (*a bog (of) potatos*).

(10) a. *head-adjunct-structure : n* $\rightarrow$ $\underline{n}$ *pp*
b. *familie* uit Amsterdam (*family from Amsterdam*)
c. *head-adjunct-structure : n* $\rightarrow$ $\underline{n}$ *rel*
d. *familie* die niemand kent (*family who nobody knows*)
e. *head-adjunct-structure : np* $\rightarrow$ $\underline{np}$ *post-np-adv*
f. *Beerta* senior/alleen/zelf/ook (*Beerta senior/only/himself/also*), *2 februari* aanstaande (*2nd (of) february next*),
g. *head-adjunct-structure : n* $\rightarrow$ *pn* $\underline{n}$

h. Chevrolet *programma* (*Chevrolet programme*)
i. *head-app-structure : n* $\rightarrow$ *$\underline{\text{app\_n}}$ n*
j. een *zak* aardappelen (*a bag (of) potatos*), het *medium* film (*the medium movie*),
k. *head-app-structure : n* $\rightarrow$ *$\underline{n}$ np*
l. de *familie* Balemans (*the family Balemans*), de *Oostenrijker* Hermann Nitsch (*the Austrian Hermann Nitsch*), de *hoofdstad* Luxemburg (*the capital Luxemburg*)

Genitive Partitives. The following examples contain a special form of modification, as they consist of a noun followed by an adjective with genitive -*s* inflection:

(11) a. 't Is me *wat moois*
It is me something special
This is quite something!
b. Dat belooft *niet veel goeds*
That promises not much good
That does not look very promising
c. *Wat voor stoms* heb je nu weer uitgehaald?
What for stupidity have you now again been up to
What stupidity have you been up to this time?
d. Het is *niets bijzonders*
It is nothing special

There is a small class of nouns which license this construction: *iets, wat, niets, niks, niet veel, weinig, genoeg, allerlei, meer* and *wat voor*). In the lexicon they are defined as being of type *iets_n* (12-a). This type only occurs in the rule for genitive partitives (12-b). Apart from an *iets_n* noun the rules requires an adjectives constituent where the head has a genitive -*s* ending. The genetive inflected forms of adjectives are constructed during compilation of the lexicon and are distinguished from attributive and predicative forms by the AFORM attribute, which takes as value *iets* in this case (12-c).

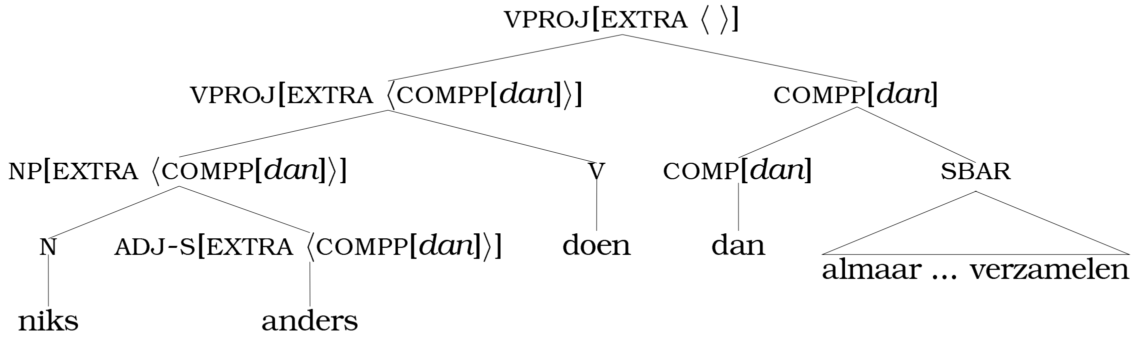
(12) a.
$$\textit{iets}: \begin{bmatrix} \textit{iets\_n} & \\ \text{AGR} & \text{thi\&indef\&sg} \\ \text{SC} & \langle\,\rangle \\ \text{NFORM} & \text{norm} \end{bmatrix}$$
b. *head-adjunct-structure: np* $\rightarrow$ *$\underline{\text{iets\_n}}$ iets_adj*
c.
$$\textit{lekkers}: \begin{bmatrix} \textit{adj} & \\ \text{AFORM} & \text{iets} \end{bmatrix}$$

Although the genetive construction above has a a number of idiosyncratic aspects, our analysis still uses general principles and lexical categories as much as possible. For instance, as the adjectives in this construction are only distinct from regular adjectives in their AFORM value, we predict that coordination of genetive forms is possible, using the general rule for adjective coordination:

(13) invallen, waarin ritme en melodie samenvloeiden tot
ideas in which rhythm and melody unite to
iets moois maar grilligs
something beautiful but capricious

The interaction of construction specific rules with general principles is illustrated in (14-a). The NP*niks anders* is analyzed as a genitival partitive. As the rule licensing such structures is an instance of a *head-adjunct-structure*, the value of EXTRA is passed on as required. Without additional stipulations, it is therefore predicted that the comparative clause license by *anders* can be extraposed.

(14) a. *Niks anders* doen *dan* almaar ruw materiaal verzamelen
Nothing else do but continuously raw material collect

b.


Other rules. Instead of describing in detail some of the more involved aspects of Dutch syntax, we will give some references to earlier work which has been used as the basis of our implementation. Following Koster (1975), we assume a relationship between finite verbs in first or second position in main clauses and the VP-final position which contains finite verbs in subordinate clauses. A non-transformational approach to expressing this relationship is discussed in van Noord et al. (1999). The analysis of Dutch verb clusters follows the analysis for German based on *argument inheritance* in Hinrichs and Nakazawa (1994) and applied to Dutch in Bouma and van Noord (1998) and van Noord and Bouma (1997a). The most important difference with the proposal in Bouma and van Noord (1998) is the fact that the Alpino grammar does not make use of *linear precedence constraints*. Thus, word order is encoded by means of ordinary rewrite rules. The computational burden of using *linear precedence constraints* in this case was less attractive than the fact that some generalizations are missed in the present implementation. Topicalization, constituent questions, and relative clauses use the analysis of extraction as proposed in Bouma, Malouf, and Sag (2001). The only difference is that Alpino does not treat the selection of adjuncts lexically, and thus, extraction of adjuncts is also implement in a non-lexicalist fashion. Again, computational considerations are the reason for adopting this solution.

3.4 The dictionary component

Accurate, wide-coverage, parsing of unrestricted text requires a lexical component with detailed subcategorization frames. A lexicon that is incomplete in this respect can seriously degrade parser performance. Carroll and Briscoe (1996) observe, for instance, that for their initial system *the largest source of error on unseen input is the omission of appropriate subcategorization values for lexical items (mostly verbs).*

Lexical databases providing subcategorization information are rare and therefore researchers have focused on the question of how to obtain such information automatically from raw or annotated text. For Dutch, the tools or corpora to do automatic acquisition are not available. On the other hand, at least two lexical resources (The CGN/Celex database and the Parole dictionary) provide detailed information concerning the syntactic valency of lexical items. In this section, we address the question to what extent using these lexical resources can lead to an adequate initial lexical component for the Alpino grammar. Below, we explain to what extent detailed dependency frames can be extracted from the two existing, general-purpose, lexical resources abd how the extracted information is incorporated in Alpino, We also provide an indication of the coverage of the resulting lexicon.

3.4.1 Acquisition of Dependency Frames

For lexicalist grammar formalisms, the availability of lexical resources which specify subcategorization frames is crucial. In HPSG, for instance, phrase structure schemata rely on the fact that each head contains a specification of the elements it subcategorizes for. If such specifications are missing, the grammar will wildly overgenerate.

Furthermore, to create lexical entries with dependency relations, the subcategorization information provided by the lexical database must be relatively detailed. For instance, to distinguish between a direct and indirect object, either a distinction between accusative and dative case must be made (for which there is no morphological evidence in Dutch), or the relevant dependency label must be provided explicitly. To distinguish between PP-complements with the prepositional or locative/directional complement relation, detailed semantic information or an explicit dependency label must be provided.

Lexica with subcategorization information are often not available or have very limited coverage, and therefore researchers have attempted to extract the relevant information from unannotated corpora automatically (Brent, 1993; Carroll and Rooth, 1998; Briscoe and Carroll, 1998; Schulte im Walde et al., 2001). While this has the potential advantage of giving frequency information for subcategorization, it also has the drawback that considerable energy has to be spent on creating a (shallow) parser able to recognize with sufficient accuracy the relevant syntactic configurations. Acquisition of subcategorization information from a syntactically annotated corpus is much more straightforward, leading mainly to questions whether a dependent is to be counted as a selected argument or an adjunct (Collins, 1999; Sarkar and Zeman, 2000), but obtaining reasonable coverage requires large corpora.

3.4.2 Using Existing Resources

Currently the resources required to do automatic extraction of dependency frames for Dutch are not available. However, two lexical resources exist which provide dependency frames. These have been used to create a lexicon for the Alpino Grammar with detailed subcategorization and dependency information for verbs and nouns. An overview of the 10 most frequent dependency frames for verbs in the Alpino lexicon is given in table 3.1. All dependency frames are also defined for verbs containing a separable verb prefix. The lexicon was too a large extent derived from existing resources. Below, we describe the verbal entries in both resources.

Celex (Baayen, Piepenbrock, and van Rijn, 1993) is a large lexical database for Dutch, with rich phonological and morphological information. For use within the project *Corpus Spoken Dutch* (CGN), this database has been extended with dependency frames (Groot, 2000). The dependency labels provided by the Celex/CGN database are intended to support the syntactic annotation of the material collected in the CGN project. As we adopted the CGN guidelines for syntactic annotation (i.e. dependency trees with specific dependency relations assigned to the constituents), the information in the Celex/CGN database can be used directly for constructing lexical entries compatible with the Alpino grammar.

Some key figures are given in table 3.2. Note that there is considerable variation in the distribution of dependency frames. A large number of frames is associated with only a few verbs, with 300 dependency frame types being associated with only a single verb.

The Dutch Parole lexicon[2] has been created as part of a project aiming at the development of uniform lexical and corpus resources for a number of European languages. The Parole lexicon comes with detailed subcategorization information, but dependency relations differ from those in the CGN proposal. Key figures are given in table 3.3.

While the mapping from Parole dependency frames into the CGN dependency frames is mostly straightforward, there are also a number of problematic cases. The ADV dependency relation in Parole, for instance, has no obvious corresponding dependency relation in CGN, although manual inspection leads us to suspect that in many cases it corresponds to the LD (*locative/directional* complement) relation. Currently, verbs with dependency frames containing the ADV relation are not extracted. Another notable difference between the two sources is the relatively small number of intransitive verbs in Parole. This is partly related to the ADV dependency relation in Parole. Adverbial elements are often optional and subject to wide variation (i.e. adverbial PPs are not restricted to a small set of *pforms*, and adverbial dependents can often be both adverbs and PPs. However, even if these elements are counted as true modifiers (and thus not as part of the subcategorized-for dependents of the verb), the number of intransitives remains relatively small.

3.4.3 Verbal entries in the Alpino lexicon

Dependency frames for the verbal lexicon of the Alpino Grammar have been constructed using the dependency information provided by CGN/Celex, Parole, and by entering definitions by hand. The latter has been done mostly for auxiliary and modal verbs, a small class of high-frequent elements which are exceptional in a number of ways. The CGN/Celex dictionary is exceptionally large. As the Celex

[2]http://www.inl.nl/corp/parole.htm

Dependency frame	Roots	Example
[SU:NP][OBJ1:NP]	3438	zij *aanvaardt* het plan *she accepts the plan* hij *bakent* het plan *af* *he marks out the plan*
[SU:NP]	2158	zij *aarzelt* *she hesitates* hij *barst los* *he explodes*
[SU:NP][LD:PP⟨*pform*⟩]	1389	zij *arriveert* in Groningen *she arrives in Groningen* hij *blijft weg* uit Groningen *he stays away from Groningen*
[SU:NP][PC:PP⟨*pform*⟩]	1271	zij *ageert* tegen het plan *she agitates against the plan* hij *barst* in tranen *uit* *he bursts into tears*
[SU:NP][OBJ1:NP][LD:PP⟨*pform*⟩]	1013	zij *aait* hem over de bol *she caresses him over the head* hij *brengt* de kinderen *onder* bij de buren *he takes the children to the neighbours*
[SU:NP][OBJ1:NP][PC:PP⟨*pform*⟩]	855	zij *achtervolgt* hem met het plan *she chases him with the plan* hij *bereidt* haar op het plan *voor* *he prepares her for the plan*
[SU:NP][OBJ1:SDAT]	418	zij *aanvaardt* dat het plan mislukt *she accepts that the plan fails* hij *biecht op* dat het plan mislukt. *he confesses that the plan fails*
[SU:NP][OBJ2:NP][OBJ1:NP]	314	zij *belemmert* hem de doorgang *she blocks the passage for him* hij *biedt* haar het plan *aan* *he offers her the plan*
[SU:SDAT][OBJ1:NP]	274	dat het plan kan mislukken *benauwt* haar *that the plan might fail bothers her* dat het plan mislukt *brengt* onrust *teweeg* *that the plan fails causes distress*
[SU:NP][SE:NP][PC:PP⟨*pform*⟩]	248	zij *baseert* zich op dit plan *she uses the plan as a basis* hij *geeft* zich *over* aan de politie *he surrenders himself to the police*

Table 3.1: The 10 most frequent verbal dependency frames in the Alpino dictionary. Frames are specified as a list of complements, where complements are specified as function:category. The LD relation denotes a locative of directional complement, and SE denotes an inherently reflexive complement.

11800	Total number of verbal stems
21800	Total number of dependency frames
650	Dependency frame types
300	Unique dependency frame types
6574	[SU:NP][OBJ1:NP]
4188	[SU:NP]
1161	[SU:NP][LD:PP⟨*pform*⟩]
1021	[SU:NP][PC:PP⟨*pform*⟩]
826	[SU:NP][OBJ1:NP][LD:PP⟨*pform*⟩]
549	[SU:NP][OBJ1:NP][PC:PP⟨*pform*⟩]
408	[SUP:⟨het⟩][OBJ1:NP][SU:SDAT]
341	[SU:NP][OBJ1:SDAT]
275	[SU:NP][OBJ2:NP][OBJ1:NP]
274	[SU:NP][SE:NP]

Table 3.2: Key figures and the 10 most frequent dependency frame types for the CGN/Celex lexical database. (*Pform* is a placeholder for various preposition forms. SUP is the relation names for expletive subjects SDAT is the category for subordinate clauses introduced by the complementizer *dat*).

database comes with frequency information, we currently only include those lexical items whose frequency is above a certain threshold. For verbal stems, this means that roughly 50% of the stems in Celex is included in the Alpino lexicon. All verbal stems from the Parole lexicon with a dependency frame covered by the grammar are included.

Extraction of verbs with a specific dependency frame from Celex and Parole requires that a particular frame in the database is identified and given a definition in the Alpino Grammar. Currently, for 28 different CGN/Celex dependency frames a definition in the grammar has been provided. This covers over 80% of the verbal dependency frames in the CGN/Celex database, 10,400 of which are sufficiently frequent to be included in the Alpino lexicon. For 15 different dependency frames in the Parole lexicon a definition in Alpino is present. Using these, we extract over 4,100 dependency frames.

As CGN/Celex is the larger database, one might suspect that this database is more exhaustive than Parole. However, the union of the frames extracted from CGN/Celex and Parole contains 11,700 frames, which means that Parole contributes 13% of the frames in the Alpino lexicon. An overview of overlap and non-overlap for the most frequent frames extractable from both sources is given in table 3.4.

For transitive and intransitive verbs, we see that over 85% of the stems in Parole are present in CGN/Celex as well. For most other dependency frames, however, the overlap is generally much smaller, and a significant portion of the stems present in Parole is not present in Celex. This suggests that, for more specific subcategorization frames, both resources are only partially complete, and that not even the union of both provides exhaustive coverage.

3200	Total number of verbal stems
5000	Total number of dependency frames
320	Dependency frame types
190	Unique dependency frame types
1566	[SU:NP][OBJ1:NP]
474	[SU:NP][PC:PP⟨*pform*⟩]
378	[SU:NP][ADV:PP⟨*pform*⟩]
208	[SU:NP][OBJ1:NP][OPT:PC:PP⟨*pform*⟩]
205	[SU:NP]
204	[SU:NP][ADV:ADV]
204	[SU:NP][OBJ1:NP][OPT:ADV:PP⟨*pform*⟩]
163	[SU:NP][OBJ1:NP][PC:PP⟨*pform*⟩]
107	[SU:NP][SE:NP][PC:PP⟨*pform*⟩]
101	[SU:NP][VC:S⟨subordinate,dat⟩]

Table 3.3: Key figures and the 10 most frequent dependency frame types for the Parole lexical database. Notation has been made conformant with the CGN/Celex notation where possible. Optional complements are marked OPT.

As we are currently only using the most frequent 50% of the CGN/Celex database in the Alpino lexicon, we also compared Parole with the complete CGN/Celex database. Here we found that the absolute number of dependency frames goes up dramatically only for transitive and intransitive verbs, and that practically all intransitive and transitive Parole stems are included in the full CGN/Celex database. For the other dependency types, however, the figures are comparable to those given in table 3.4. The relatively high number of transitive and intransitive verbal stems in Parole also present in Celex is therefore probably due to the fact that in Celex these are assigned as a default to most verbal stems. This also explains why the low frequency verbs consist almost exclusively of stems with transitive or intransitive dependency frames.

A more direct method to establish coverage of the lexicon is to see to what extent the dependency frames present in a treebank are covered by the lexicon. For a small dependency treebank, annotated according to the format presented in section 2, we extracted all verbal heads, together with their non-modifier dependents. Sets of dependents were identified with specific dependency frames. For instance, if a verb occurred with an NP subject and a PP with the PC dependency relation and *prep* as head, it is assumed that this verb must be associated with the `[SU:NP][PC:PP⟨`*prep*`⟩]` dependency frame. Coverage can now be tested by counting how often a dependency frame in the treebank also occurs in the lexicon. Extraction of dependency frames is mostly straightforward. Problematic cases are those where one dependency frame is more general than another. For instance, a verb occurring with a VP-dependent introduced by the complementizer *om* might be associated with a dependency frame selecting for an *om*-VP, but also with a more general dependency frame selecting for a VP (with or without complementizer). In such cases, we check whether at least one of the potential frames occurs in the

Dependency Frame	Overlap	Celex only	Parole only	Total
[SU:NP][OBJ1:NP]	1810	1211	240	3261
[SU:NP]	257	1697	42	1996
[SU:NP][PC:PP⟨*pform*⟩]	337	541	273	1151
[SU:NP][OBJ1:NP][PC:PP⟨*pform*⟩]	129	375	308	812
[SU:NP][VC:S⟨subordinate⟩]	103	136	103	342
[SUP:NP⟨het⟩][OBJ1:NP][SU:CP]	7	247	5	259
[SU:NP][OBJ2:NP][OBJ1:NP]	65	171	28	264
[SU:NP][SE:NP][PC:PP⟨*pform*⟩]	65	62	102	229
[SU:NP][SE:NP]	49	137	65	251
[SU:NP][VC:VP]	10	16	37	63

Table 3.4: Dependency Frames and the number of stems occurring with this frame in both resources, in CGN/Celex only, in Parole only, and the total number of stems with this dependency frame in the resulting Alpino Lexicon.

lexicon.

We applied the evaluation method described above to a treebank, constructed for grammar evaluation purposes, consisting of 424 short sentences (up to 10 words) selected from the Eindhoven-corpus (Uit den Boogaart, 1975), with a total of just over 2,200 words. The test-set contained 473 verbal heads, 417 of which (88%) occurred in a dependency configuration which was also present in the lexicon. Although one obviously would like to obtain figures from a larger test-set, we believe that this is an encouraging result. Carroll and Briscoe (1996), for instance, report that in a small test set 12% of sentences failed to parse due to missing subcategorization information in their ANLT lexicon (which is comparable in size to our lexicon, and contains subcategorization information extracted automatically from a learners dictionary). Coverage seems higher than what can be achieved by methods based on automatic extraction of subcategorization frames. Briscoe and Carroll (1997), for instance, estimate a token recall (i.e. the percentage of true positives of the learned frames in a corpus) of 81%.

We have extracted dependency frames for nouns, but have not carried out a systematic evaluation for these dependency frames. Currently, we are extracting almost 2.000 dependency frame tokens for nouns selecting prepositional complements, more than 1.000 dependency frame tokens for nouns selecting verbal (infinitival or finite sentential) complements, and over one hundred frames for measure nouns and titles (*vice-president Jansen*).

3.5 Grammatical Coverage

We performed various experiments to gain insights in the performance of the grammar and disambiguation component of the Alpino system. The quantitative results of these experiments are discussed elsewhere in this report. A manual, qualitative, error analysis learns that the most important linguistic phenomena which are problematic for the grammar are:

- Ungrammatical sentences and spelling errors:

(15) a. Het EEGtoporgaan heeft besloten WestDuitsland zijn beperkende maatregelen moet opheffen.
The EEG-institute has decided (that) West-Germany must remove its restrictions.
b. Ruim dertig percent van de tientallen *mijoenen* Japanse tv-kijkers slaat nooit een aflevering over.
Over 30% of the tens of mi(l)lions Japanese tv-viewers never misses a show

- Unknown words not properly analyzed by the heuristics:

(16) Langzaamaan werden we bekend.
Slowly, we became known

The word *langzaamaan* was erroneously analyzed as a noun.

- Complex compounds:

(17) a. de 8 procent staatsleningen
the 8% loans
b. de 4 x 200 meter ploeg
the 4 x 200 meters team

- Interjections and other inserted material:

(18) a. De Nachtwacht van Rembrandt kun je, *plus hondje*, in levende lijve tegenkomen in Berg en Terblijt.
Rembrand's Nachtwacht you may, with dog, encounter alive in Berg and Terblijt
b. Weinig goeds, zo heeft Leo Ferre ondervonden in Vichy.
Not much good, so Leo Ferre has experienced in Vichy
c. "Ook over de wijk waar ik zelf woon (Buitenveldert in Amsterdam) worden de meest krasse veroordelingen uitgesproken."
About the part of town where I live myself (Buitenveldert in Amsterdam) one hears the most outrageous statements.

- Missing lexical valency frames:

(19) Dit jaar *ziet* men zich al voor problemen gesteld.
this year sees one oneself already for problems posed
this year, one already sees oneself confronted with problems

The verb *zien* apparently selects of an inherent reflexive as well as a past participle phrase in this case.

- PP-complementation of nouns and extraposition and topicalization of PP-complements selected by nouns:

(20) a. Bij de minister werd veel *begrip* gevonden *voor* de bij de vakbeweging levende wensen.
The minister showed much sympathy for the demands of the union
b. *Op* bijna alle brieven hebben we geen *reacties* ontvangen.
At almost all letters, we got no response
c. *Van* 't woonhuis bleef een groot *gedeelte* gespaard.
Of the house, a large part remained intact
d. *Van* dat alles bleef *niets* heel.
Of it all, not much remained intact

- Recognizing predicative modification (*bepaling van gesteldheid*):

(21) *Gebouwd op een wielbasis van 2,95 m* is de wagen 23 cm korter dan een Impala Coupé en ook 10 cm smaller
Built on a basis of 2.95 meters, the car is 23 cm shorter than the Impala Coupé and 10 cm less wide.

- *Floating quantifiers*

(22) De kostelijke ladingen werden *allemaal* afgekeurd, [. . .]
The valuable cargos were all rejected

- *Clefts*:

(23) a. Het zijn overigens niet de eerste plaatsen die haar het meeste plezier hebben gedaan.
It are, by the way, not the first places which gave her most pleasure
b. Het zijn de spellen zelf die de gemoederen in beweging brengen.
It are the games themselves which cause a commotion

- Elliptic constructions, some coordinations, extraposed conjuncts:

(24) a. Kantfluweel en in combinatie met lurex is zijn bescheiden doorkijkmateriaal voor de avond
velvet and, in combination with, lurex is his modest transparent material for the evening
b. De vertegenwoordigers van het gas- en electriciteitsbedrijf zouden vandaag en die van de mijnwerkers overmorgen hun stakingsplannen bekend maken [. . .]
The representatives of the gas and electricity companies would announce their plans today and the miners the day after tomorrow
c. Er worden bloembakken gewenst, goede gordijnen, en sfeervolle verlichting.
One wants flower pots, good curtains, and fancy lights

3.6 Future Work

We have already made significant progress towards the goal of developing and implementing a wide-coverage grammar of Dutch. In the remainder of the project, we will focus our attention at a number of specific issues which we think will increase the coverage of the grammar even further.

First of all, lexical coverage can still be improved. The most important issue for our lexicalist grammar is to improve the coverage of valency patterns. Using existing resources, we managed to construct a lexicon which covers approximately 88% of the valency patterns of verbs found in real text. We hope to improve on this number by deriving valency patterns from annotated treebanks (i.e. our own material as described in chapter 5 and the syntactically annotated part of the CGN) and by means of automatic acquisition of valency from larger (but unannotated) corpora.

Second, idiomatic, more or less fixed, expressions turn up very frequently in the treebank. At the moment, we only deal with a fraction of these expressions. To improve coverage of such expressions, we expect that the lexicon needs to be extended with the relevant lexical items, and that in some cases the grammar needs to be adapted in order to account for the fact that some idiomatic expressions exhibit irregular syntax. The work on idiomatic expressions is described in more detail in the next chapter.

Finally, we hope to improve on our coverage of coordination and elliptical constructions. The error analysis in the previous section pointed out that complicated forms of coordination, as well as most aspects of ellipsis, are outside the scope of the current grammar. Below, we outline how we hope to make progress on the treatment of ellipsis in the grammar.

The resolution of elliptical structures requires a method for determining which constituents in one part of an utterance have to be considered as elided in a second part of the utterance. In addition, we have to determine which constituent a non-elided phrase is parallel with. Our goal is to determine a list of parallelism constraints that contains (1) the hard constraints that have to be met in order to have a grammatical elided phrase and (2) the soft constraints that reflect the probability of certain reconstructions of elided phrases. These constraints should be ordered according to their violability. In a followup, the hard constraints should be implemented in a grammar for elliptical structures and the soft constraints should be implemented in a probabilistic postprocessor. We expect that these soft constraints are applicable in other contexts as well, e.g., for the disambiguation of ‘ordinary’ coordination.

In the literature on ellipsis, some constraints can be found. These are hard constraints, mainly on English elliptical structures. Dalrymple, Shieber, and Pereira (1991) for example illustrate the existence of parallelism constraints with some examples. The examples they give of parallelism constraints are stativeness of verbs (25-a), pleonasticity of nouns (25-b) and the constraint on depth of embedding (25-c). These are constraints on elements in the source clause and their corresponding elements in the target clause. Kehler adds to this two constraints on parallelism between the elided phrase and the antecedent in the source clause in symmetric coordinations (Kehler, 1994). The two elements should be of the same voice (25-d) and the antecedent of an elided verbal phrase should be verbal as well (25-e). However, these constraints on English elliptical structures can not straightforwardly be translated to Dutch. For example, in Dutch the constraint on identical voice is not valid (26).

(25) a.*Dan likes gold and George is too
b.*It is raining and George is too
c.*The major of Washington left, and New York did too
d.*The decision was reversed by the FBI, and the ICC did too [reverse the decision]
e.*This letter provoked a response from Bush, and Clinton did too [respond]

(26)

Deze	week	is	de	oude	president	vertrokken	en	de	nieuwe	benoemd
this	week	is	the	former	president	left	and	the	new	appointed

This week, the former president left and the new one was appointed

From the standard work of Dutch descriptive grammar, the following hard constraints on Dutch ellipsis and parallelism can be extracted (Haeseryn and et al., 1997):

- an elided verb does not have to agree with the antecedent in person or number

(27)

Jullie	komen	vandaag	en	ik	morgen	pas
you	come	today	and	I	tomorrow	only

You are coming today and I will only come tomorrow

- an elided noun does not have to agree with the antecedent in number

(28)

Hij	koopt	twee	boeken	en	ik	een
he	buys	two	books	and	I	one

He buys two books and I buy one book

- in forward contraction, only constituents are elided, except for measure phrases or wh-heads.

(29) a.*

Piet	ligt	onder	de	bank	en	Marie	ligt	op
Piet	lies	under	the	couch	and	Marie	lies	on

Piet lies under the couch and Marie lies on the couch

b.

Piet	koopt	een	pond	andijvie	en	Marie	een	kilo
Piet	buys	a	pound	endive	and	Marie	a	kilo

Piet buys a pound of endive and Marie buys a kilo of endive

(30)

Piet	weet	hoeveel	jongens	er	meegaan	en	Marie	hoeveel	meisjes
Piet	knows	how-much	boys	there	go-with	and	Marie	how-much	girls

Piet knows how many boys are coming with us and Marie knows how many girls are

- if a finite verb is a parallel element, the verbal and predicative complement must be realized as well

(31) *Wij zullen jullie niet kunnen helpen en jullie zullen ons ook niet kunnen
we will you not can help and you will us too not can
We won't be able to help you and you won't be able to help us either

(32) *Ik was gisteren ziek en jij was eergisteren
I was yesterday ill and you were the-day-before-yesterday
Yesterday I was ill and the day before you were ill

- in a subordinate source clause with a subject and a direct object, a noun phrase in the target clause can only be parallel with the subject in the source clause if the object in the target clause is parallel with some clause too

(33) a.*Hij zegt dat hij tulpen haat en zij
he says that he tulips hates and she
He says that he hates tulips and she too
b. Hij zegt dat hij tulpen haat en zij rozen
he says that he tulips hates and she roses
He says that he hates tulips and that she hates roses

Furthermore, the following soft constraints are mentioned:

- In both main clauses and subordinate clauses, parallelism of noun phrases with subjects is dispreferred

(34) Jan vindt Marie schuldig en Frans onschuldig
Jan finds Marie guilty and Frans not-guilty
preferred: Jan judges Marie guilty and Jan judges Frans not guilty
dispreferred: Jan judges Marie guilty and Frans judges Marie not guilty

- A non-stressed pronoun is preferred not to be parallel to a noun phrase

(35) Jan geeft me een stuiver en Piet een dubbeltje
Jan gives me a penny and Piet a dime
Jan will give me a penny and Piet will give me a dime

This also illustrates the fact that a certain ranking exists in the soft constraints: the first one mentioned above is in conflict with the second one, but apparently, the second is stronger (in (35), the subject is parallel, which is dispreferred by the first soft constraint).

We expect to find more constraints in the Dutch literature on elliptical structures. Another important source of information is corpora. The examples of ellipses that we have already collected and that we will find after a corpus search provide a good source of information on what elliptical structures can be found. Information from corpora is especially important for finding soft constraints and ranking them, for only corpora can provide information on the use of different types of ellipses and their frequency.

The next step would be to rephrase the constraints in terms that are compatible with an HPSG framework. We hope to be able to make generalizations over the observed data. During this formalization, we will have to keep in mind that the account is to be implemented in Alpino. On the one hand, this means that we have to think about the computational aspects of our account. On the other hand, the Alpino grammar as it is right now contains information that can be used in parallelism constraints. For instance:

- an adverb is preferred to be parallel with an adverb with the same value for the feature TMPLOC, which specifies if it is temporal or locative. Else, it is preferred not to be a parallel element at all

For evaluation of our account, we will again use a corpus. The Alpino Dependency Treebank already contains enough occurrences of ellipsis to give a first impression of the quality of an analysis. For a thorough quantitative analysis, a test suite of elliptical structures should be constructed.

Chapter 4

Modification in Dutch semi-fixed phrases

4.1 Overview

Phrases that exhibit irregular syntax and semantics pose difficulties for a computational parser. Currently, the Alpino parser fails to find a correct syntactic representation (parse) for idiomatic expressions due to missing lexical entries and subcategorization frames of frequent verbs (e.g. *hebben*) that may also be part of idiomatic phrases.

(Semi-)fixed phrases exhibit idiosyncratic behavior in their syntactic distribution, and they may often have two different semantic interpretations: a literal and an idiomatic one.

Saying that fixed phrases exhibit idiosyncratic syntax means that it cannot be easily predicted when an idiomatic expression undergoes passivization, raising, insertion of modification, 'it-cleft', etc. This supports the argument that fixed phrases cannot always be built by regular grammar rules (Sailer, 2000; Riehemann, 2001).

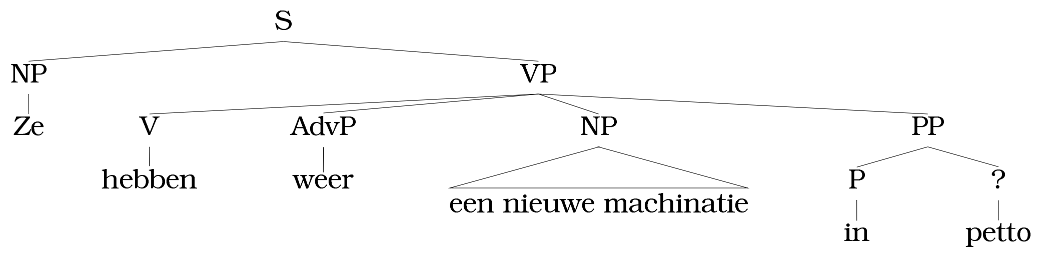
To distinguish which interpretation is the correct one, a parser may assign a different syntactic representation to a sentence depending on which semantic interpretations the sentence entails.

To illustrate this, consider example (1).

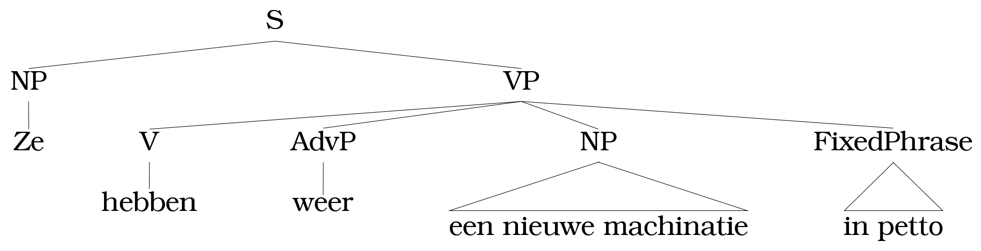
(1)	Ze	hebben	weer	een	nieuwe	machinatie	in	petto
	they	have	again	a	new	plot	in store	

'They have a new plot in store.'

A computational parser will, most likely, assign the syntactic representation in (2) to the sentence in (1); the parser will take *een nieuwe machinatie* as direct object of *hebben* and *in* as head of an adjunct PP. Unfortunately, the word *petto* would not be recognized at all (see (2)). *Petto* does not occur in other expressions in the language and, therefore it needs to be included in the lexicon. It is unclear what word category should be assigned to *petto*. Since *petto* seems to be the complement of the preposition *in*, let us assume that *petto* is a noun even though, *petto* shows no modification, no plural morphemes nor determiners. If some words inside the sentence are missing in the lexicon, parsing of the sentence might fail.[1]

(2)


However, ensuring parse completion is not simply a matter of expanding the lexicon with all words found in a given corpus. In (1), *in petto* together with *hebben* form an unsaturated 'phrase' whose meaning is 'to have something (surprising) in reserve'. Independently of the syntactic representation we choose, the phrase *in petto hebben* should be assigned meaning as if it were a single lexical entry in a dictionary; on the other hand, in syntax, a parser should not treat the string *in petto hebben* as a single constituent because *in petto* is often separated from *hebben* (1). Past research on idiomatic expressions in language engineering proposed that the fixed complement be entered as a 'multi-word lexeme' in the lexicon (Breidt, Segond, and Valetto, 1996). For the time being, we adopt this solution and label *in petto* as a special *fixed phrase* selected by the verb *hebben* and thus, inserted as a multi-word unit in a lexicon. A possible syntactic representation is given in (3).

(3)


In petto hebben is one of many fixed expressions in Dutch which shows idiosyncracies in morphology, syntax and semantics. These idiosyncracies of fixed expressions require that fixed expressions be treated differently from regularly built phrases. If we treat fixed expressions as regular phrases, a parser used for natural language generation would allow such expressions in any syntactic context and with unlimited modification (overgeneration); on the other hand, treating fixed expressions as totally fixed units would not allow modification or distributional variation of those fixed expressions that allow passivization, adjectival modification, etc. (undergeneration). Ultimately, we want to avoid problems of both, over- and undergeneration.

Fixed expressions are difficult to formalize in a computational grammar. Because of their non-uniform syntactic properties and distribution, fixed phrases are treated as lexicalized phrases included in the lexicon. This measure helps to constrain the syntactic contexts of fixed phrases.

Work done on idiomatic expressions in theoretical linguistics proposed to treat complement-like phrases selected by the verbal head in an idiomatic expression as

[1]Section 2.1 describes the heuristics used to guess word categories of unknown words.

a *fixed* lexicalized unit. This ensured that some grammar rules apply only to *fixed* phrasal units which combined with a given verb license an idiomatic expression. If either complete fixed expressions (VPs) or their fixed constituents were to be formalized as a fixed multi-word-unit, we should have evidence that no internal modification, no extraction nor any sort of variation is possible.

We carried out a case study that investigated the automatic identification and extraction of Dutch *voorzetsel uitdrukkingen* (Paardekooper, 1962) such as the ones in (4). *Voorzetsel uitdrukkingen* correspond to phrases that exhibit the pattern [preposition NP preposition] and their meaning is not always compositional. Often the NP is realized by an abstract noun without any quantifiers or modifiers (4); this noun sometimes exhibits idiosyncratic morphology. Because of these features, one possibility is to consider these 'fixed' expressions as lexicalized multi-word units and add them as such in the lexicon.

One might suppose that these strings are proper instances of fixed expressions and therefore, need to be separate from other slightly more flexible expressions. However, our case study revealed that limited modification is possible within *voorzetsel uitdrukkingen.*[2] In fact, some expressions claimed to be totally fixed can be split by adverbial modifiers or sometimes, discourse markers (5).[3,4]

(4) op advies van, op initiatief van

(5) a. De teneur is dat Camus 'niets aan actualiteit heeft ingeboet', *in tegenstelling mischien tot* andere 'existentialisten' in het naoorlogse Parijs ...
b. Economisch rekenen kunnen de ambtenaren al 47 jaar, *in tegenstelling dus tot* de dames en heren van ...
c. *met dank natuurlijk aan* Shakespeare
d. *met dank ook aan* Juan Moredo Ramos en ...

Supposed 'fixed' chunks within fixed expressions are not always totally fixed. Flexibility needs to be allowed inside the 'fixed' phrases. Fortunately, recent work on idiomatic expressions emphasize the need to allow for variation within the idiomatic constituents (Riehemann, 1997; Sailer, 2000; Sag et al., 2001).

We have presented the framework that prompts the need to study modification within fixed phrases in Dutch. We have briefly explained why fixed phrases are problematic for a computational parser. Finally, we tried to show the need to look into modification. In section 4.1.1 we enumerate the goals and objectives of this study.

4.1.1 Goals and objectives

The Phd project being described here aims to understand the grammar of fixed and semi-fixed expressions in Dutch, particularly their potential for modification.

To achieve a thorough understanding of allowable modification within (semi-)fixed phrases, we aim at

[2]Section 4.4 gives a complete description of this case study.
[3]Previously, Paardekooper (1973) pointed this out.
[4]The Dutch data used in this section is taken from *De Volkskrant op CDROM 1997*; otherwise the original source will be explicitly cited.

- identifying several classes of fixed phrases in naturally occurring Dutch text. Our initial targets are four types of fixed phrases: (i) [PP copular verb], (ii) [NP copular verb], (iii) [*laten* infinitive VP] and (iv) support verb constructions (verbs: *hebben*, *maken*, *houden* and *doen*).

- determining what sort of modification (if any) is allowed within fixed phrases and whether some sub-classes emerge regarding the allowable modification

- motivating the lexical representation of fixed phrases in the lexicon and the grammar rules and constraints that will license them in the appropriate contexts

In short, we aim to propose an accurate lexical representation with the necessary linguistic constraints affecting the modification of the mentioned types of fixed phrases in Dutch. To achieve these goals we will investigate and develop statistical models to automatically extract from large corpora the 4 types of fixed phrases studied. The output of the statistical model should facilitate the statement of generalizations about the presence of modification in the phrases at stake.

Questions we seek to answer are:

- What data-driven models are useful to automatically acquire fixed expressions?

- How do models improve if the extraction data is annotated with rich linguistic information? What features or linguistic information are most helpful?

- Applying corpus-based techniques, can we discover what modification is allowed within fixed phrases and infer the linguistic description?

- In order to handle modification within ‘fixed’ expressions what features and linguistic constraints need to be specified in a computational grammar?

With this investigation we aim at providing: (i) a characterization of the 4 types of Dutch fixed phrases, by proposing the required lexical representation and grammatical description under a lexicalist grammar framework; (ii) a feasible statistical model for their extraction and identification and (iii), evaluation of the extraction models and evaluation of the improvement in the Alpino parser’s coverage.

4.2 Fixed expressions

Before attempting the formalization of the linguistic analysis of fixed expressions in the Alpino grammar two requisites need to be satisfied: (i) enumeration of linguistic properties and constraints that model fixed expressions and (ii) a detailed description of the syntactic behavior of fixed expressions.

4.2.1 Terminological remark

The terminology used in the literature to refer to *idiomatic expressions* (Sailer, 2000) is most varied. Terms such as *idiom* (Katz, 1973; Schenk, 1994; Riehemann, 2001), *fixed expression* (Moon, 1998), *idiomatic word combination* (Nunberg, Sag, and Wasow, 1994), *multi-word lexeme* (Breidt, Segond, and Valetto, 1996), *collocation* (Krenn, 2000) and *multi-word-expression (MWE)* (Sag et al., 2001), all somehow subsume the type of objects that will be investigated.

A uniform definition of *fixed expression* is difficult to provide,[5] the reason being that there is not yet a well-established enumeration of their properties and characteristics nor an exhaustive classification of these linguistic entities (Sailer, 2000; Krenn, 2000; Riehemann, 2001; Sag et al., 2001).

A common feature of *fixed* and *semi-fixed* phrases is the mutual lexical selection that takes place between the lexemes in the phrase. Krenn (2000) proposes 'lexical selection' as the property of collocational expressions in which word co-occurrence is determined by lexical rather than semantic criteria.

The difference between *fixed* and *semi-fixed phrase* pertains to their syntactic flexibility and also their semantics. A *fixed phrase* is characterized by having completely rigid syntax which does not allow internal modification (adjectives, adjunct prepositional phrases nor relative clauses) or extraction. Examples are *in petto* 'in store', *ten opzichte van* 'with respect to' or *per slot* (in Dutch) and *by and large* (English). Nevertheless, larger fixed phrases may exhibit tense inflection of the head verb.

Semi-fixed phrases may allow insertion of (limited) modification, quantification, morphological variation and lexical rules such as passivization, raising, topicalization may apply. *Semi-fixed phrases* may have a compositional or non-compositional semantics. The more syntactic flexibility allowed in the constituents of a *semi-fixed phrase*, the more compositional it is (Nunberg, Sag, and Wasow, 1994; Sailer, 2000)(cf. Abeille (1995)). A tricky aspect observed in *semi-fixed* phrases pertains to the fact that not all of them appear in the same syntactic contexts or allow the same degree of flexibility.

4.2.2 Data

This section gives a general description of the facts about fixed phrases that the analysis needs to account for. The purpose is to highlight relevant properties of fixed phrases that distinguish them from regular phrases. In the course of this project, we aim at gathering more observations on the syntactic variation and distribution of fixed expressions by using data-driven methods. These observations will be used as the basis of the classification of fixed phrases and the deep linguistic analysis of fixed expressions in the Alpino grammar.

Non-homomorphism The argument structure of the main predicate in a sentence is often used in lexicalist grammars to build the skeleton of the syntactic repre-

[5]I avoid using the term *idiom* because it often refers to a phrase that exhibits non-compositional semantics (Perlmutter and Soames, 1979; Culicover, 1976). In the remainder the term *fixed expression* refers to the group of *fixed* and *semi-fixed phrases*.

sentation of such sentences. The predicate argument structure of an idiomatic expression does not always mirror the syntactic constituency of the verbal head (non-homomorphism). An example will illustrate this better.

The Dutch expression *uit de weg* 'out of the way' may combine with the verb *gaan* 'to go' regularly, where *gaan* has an intransitive use (6).

(6) Als John uit de weg gaat . . .
    if John out of the way goes
    'If John goes out of the path . . . '

The phrase *uit de weg* also combines with *gaan* in a different context (idiomatic use) where *gaan* is transitive (7). In this case, the PP *uit de weg* is not simply a locative adjunct since, removing the phrase *uit de weg* brings up ungrammaticality (8). The interpretation of the expression in (7) differs from that of (6).

At a descriptional level, non-homomorphism between the argument structure of the main predicate in a fixed expression and its syntactic valence is a consequence of the fact that the fixed expression exhibits non-compositional semantics.

Non-compositionality The figurative meaning 'John avoided the problems' conveyed by the expression in (7) cannot be derived from the individual meanings of the constituents inside the phrase.[6] The interpretation of the expression is not totally compositional.

(7) John ging de problemen uit de weg.
    John went the problems out of the way
    'John avoided the problems.'

(8) * John ging de problemen
    John went the problems

A few tests proposed by Sailer (2000) serve to check the idioms' semantic non-compositionality. First, the noun inside the phrase *uit de weg* cannot be replaced by a synonym since the whole sentence becomes nonsensical (9). Second, adding an adjective or a relative clause inside the phrase *uit de weg* brings up ill-formedness (see (10)) and (11) respectively).

(9) *John ging de problemen uit de straat/het pad
    John went the problems out of the street/the path

(10) *John ging de problemen uit de goede/gladde weg

(11) * John ging de problemen uit de weg die naar een plein leidde
    John went the problemen out of the way that to the square led

In spite of the restricted semantics of the phrase *uit de weg* shown above, the phrase can occur in a related semantic context next to the verb *ruimen* 'remove' (12). It seems reasonable to conclude that, the meaning of the phrase *uit de weg* in

[6]Notice that the glossed literal translation makes no sense in English. This is not surprising since some idioms cannot be translated compositionally (Schenk, 1994).

the expression is not completely opaque and some parts of the VPs *iets uit de weg gaan* and *iets uit de weg ruimen*, still carry meaning.

(12) Graag wil ik even een misverstand uit de weg ruimen.
Willingly want I just a misunderstanding out of the way to-remove
'I would like to get rid of a misunderstanding'.

Often, internal modification is not admitted inside the phrase *uit de weg* without altering the original figurative meaning of the whole expression (10),(11). Nevertheless, addition of internal modification inside idiom constituents is possible in some cases (13)). It is commonly agreed that in semantics the adjective modifies the whole idiom (13-a),(13-b) (Nicolas, 1995; Abeille, 1995). However, claims that idiomatic parts with internal modification necesitate a referent are well-founded given examples (in German) like (13-c) taken from Fischer and Keil (1996) or in Dutch (13-d).[7]

(13) a. John kicked the *social* bucket
b. make *rapid* headway (taken from Nicolas (1995))
c. Tom hat auf der Sitzung einen *gross*en Bock geschossen
Tom has on the meeting a big buck shot
'Tom made a big mistake in the meeting.'
d. Dupuis gooit in het debat over de gezondheidszorg *verschillende*
Dupuis throws in the debate about the healthcare different
knuppels in het hoenderhok
sticks in the hen-house
'Dupuis dropped a bombshell in the debate about healthcare.'

Examples in (13) show that fixed expressions allow insertion of modification; furthermore, such modifiers alter the idiomatic meaning, therefore parts of fixed expressions need to be assigned meaning in a formalization of fixed expressions.

The existence of instances of *uit de weg* with similar meaning but combined with a different verb (*ruimen*) prove that this constituent is not semantically opaque. Clearly, semantic transparency of a constituent does not necessarily imply that the meaning of that constituent contributes to the idiomatic interpretation of the whole expression.[8,9] This points to a difference between literal expressions and some idiomatic ones. The interpretation of literal expressions follows from combining the meaning of each constituent via general combinatorial semantic rules. In contrast, a large number of fixed phrases introduce obligatory (syntactic) constituents that are 'meaningless' in the argument structure of the main verb due to the predicate's

[7]In (13-a), *bucket* does not have a referent of its own; *social* modifies the complete idiomatic denotation of the VP; the sentence means 'John disappeared from the social spheres.' In contrast, *gross* in (13-c) modifies only the idiomatic meaning of *Bock*, that is, 'mistake'.

[8]Schenk (1994) highlights the importance of keeping semantic transparency separate from compositionality. The former implies that the meaning of a word inside a phrase is literal at an observational level; the latter pertains to grammar internal mechanisms to derive the meaning of word combinations.

[9]In the expression *Peter sawed logs all night* both 'sawed' and 'logs' are semantically transparent. However, in the idiomatic interpretation combining 'sawed' with 'logs' gives 'snored'. The semantically transparent meaning of each constituent does not transfer to the idiomatic interpretation of the idiom. The idiom is not compositional.

non-compositional nature.

See examples of non-compositional idioms in English (14).

(14) a. John's father kicked the bucket ('John's father died')
b. They kept tabs on John ('They observed John')
c. They pulled John's leg ('They teased John')

We want to emphasize that non-compositionality is not a constant property of fixed expressions given the existence of idiomatic expressions like (15).

(15) a. Marie spilled the beans ('Marie revealed the secret')
b. Wij zitten in het vaarwater van andere onderzoeken
we stand in the way of other research
'We are poaching on other research's territory.'
c. De bondskanselier zelf heeft nog niet in zijn kaarten laten kijken
The federal chancellor in person has yet not in his card let look
'The chancellor plays his cards close to his chest.'

A syntactico-semantic treatment needs to handle this non-homomorphism between syntactic constituency and semantic argument structure. Such approach also requires that individual lexemes of compositional fixed expressions be assigned a semantic variable since they may accept modification (*in elkaars vaarwater*) and their specifiers may participate in scope relations. On the other hand, the idiomatic meaning of non-compositional fixed expressions may not be split among its individual lexemes but, in syntax some mechanism needs to allow insertion of modification that modifies the meaning of the expression as a whole (13-a).

Syntactic versatility Non-compositionality in deriving the semantic interpretation of idioms has been used to explain irregularities in their syntactic distribution (Nunberg, Sag, and Wasow, 1994). Diagnostic tests such as topicalization, extraction, passivization, raising, control, pronominalization, etc. are often applied to determine whether certain constituents behave as regular constituents in a non-idiomatic sentence. The more syntactic rigidity shown by phrases, the more evidence is gathered to account for their non-compositional nature. We explore whether the constituents inside a fixed phrase behave as other phrases in normal contexts.

Topicalization refers to the realization of a constituent (other than the subject in SVO languages) in initial position in a sentence. The referent of a topicalized constituent receives emphasis and therefore it is assumed that it must carry semantic content. Applied to our example fixed phrase *uit de weg gaan* the object NP may occur in topic position (16). The fixed part could occur in topic position if we enrich the context as in (17) (cf. (18).[10]

(16) Een rechtstreekse aanval op de vorstin ging Colijn echter uit de weg
A direct attack on the queen went Colijn out the way
'A direct attack on the queen, Colijn avoided.'

[10]Some speakers find (18) acceptable if negation follows the matrix verb.

(17) Colijn ondernaam weinig actie; maar uit de weg ging Colijn
Colijn undertook little action; but out the way went Colijn
een rechtstreekse aanval op de vorstin echter wel
a direct attack on the queen ...
'Colijn undertook little action; but he avoided a direct attack on the queen.'

(18) * Uit de weg ging Colijn een rechtstreeks aanval op de vorstin echter
Out the way went Colijn a direct attack on the queen ...

Here, the focus lies on the syntactic properties of fixed expressions, therefore, we simply assume that the fixed part of the idiom may be topicalized independently of it carrying meaning on its own or not (cf. Schenk (1994)).[11] Nunberg, Sag, and Wasow (1994) observe that Dutch and German allow topicalization of idiomatic constituents that are part of non-compositional idioms; however, they argue that since the topicalized constituent receives no emphasis, it is only viewed as a syntactic phenomenon. This property points at a similar behavior between idiomatic and non-idiomatic expressions in syntax (even if the topicalized sentence is really marked).

The object NP may promote to subject of a related passive sentence (19), (20). This shows that the object NP conveys meaning on its own and also that this use of *gaan* is transitive. Thus, the argument structure of the idiomatic use requires at least one more dependent than the argument structure of the (intransitive) literal use and it resembles transitive predicates.

(19) Principiële keuzes worden uit de weg gegaan
Essential choices were out the way gone
'Fundamental choices were avoided.'

(20) Harde overtredingen werden niet uit de weg gegaan
Strong offences were not out the way gone
'Strong offences were not avoided.'

A fixed complement (*gevolg*) may also be subject of a passive sentence (21). Similarly, some fixed constituents can realize the subject of a raising construction (22) or the subject of a control VP (23).[12] This also shows that idiomatic phrases may occur in embedded contexts.

(21) Maar er werd nooit gevolg aan gegeven.
But there was never consequence on given
'But they never suffered any consequences.'

(22) (...) omdat er schot lijkt te zitten in de
...... because it progress seems to sit in the arrangements
regeling (...)
...

[11]The decision whether a topicalized fixed constituent carries meaning or not, is crucial in a syntax-semantics analysis since one needs to assign the (formal) semantic representation to the topicalized constituent. In the example *Marie's hart brak Piet* from (Schenk, 1994, ex.67), part of the topicalized constituent obligatorily carries meaning.

[12]Example found at `http://www.auburn.edu/student_info/plainsman/archives/97FA/1009/front.html`.

'. . . because it seems that there has been progress in the arrangements . . . .'

(23) The lawsuit over contested seats on the Auburn Board of Trustees may be over, but now *the piper wants to be paid.*

The fact that some fixed phrases cannot occur in certain distributional contexts is not enough evidence to argue that fixed phrases are always non-compositional. There are fixed phrases whose semantic interpretation may be derived compositionally and in syntax, they may exhibit restrictions on their syntactic distribution. *uit de weg gaan* is syntactically rigid but its semantics is rather compositional.

Flexibility within sentence boundaries The constituents inside the phrases *zich ten doel stellen* ('to set as a goal') and *op zoek zijn/gaan naar*('seek') may occur in different locations, thus ruling out a linear fixed ordering of the constituents. The fixed part *ten doel* or *op zoek* may immediately follow the reflexive NP (24), occur at the end (25),(26), precede the verb in nonfinite embedded contexts (27),(28),(29) or be separated from the finite verb by an adverbial (30),(31).

(24) Hij stelde zich ten doel Nederlands talent te verzamelen.
'He set as a goal to bring together Dutch talent.'

(25) We stellen ons tegelijkertijd academische uitnemendheid ten doel.
'At the same time we aim at achieving academic perfectness.'

(26) Als ik het doe, probeer ik het zo goed mogelijk, stelde De Boer zich vervolgens tot doel.
'After this De Boer set the following as a goal: if I do it, I'll try my very best.'

(27) Nissan heeft zich ten doel gesteld de productiviteit jaarlijks met 10 procent te verhogen.
'Nissan aims at a 10 per cent increase of its productivity each year.'

(28) Ik heb gehoord dat vijftien geheime agenten naar mij op zoek zijn.
'I have heard that 15 secret agents are looking for me.'

(29) Naar een pure resultaattrainer zijn we niet op zoek gegaan.
'We've started looking for a new top-trainer.'

(30) Ze zou slechts op zoek zijn naar huisvesting in een rustiger, veiliger deel van de stad
'She would only be looking for living space in a quieter, safer part of town.'

(31) Joep is naarstig op zoek naar nieuw en geschikt personeel voor zijn helpdesk.

'Joep is in a hurry to find a new and suitable personnel for his helpdesk.'

Pronominalization NPs inside PP complements of fixed phrases may be pronominalized into an R-pronoun. In the support verb construction *last hebben van* ('have trouble with') (32), the NP complement of *van* may pronominalize into an R-pronoun. If pronominalization has taken place, the R-pronoun may be realized next to the preposition (one word) (33) or separate (34).

(32) 'Wij hebben minder last van de slechte Russische infrastructuur dan
we have less trouble from the bad Russian infrastructure than
andere bedrijven', zegt pr-manager.
other business said pr-manager
"We have less trouble with the Russian infrastructure than other companies", said the pr-manager.'

(33) Wij hebben minder last ervan dan andere bedrijven.
we have less trouble it-from than other companies
'We have less trouble from them than other companies.'

(34) Wij hebben er minder last van dan andere bedrijven.
we have it less trouble from than other companies
'We have less trouble from them than other companies.'

A last example (35) shows that wh-extraction of the NP object of a preposition is possible; the preposition occurs separate from its complement.

(35) waar zat ik ook alweer mee in mijn maag ?
what sit I also with in my stomach ?
'What concerned me a lot?.'

A difficult aspect in every study of idiomatic expressions is to find generalizations on their syntactic distribution. On one hand, some fixed phrases only occur in particular syntactic contexts (e.g. *het pleit is/wordt beslecht* ('the argument was settled')). On the other hand, constraints observed in normal non-idiomatic expressions also hold for idioms (Katz, 1973; Abeille, 1995; Nunberg, Sag, and Wasow, 1994). For example, passivization applies to transitive verbs (but also impersonal verbs); control is sometimes possible if the fixed constituent denotes an animate referent, etc.

A uniform account of the syntax of fixed phrases cannot be easily formalized given that not all fixed phrases exhibit the same degree of flexibility nor occur in the same syntactic contexts. The analysis of fixed expressions needs to plan an adequate lexical representation of the lexemes involved, and to state what constraints and grammar rules license the syntactic distribution.

It is worth mentioning a few remarkable characteristics of some fixed phrases in Dutch. Our aim is to present evidence that suggests that the syntactic relations between lexemes which are part of idiomatic expressions are complex, not simply selectional preferences.

Idiosyncratic morphosyntax Another feature of some Dutch fixed phrases is the presence of frozen words that only occur in such type of phrases in the language. One example is *iets in petto hebben* ('to have something in store'); *in petto* exclusively occurs with the verbs *hebben* and *houden*, nowhere else in the language. This suggests that some enforced co-occurrence restrictions take place between the phrase and these two verbs.

Petto derives from Italian *petto* ('heart') (Geerts and Heestermans, 1992). It is a loanword that, according to its morphological variations in Dutch, seems not very productive. Assuming that *petto* is a noun, it shows many irregularities: addition of determiners is not possible (37), it fails to admit any type of modification (38), the noun cannot be replaced by a synonym (39) and, it has no plural counterpart. The phrase *in petto* is required by the verb; removing *in petto* changes the meaning of the transitive verb *hebben* (40).

(36)	Ze	hebben	weer	een	nieuwe	machinatie	in petto
	they	have	again	a	new	plot	in store

'They have a new plot in store.'

(37) *Ze hebben weer een nieuwe machinatie in een petto

(38) *Ze hebben weer een nieuwe machinatie in veilig petto

(39) *Ze hebben weer een nieuwe machinatie in zak

(40)	Ze	hebben	weer	een	nieuwe	machinatie
	they	have	again	a	new	plot

'They have a new plot.'

At the sentence level *in petto hebben* exhibits similar restrictions to topicalization, passivization, raising and control as *uit de weg gaan.* The difference between *uit de weg gaan* and *in petto hebben* thus pertains to the idiosyncratic morphology affecting the 'noun' inside the fixed constituent.

Idiosyncratic morphosyntax is also found in *ten gevolge hebben* ('to cause'), *ten goede komen* ('help, contribute to'), *ten beste geven* ('demonstrate, show'), *ter sprake komen/brengen* ('come up/bring something up'), *ten laste leggen* ('charge someone with something') given in (41).

(41) a. Dit had een ongeluk ten gevolge.
b. De opbrengst komt ten goede aan onderzoek naar kinderziekten.
c. Dat komt de prestaties niet ten goede.
d. Hij gaf een concert ten beste.
e. Het kwam ter sprake.
f. Een van hen wordt doodslag ten laste gelegd.

'Frozen' words found inside fixed phrases do not provide much morphological and categorial evidence. Their limited productivity and restricted use justify the decision to consider the fixed chunk (e.g. *in petto*) a multi-word lexeme inserted in the dictionary. Note that fixed lexemes do show morphological variety in many fixed expressions.

Agreement between subject and determiner inside fixed phrase In the fixed phrase described above, we claim that the fixed part often exhibits little syntactic flexibility even to the extent of blocking the insertion of modifiers and NP extraction. Rare cases exist where the subject agrees with a possessive determiner inside a fixed constituent. Agreement between the subject and a complement inside the VP is important to get the correct semantic interpretation of a sentence with a fixed phrase. In (42), the possessive determiner in *in haar maag* is co-referential with the NP *De Italiaanse regering.*

(42) De Italiaanse regering zit lelijk in haar maag met de vluchtelingen.
the Italian government sits nasty in her stomach with the refugees

'The Italian government has a big problem with refugees.'

From a syntactic point of view, this property entails the sharing of agreement between two NPs inside the sentence. In more formal terms, this feature raises the question of the locality constraints affecting the complements of a lexical head. It is taken for granted that a lexical head and its subject share agreement values, and since the NP subject and the VP phrasal nodes are sisters, they belong to the same local tree. The agreement exemplified in (42) requires the subject NP node and an NP object of the preposition among the verbal complements to share agreement values.

Subject idioms The idiomatic expression *schot zitten in* exhibits a 'fixed' phrase subject that can admit a restricted set of modifiers (geen) (43).

(43) In deze zaak zit geen schot
in this case is no progress
'There is no progress in this case.'

Idiomatic expressions with a fixed subject have been classified as subject idioms (Schenk, 1994). According to (Schenk, 1994), subject idioms are sentence level idioms and they do not admit tense variation nor exhibit free arguments. In our example, the idiom still has a free argument, thus it seems different from subject idioms proper. We assume that this expression is an instance of a semi-fixed phrase since the preposition *in* introduces a free argument, therefore the idiom is not totally fixed.

Data showing the relevant facts about the syntactic behavior of Dutch fixed phrases was discussed. In order to determine the syntactic variation of individual (semi-)fixed phrases, a more detailed analysis of their contextual distribution in corpora is to be pursued in future work. This has been done for English idiomatic expressions by Riehemann (2001). Riehemann (2001) claimed that 25% of the occurrences of decomposable idioms and 7% of non-decomposable ones exhibit some variation. In Riehemann's opinion, the percentages are significant enough to be ignored.

4.3 Related work

In the previous section we described the relevant properties that Dutch fixed expressions exhibit. Section 4.3.1 reviews recent proposals to formalize idiomatic expressions in the HPSG framework. Section 4.3.2 comments on approaches to extraction of multi-word lexemes and collocations.

4.3.1 Formalization of idiomatic expressions

To explain the difference between the two approaches taken to formalize idiomatic expressions, we set out with some background information about the HPSG framework. This will help to understand the limitations that these approaches face.

HPSG **Background** The basic unit of linguistic description in standard Head-Driven Phrase Structure Grammar (HPSG) is a *sign* (Pollard and Sag, 1994). The properties of *sign*s are stated via attribute-value pairs. These properties can also be thought of as constraints over the feature structures used to represent *sign*s.

In practice, when we enter a word in the lexicon, this word must belong to some *type*, e.g. noun, adjective, preposition, etc. Each of these *type*s exists in a *type hierarchy* where the relationships between *types* and *subtypes* are declared. Each *type* is defined in terms of syntactic, semantic, phonological and pragmatic properties that are appropriate. In addition, general principles of language are stated to license combinations of *signs* to form larger structures.

Standard HPSG postulates two subtypes of *sign*: *word* and *phrase*. Simplifying, most of the research done on idiomatic expressions struggles around the question of **What licenses an idiomatic expression:** a *word* or a *phrase* (*sign*)? Earlier HPSG proposals favored a **word-level approach** (Krenn and Erbach, 1994). In contrast, Riehemann (2001) adopts a constructional HPSG approach to idioms (**phrasal approach**).

A word-level approach to fixed expressions

Krenn and Erbach (1994) attempt to account for the idiosyncracies of fixed phrases in the lexicon. In standard HPSG a head can only select for SYNSEM objects that declare the syntactico-semantic information of the head's complements. Krenn and Erbach (1994) allow the verbal head within an idiomatic expression to select for whole *sign*s, rather than only SYNSEM information of its complements. This is needed to specify the phonology of the fixed lexemes that together with the verbal head constitute the idiomatic expression. Furthermore, to achieve the adequate interpretation of non-compositional idiomatic expressions, fixed lexemes selected by the verbal head contribute no semantic load to the semantic interpretation of the whole idiom.

Consider example (44). The relaxation of subcategorization requirements allows the verbal head (*hebben*) to select for the phonology of its fixed complement (*in petto*).[13] This ensures that *hebben* is idiomatic only when *in petto* is present among its complements. Assuming that *in petto* is entered as a special phrase (multi-word unit) in the lexicon, this constituent has null semantics. No semantic variable is reserved for the contribution of *in petto* in the head's CONTENT description. Instead, the idiom is assigned semantic interpretation as a whole: an *in_petto_hebben* relation.

[13]The feature structure is incomplete in (44). Only relevant parts for the discussion are included.

(44)
$$\begin{bmatrix} \text{PHON } \textit{Ze hebben weer een nieuwe machinatie in petto} \\ \text{SYNSEM} \begin{bmatrix} \text{LOC} \begin{bmatrix} \text{CAT} \begin{bmatrix}\text{SUBCAT } \langle\rangle\end{bmatrix} \\ \text{CONTENT} \begin{bmatrix} \text{RELATION} & \text{inpetto\_hebben} \\ \text{ACTOR} & \boxed{3} \\ \text{UNDERGOER} & \boxed{4}\end{bmatrix}\end{bmatrix} \\ \text{NONLOCAL } [\,] \end{bmatrix} \\ \text{DTRS} \begin{bmatrix} \text{HD-COMP-ST} \begin{bmatrix} \text{H-DTR} \mid \text{SS} \mid \text{LOC} \mid \text{CAT} \begin{bmatrix}\text{HEAD HEBBEN} \\ \text{SUBCAT} \left\langle \boxed{1}, \boxed{2}, \text{PP}\begin{bmatrix}\text{PHON} & \text{in petto}\end{bmatrix}\right\rangle \end{bmatrix} \\ \text{COMP-DTRS} \left\langle \begin{array}{l} \boxed{1} \begin{bmatrix}\text{SS} \mid \text{LOC} \begin{bmatrix} \text{CAT} \begin{bmatrix}\text{HEAD NOUN}\\ \text{CASE NOM}\end{bmatrix} \\ \text{CONTENT} \begin{bmatrix}\text{INDEX} & \boxed{3}\end{bmatrix}\end{bmatrix}\end{bmatrix}, \\ \boxed{2} \begin{bmatrix}\text{SS} \mid \text{LOC} \begin{bmatrix} \text{CAT} \begin{bmatrix}\text{HEAD NOUN}\\ \text{CASE ACC}\end{bmatrix} \\ \text{CONTENT} \begin{bmatrix}\text{INDEX} & \boxed{4}\end{bmatrix}\end{bmatrix}\end{bmatrix}, \\ \text{PP}\begin{bmatrix}\text{PHON IN PETTO}\end{bmatrix}\end{array}\right\rangle \end{bmatrix}\end{bmatrix}\end{bmatrix}$$

With these changes Krenn and Erbach (1994) may account for idiosyncratic selectional requirements of the verbal head within idioms. Since *in petto* is a multi-word fixed lexeme, it is predicted that neither modification of *petto* nor extraction out of the fixed chunk is possible. This proposal would be relatively successful for fixed expressions whose fixed lexemes allow no modification. However, as we argued in Sections 4.1 and 4.2.2 fixed expressions allow variation.

In this approach, the verbal head imposes restrictions on the internal structure of fixed complements that make up the idiomatic expression. To allow modification of fixed (idiomatic) lexemes, Krenn and Erbach (1994) introduce a HEAD feature LEXEME inside INDEX. This forces the fixed complement to have the value of LEXEME as its head but allows the insertion of specifiers and modifiers. As illustration, adjectival modification is possible in the fixed expression *onder . . . druk staan* (45). Under this approach, the verbal head *staan* specifies *druk* as head LEXEME of the fixed complement and declares no constraint on what modifiers may occur next to *druk* (46).

(45) a. De ECB komt dan onder *grote* druk te staan om het enige instrument in te zetten dat . . .
b. Maar de onderhandelingen zijn onder *zware* druk komen te staan door de recent bekend . . .

(46)
$$\begin{bmatrix} \text{PHON } \textit{staan} \\ \text{SS} \begin{bmatrix} \text{LOC} \begin{bmatrix} \text{CAT} \begin{bmatrix} \text{HEAD STAAN} \\ \text{SUBCAT} \left\langle \boxed{3}, \text{PP}\begin{bmatrix}\text{HEAD ONDER}\\ \text{SUBCAT}\left\langle \text{NP}\left[\ldots \text{INDEX}\left[\text{LEXEME DRUK}\right]\right]\right\rangle\end{bmatrix}\right\rangle \end{bmatrix} \\ \text{CONTENT} \begin{bmatrix} \text{RELATION} & \text{onder\_druk\_staan} \\ \text{ACTOR} & \boxed{3}\end{bmatrix} \end{bmatrix} \\ \text{NONLOCAL } [\,] \end{bmatrix} \end{bmatrix}$$

Allowed adjectives modify the meaning of the fixed expression. An approach that reserves no semantic variable for the 'fixed' constituent will be too restrictive to deal with data like (45). A different mechanism to license the compositional semantics of fixed expressions is needed. In this case, a semantic variable needs to be assigned to parts of the fixed expression.

In a word-level approach, all fixed words that appear within idiomatic expressions need a special lexical entry stating their idiosyncratic properties when used idiomatically. Finally, a mechanism that ensures that these words cannot occur by themselves is required to avoid expressions like *iets in petto zitten*. In addition, somehow one needs to restrict the modifiers and specifiers that may occur within fixed expressions. Also, some features would be needed inside the verbal head that licenses the fixed expression to restrict the application of lexical rules like passivization, raising, control, etc. and extraction phenomena.

A phrasal-level approach

Riehemann (1997) represents idioms as *idiomatic phrases* which are subtypes of *phrase*. To the type *phrase*, we add a new attribute WORDS whose value is the set of all idiomatic words dominated by the idiomatic phrase. The WORDS set includes *signs* of *idiomatic-words* which inherit their description from the corresponding 'ordinary literal lexical entry' via default unification(Riehemann, 1997, p.8). The effect of the WORDS feature is to provide local access to the words within an idiom. Modification and syntactic variation of parts of the idiomatic phrase is inherited from the literal counterparts of the idiomatic words unless otherwise specified. By subtyping, that is, cross-classification of *idiom phrases* and *constructions* Riehemann (1997) licenses the syntactic distribution of idiomatic phrase in contexts of topicalization, raising, relativization, passivization, etc. (Riehemann and Bender, 1999).

Riehemann's approach nicely states regularities between the syntax of literal and idiomatic uses of (idiom) phrases.[14] The idiomatic verb inherits its subcategorization information from the literal counterpart lexeme. This poses no problems for those idiomatic verbs whose SUBCAT is enlarged with fixed constituents that are not selected dependents of the literal verbal head (*hebben* in *iets in petto hebben*) because defaults can be overwritten. Consider example (47). Note that (47) does not include syntactic information about the canonical complements of the lexical

[14]This is enforced by the '$\leq$' symbol in (47).

head *hebben.* WORDS lists the required fixed lexemes of an *idiomatic phrase.*

(47)
$$\begin{bmatrix} \textit{in\_petto\_hebben\_idiom-phrase} \\ \text{WORDS} \left\{ \begin{array}{l} \begin{bmatrix} \textit{i-word} \\ \ldots\text{COMPS} \left\langle \begin{bmatrix} \ldots\text{KEY} \ \boxed{2}, \text{KEY} \ \boxed{3} \end{bmatrix} \right\rangle \\ \ldots\text{KEY } \textit{empty\_rel} \end{bmatrix} \leq \begin{bmatrix}\text{HEBBEN}\end{bmatrix}, \\ \begin{bmatrix} \textit{i-word} \\ \ldots\text{KEY } \boxed{2} \ \textit{empty\_rel} \end{bmatrix} \leq \begin{bmatrix}\text{IN}\end{bmatrix}, \\ \begin{bmatrix} \textit{i-word} \\ \ldots\text{KEY } \boxed{3} \ \textit{empty\_rel} \end{bmatrix} \leq \begin{bmatrix}\text{PETTO}\end{bmatrix} \end{array} \right\} \\ \text{CXCONT} \mid \text{LISZT} \left\langle \begin{bmatrix} \textit{in\_petto\_hebben\_rel} \\ \text{ACT } \boxed{1} \\ \text{UND } \boxed{4} \end{bmatrix} \right\rangle \end{bmatrix}$$

To restrict the application of lexical rules to the *idiomatic phrase*, location of canonical arguments of the verbal head need to be explicitly described. For instance, if the idiomatic phrase above did not allow passivization, then the object complement *een machinatie* has to be specified within SUBCAT. When passivization is possible, it applies before the idiomatic verb inherits syntactic constraints from the literal counterpart. In this particular case, the verb within the idiomatic phrase would not specify the requirement that COMPS contains the accusative NP.

Constructional rules (topicalization, wh-extraction, etc.) can also be handled by this approach. If a constituent is topicalized, a WH-construction rule will license the adequate configuration of the phrase. The local value of the topicalized constituent is specified within NONLOCAL of head verb. But its semantic index is co-referential with the value of the corresponding semantic role (within LOCAL CONTENT). We get the correct semantics of the phrase, independently of the syntactic configuration.

The semantics of the idiomatic phrase is described using Minimal Recursion Semantics (MRS). The CONTENT value of an idiomatic word is related to the CONTENT value of the non-idiomatic counterpart except for its main semantic contribution (KEY value). Duplication of lexical entries is thus avoided. The meaning of idiomatic phrases with internally regular syntax is derived compositionally. To derive the meaning of non-compositional idioms, Riehemann (1997) inserts a new feature CONTXT. The use of CONTXT requires that the relation value of the constituent words be of type *empty-relation.* This attribute provides the meaning to the whole phrase.

Riehemann (1997, p.5) justifies her departure from word-level approaches due to the difficulty to state subcategorization requirements of the lexical head without violating the subcategorization principle and difficulties in restricting the occurrences of idiomatic words e.g. *spills* exclusively to constellations where other idiomatic words e.g. *beans* are also present. The phrasal approach proposed by Riehemann (1997) handles the subcategorization problem but it is not clear how it can solve the problem with limited modification mentioned in Section 4.3.1.

4.3.2 Corpus-based methods for automatic extraction

Attempts to automate lexical acquisition of fixed expressions may benefit from prior experiences on acquisition of lexical and structural collocations and also, acquisition of verb subcategorization information. In this section, we focus on automatic acquisition of collocations.

Approaches to automatic extraction from corpora can be divided into two:

- Statistics-based approaches
- Linguistically-informed statistical approaches (also known as hybrid approaches (Krenn, 2000))

Statistics-based approach

This approach relies on the fact that co-occurrence frequency is a good indicator of collocativity. Collocations are identified on the basis of frequency of word combinations in corpora. To build the datasets, word n-grams are extracted. A numeric span is used to either extract adjacent words or tuples of words selected by applying various spans (or window sizes). Then, the lexical association between words inside the n-grams is measured by applying statistical measures such as: mutual information, dice coefficient and log-likelihood. To evaluate the performance of the different statistics, significance tests like Z-score or t-score are applied.

Three problems weaken the effectiveness of this approach. The first drawback is known as the sparse data problem: rare collocations will not be selected on the basis of mere co-occurrence frequency. Secondly, many highly frequent word combinations are selected as strong candidates although they have a non-collocational status. Finally, a lot of noise pervades in datasets (e.g. combinations of determiner and noun, or a preposition and a determiner, etc.) which consequently degrades performance of statistical tests.

To improve performance of a purely statistics-based approach, Weeber, Vos, and Baayen (2000) claim that considering only window-size as a parameter is not enough to reach an optimal recall in the extraction task. The reason is that tests such as log-likelihood or Fisher's test reach several high peaks (high recall) at different window-sizes. They suggest a ratio that compares the frequency of a target collocation in subcorpus W[15] with the frequency of that target in subcorpus C. They name this ratio the *W/C* ratio. Weeber, Vos, and Baayen (2000) report positive results with low frequency data. The task involves the extraction of verbs that occur with the particle *af* in Dutch. When the window size is 8 and the *W/C* ratio is 0.6689, their model achieves results that suggest that both log-likelihood and Fisher's test perform relatively well on low-frequency data (46% and 46.7% (recall) respectively).[16] However, one still needs to find out how their approach would work

[15]This consists of all instances of words occurring next to a seed term within a specific window size. For instance, with a window size of 10, all the words occurring within 5 words to the left and right of the seed term are included. The rest of the words falling outside that window are included in a complement subcorpus C.

[16]After removing all hapax legomena.

in a large scale collocation extraction task that does not attempt extraction of collocations with specific seed terms.

Hybrid approaches

These approaches combine the use of rich linguistic information throughout the whole extraction and identification processes with statistical models. In this section, we first describe what kind of linguistic information is used in current methodology to aid extraction of datasets. The next paragraphs describe different models that we split into *standard* and *non-standard* models with emphasis on three proposals by Lin (1998), Krenn (2000) and Blaheta and Johnson (2001). To conclude the presentation of hybrid approaches, we report on how evaluation of these models is carried out.

- BUILDING DATASETS: WHAT LINGUISTIC INFORMATION IS USEFUL?

Extraction of datasets from naturally occurring text in corpora is facilitated by adding linguistic information encoded as POS tags, lemmas, syntactic information about phrasal constructs and dependency relations. Already Church and Hanks (1990) proposed that preprocessing the corpus with a part of speech tagger improves the leverage of the datasets. Given the significant improvement in parsing efficiency and accuracy, most current work on collocation extraction from English corpora performs extraction of datasets from fully parsed data (Lin, 1998; Lin, 1999; Blaheta and Johnson, 2001; Pearce, 2001; Pearce, 2002).

Dependency relations Lin (1998) uses fully parsed data originally part of the Wall Street Journal corpus. Using a parser, Lin (1998) extracts dependency triples of the form (head POS:dependency relation:POS dependent) (e.g. have V:subj:N I). Lin (1998) aims to extract the following types of collocations: subj-verb, verb-object, adj-noun and noun-noun.

Phrasal chunks One solution to avoid skewed frequencies in datasets is to use rather shallow annotation of phrasal chunks and simply extract all instances that satisfy the pattern we are interested in. Krenn (2000) extracts PNV (Preposition Noun Verb) triples out of the POS-tagged NEGRA corpus that also includes basic syntactic structure. Krenn (2000) decides against using complete syntactic annotation because that would increase the number of structural patterns and consequently, the variation of grammatical functions in datasets.

Also, Blaheta and Johnson (2001) in an attempt to identify multi-word verb-particle combinations, they extract all the prepositions and particles that are siblings to a verb within a sentence. Later in the identification process, those combinations that have non-collocational status will be discarded because of their weak association score.

Lemmas A lemmatizer is often used to merge different morphological realizations of component lexemes within collocation candidates (Krenn, 2000; Blaheta and Johnson, 2001). After lemmatization, the number of different candidate collocation types is reduced and consequently the type's frequency is increased.

One can see that the linguistic annotation encoded in the extraction data is partly determined by the type of collocations to be extracted and the ultimate application of the list of collocations identified. Krenn (2000) investigated the very specific syntactic pattern (PNV) and showed that having access to the grammatical relation fulfilled by the PP would not help.[17] In contrast, Lin (1998) applies the list of extracted collocations as a test of word similarity and to compile a thesaurus. Capturing the most salient dependency relations between two words throughout a large corpus brings about a double benefit: grouping words into collocations and determining synonymic relations between words which fulfill a certain dependency relation with respect to a head word.[18]

Rich linguistic annotation in input data facilitates the extraction of non-contiguous word combinations with specific syntactic properties. Ultimately, the gain of adding linguistic information is a decrease in noise in datasets and hopefully, a more efficacious identification algorithm.

Extraction aids In case fully parsed data were not available, one could follow Lapata (1999) who uses a corpus query tool called *Gsearch* (Corley et al., 2001) to extract the datasets. In combination with a left-corner parser, *Gsearch* uses a context-free grammar whose rule terminals correspond to POS-tags of the tagset used for the corpus annotation.

- IDENTIFICATION MODELS

For identification of true collocations, models differ with respect to underlying assumptions on the properties of collocations and statistical scores used to measure the association between collocate words. For explanatory reasons, we divide the identification models in *standard* and *non-standard models.*

Standard models The input to the models are contingency tables that represent the frequency of the extracted candidates (n-grams) and of the component unigrams. Among the standard association measures applied to datasets are mutual information (Church and Hanks, 1990), Pearson's χ^2 test and log-likelihood ratio (Dunning, 1993; Lapata, 1999).[19]

It is well-known that mutual information is rather sensitive to low frequency data and it overestimates the association score of low-frequency candidates. To correct the poor performance of mutual information with low-frequency data a frequency cut-off may be applied. This cut-off should not be too high, otherwise a

[17]I think that the reason is the difficulty to distinguish an adjunct PP from a real verbal argument PP.

[18]Shortly, Lin (1998) measures word similarity by computing the mutual information score between a verb and a noun. Synonyms of say, a noun, are found by substituting that noun by another one which stands in the same relation (object of) to the verb and, exhibits a similar mutual information score. In this way, Lin (1998) successfully creates a thesaurus which he will use to extract non-compositional collocational phrases. Refer to Lin (1999) for further details.

[19]These three measures are used in a case study described in Section 4.4, so we will discuss them extensively there.

large percentage of candidate collocations (rare) in datasets will be removed (Weeber, Vos, and Baayen, 2000).

Pearson's χ^2 and log-likelihood tests are less sensitive to low-frequency data. Weeber, Vos, and Baayen (2000) and Pedersen (1996) claim that Fisher's score performs well even with low-frequency data. Weeber, Vos, and Baayen (2000) report a 46 % and 46.7% recall in low-frequency data and high frequency data, respectively, excluding hapax legomena only; these results show that the test is not as sensitive as mutual information to low-frequency data.

Mutual information an example model: Lin 1998 Mutual information is easy to apply to lower-order n-gram models. In addition, if more effort is put into extracting more homogeneous datasets, then the test may still be a good score to identify collocations. Lin (1998) extracts all dependency triples that exhibit a given dependency relation (e.g. subject of) and automatically corrects parser mistakes. The datasets list all found dependency triples with a frequency greater than 1 in the extraction corpus (699,219 pairs of words). Lin (1998) assumes that the dependency relation determines the part-of-speech of the head word and the modifier in a triple. The mutual information of candidate triples is computed as:[20,21]

- Mutual Information of (Head,Dependency_relation,Modifier) triple:

$$MI(H,D,M) = \log_2 \frac{P(H,D,M)}{P(H|D)P(M|D)P(D)}$$

- P(H,D,M)

$$P(H,D,M) = \frac{|w_1, rel, w_2| - c}{|*,*,*|}$$

To evaluate the model, Lin (1998) compares the list of dependency triples identified as collocations to a list of dependency triples with the same relation type extracted from the SUSANNE corpus. Lin (1998) reports a 65.3% recall and 98.6% precision in extraction of subj-verb, verb-object, adj-noun and noun-noun collocations.

Lin's 1998 approach is straightforward provided a large and fully parsed corpus is available. He used a 100 million word corpus with an estimated parsing accuracy of 95%. The simplicity of the identification model contrasts with the difficulty to evaluate its performance. Evaluation proves to be hard given the lack of a gold standard list of collocations. The biggest drawback to reproduce this approach for languages other than English is to gather such large fully parsed corpora.

Non-standard models Non-standard models combine two or more statistical scores to split apart collocational and non-collocational extracted candidates. The two models we describe below depend on data annotated with either shallow-parsed or fully parsed syntactic information.

[20]The constant c is used to adjust ranks of triples that occur only once. $|*,*,*|$ denotes the counts of all dependency triples in the extracted datasets.

[21]Where H=head word, M=head of constituent satisfying the dependency relation D and P the probability.

Krenn 2000 Section 4.2.2 emphasizes the limited variation and modification that constituent parts in fixed expressions may allow. Krenn (2000) uses the restriction on variation inside potential collocations as a measure to identify collocations.

Aiming at finding [Preposition Noun Verb] (PNV) collocations which correspond to figurative expressions and/or support verb constructions, Krenn (2000) proposes phrase entropy as a suitable method to model the variation of PP instances related to a particular tuple $(\mathrm{Preposition}, \mathrm{Noun})$.

Let us first show how entropy and phrase entropy relate to each other. Entropy is defined as the average uncertainty of a single random variable (Manning and Schütze, 1999, p.61). Put differently, the entropy measure reflects the amount of information we have about a random variable. The entropy *H(x)* formula is:

$$H(p) = H(x) = -\sum_{x \in X} p(x) \log_2 p(x)$$

Phrase Entropy calculates the entropy observed in (prepositional) phrases that may be collocates inside a larger phrase. Krenn (2000) extracts triples consisting of $(\mathrm{Prep}, \mathrm{Noun}, \mathrm{Verb})$ and measures the rigidity within the tuple $(\mathrm{Prep}, \mathrm{Noun})$ taking into account (pre/post)modification, quantification, etc. To compute the phrase entropy score of a tuple (P, N) (e.g. $(\mathrm{in}, \mathrm{spanning})$) the formula proposed is

$$\mathrm{PE}(P, N) = -\sum_{i=1}^{k} \frac{f(PP_{instance_{i_{PN_j}}})}{f(PN_j)} \log \frac{f(PP_{instance_{i_{PN_j}}})}{f(PN_j)}$$

where $f(PP_{instance_i}) = m$ and m corresponds to the number of occurrences of $PP_{instance_i}$ (e.g. $PP_{instance_1}$ is *in grote spanning*, $PP_{instance_2}$ is *in spanning*, etc.) in the extraction corpus, and $f(PN_j)$ the number of (P, N) tuples j (i.e. total number of all PP instances of $(\mathrm{in}, \mathrm{spanning})$) in the extraction corpus.

Krenn (2000) establishes the threshold $t = 0.7$ as the entropy value that divides collocational (P, N) tuples from non-collocational. Tuples with $t \leq 0.7$ are considered to be collocates.

In identifying support verb constructions, the results achieved by this model reach a recall between 51.7% and 39.8% in triples (support verb construction) where the verb has been lemmatized to its base form in datasets. In figurative expressions, recall ranges from 52.1% to 45.8% for decreasing frequency thresholds. The recall of support verb constructions gets worse for low-frequency data ($f \leq 3$). Krenn (2000) also applied the mutual information score (among others) and reported that phrase entropy achieves better results than standard association measures in medium and high frequency data.

Krenn's results are worse than Lin's 1998. In our opinion, the identification of PNV collocations (support verb constructions or figurative expressions) is harder than the identification of subject-verb, verb-object, adj-noun or even noun-noun pairs among dependency triples extracted from fully parsed data and manually corrected. Note also that these two models were applied on different languages, one of them (German) exhibits more free word order and richer inflectional morphology. Inflectional variations of the verb do have an effect on the distribution of candidate triples in datasets (Krenn, 2000). This could have repercussions on the performance of the model.

Log-linear model: Blaheta and Johnson 2001 Blaheta and Johnson (2001) attempt the extraction of particle-verb word combinations in English from 30 million words of a fully parsed section of the Wall Street Journal corpus. To build the datasets, for every verb (VB) in the corpus, the verb is tallied with every particle (PRT) or preposition (PREP) which occur as siblings to the verb within a sentence.

Blaheta and Johnson (2001) propose a log-linear model that does not assume a normal distribution of the data but a 'multinomial or Poisson distribution'; this model, they claim, should result in a better fit to count data. This type of model does not assume independence between component words, rather it models the dependences (interactions) between them. Their model (i) discounts the probability of low-frequency items, and (ii) estimates the likelihood of seeing given *n*-grams while discounting those *n*-grams that are only likely due to their component parts.

Step 1: representation of candidate *n*-tuples

N-tuples consist of a verb and its siblings in a sentence which happen to be particles or prepositions.

- Each *n*-tuple is represented as a sequence of random variables $X_1 \ldots X_n$ where X_i ranges over the $i_t h$ component of the tuple; thus, X_1 ranges over verbs and X_i, s.t. $2 \leq i \leq n$ ranges over particles and the null symbol '$\Box$'. '$\Box$' fills the value of X_i, $i > n+1$. This symbol serves as an end delimiter in tuple. Finally, a tuple is considered as a conjunction of equalities or inequalities of random variables.
  - Example: The sequence of equalities representing the English verb particle combination *look up to* would be $X_1 = \text{look} \wedge X_2 = \text{up} \wedge X_3 = \text{to} \wedge X_4 = \Box$ with $n = 4$. The equality $X_4 = \Box$ denotes that the tuple is not followed by any other particle or preposition.

Step 2: collect counts of possible subtuples of the *n*-tuple

- Each possible combination of equalities and inequalities of variables is represented with an 'n-bit integer b, $0 \leq b \leq 2^n - 1$'. For example, a few of the possible subtuples of *look up to* will be represented as:
  - * $b = 0$: $X_1 \neq \text{look} \wedge X_2 \neq \text{up} \wedge X_3 \neq \text{to} \wedge X_4 \neq \Box$
    * $b = 1$: $X_1 \neq \text{look} \wedge X_2 \neq \text{up} \wedge X_3 \neq \text{to} \wedge X_4 = \Box$
    * ...
    * $b = 14$: $X_1 = \text{look} \wedge X_2 = \text{up} \wedge X_3 = \text{to} \wedge X_4 \neq \Box$
    * $b = 15$: $X_1 = \text{look} \wedge X_2 = \text{up} \wedge X_3 = \text{to} \wedge X_4 = \Box$
- For each tuple $X_1 \ldots X_n$, C_b is the number of times the conjunction of equalities and inequalities represented by b is encountered in training data. In example above, C_{14} is the number of times that *look up to* is followed by a particle or a preposition in the corpus.
- Finally, given that each combination of equalities and inequalities is represented as 'an n-bit integer b', b is a sequence of 0 and 1 bits, such that if only one $X_i = x_i$ then the number of true equalities, $\#(b)$, is 1. In example above, $\#(14) = 3$, $\#(0) = 0$, etc. This is needed to calculate the values of λ and λ_1.

Step 3: Two different measures of association: μ and μ_1

μ and μ_1 are estimates of λ and λ_1, respectively, which are particular parameters of certain log-linear models' (Blaheta and Johnson, 2001). To prevent low-frequency data from being assigned high λ and λ_1 values, Blaheta and Johnson (2001) discount the standard error σ and σ_1 of λ and λ_1 respectively and set:

- All subtuples measure:
$$\mu = \lambda - 3.29\sigma$$
- Unigram subtuples measure:
$$\mu_1 = \lambda_1 - 3.29\sigma_1$$

They also add $\frac{1}{2}$ to each C_b to correct for small counts and to avoid problems with zero counts while computing the association measures. Thus, λ, λ_1 and the respective standard errors σ and σ_1 are given by:

$$\lambda = \sum_{b=0,..2^n-1} (-1)^{n-\#(b)} \log C_b$$

$$\lambda_1 = \log C_{2^n-1} - \sum_{\#(b)=1} \log C_b + (n-1) \log C_0$$

$$\sigma = \sqrt{\sum_{b=0}^{2^n-1} \frac{1}{C_b}}$$

$$\sigma_1 = \sqrt{\frac{1}{C_{2^n-1}} + \sum_{\#(b)=1} \frac{1}{C_b} + \frac{(n-1)^2}{C_0}}$$

To compute λ_1 the relevance of the individual unigram components of the n-tuple is subtracted.

After computing λ, λ_1 and the respective standard errors σ and σ_1, collocations are scored by calculating μ and μ_1. Subtracting the standard error has the effect of trading recall for precision (Blaheta and Johnson, 2001).

Results No recall and precision figures are given by Blaheta and Johnson (2001), thus it is hard to compare how good the model is. The authors claim that 'the collocations are almost all good for several hundred, and good collocations continue to appear well into the thousands.' To get a better idea of the model's performance, in a comparative evaluation of collocation extraction models Pearce (2002) reports that Blaheta and Johnson's method reaches an 80% recall and less than 2% precision after 40% of N-best candidates have been seen. Precision reaches 0.5% when 80% of N-best candidates have been evaluated. The task was to find two-word collocations ((Adj,Noun) or (Noun, noun) pairs) such as *hand grenade, value judgment* in English.

The emerging question now is how we should interpret these contradictory results? Perhaps the identification of verb-particle combinations is easier than $(\mathrm{Adj}/\mathrm{Noun}, \mathrm{noun})$ pairs. Particles and prepositions do not show any morphological variation, they belong to a closed class thus, it is hard to miss them in the datasets extraction.

Further remarks Pearce (2001) and Pearce (2002) propose yet a different model that rests on the assumption that a combination of words constitutes a collocation if the collocate words cannot be replaced by a synonym. This model needs synonym information extracted from WORDNET. To measure how strong a collocation we have, the algorithm computes the difference between the joint probabilities of the candidate word pair (w_1, w_2) and the probabilities of the related phrases, this being the result of substituting w_1 and w_2 for synonyms. Pearce (2002) report that his model achieves better results than Blaheta and Johnson (2001).

- EVALUATION METHODOLOGY

There is not a well-established methodology for evaluating automatically acquired collocations.

Precision and recall These two measures are typically used to account for the coverage of statistical models. Precision measures the proportion of selected items that are correct; whereas recall measures the proportion of correct items that were selected.

$$\text{Precision} = \frac{true\_positives}{true\_positives + false\_positives}$$

$$\text{Recall} = \frac{true\_positives}{true\_positives + true\_negatives}$$

In collocation extraction, precision and recall values are very rarely reported because there is no gold standard list that includes all collocations. Attempts to provide recall and precision values manually compile a list of collocations from machine-readable dictionaries. For instance, Pearce (2002) extracts a list of 17,485 two-word collocations which is later reduced to 4,152 entries that occurred at least once in the extraction corpus. This reduced list is used as gold standard.

Krenn (2000) and Evert and Krenn (2001) manually select the true collocational PREP NOUN VERB triples included in the candidate datasets that will be input to statistical models. Krenn (2000) and Evert and Krenn (2001) manually created a gold standard list which contains all true positives. They evaluate the performance of the models on the basis of (i) varying frequency thresholds applied on datasets and (ii) varying N-best candidates lists. They also plot recall and precision curves for the whole set of candidate data.

Another alternative is to manually examine the list of collocations identified by the statistical model. Typically, a random sample of frequent words (nouns, verbs) is selected. Then, all possible collocations with those words are (manually) extracted either from a learner's dictionary or an idiom's dictionary. This list will be the gold standard. Rather than comparing the gold standard list to all the extracted collocations, Lin (1998) selects only those collocations that contain the words randomly selected. Precision and recall can now be given with reference to a random sample of the extracted collocations.

Human evaluation The human judges are given guidelines that describe the properties of the type of collocations to be evaluated. Given the difficulty to define *collocation*, human evaluation is tremendously hard for those who determine the guidelines for the judgment process and for the judges themselves.

4.4 Case Study

We carried out a case study on automatic extraction of collocational prepositional phrases in Dutch. This experiment was designed to assess the performance of a hybrid model that uses standard association measures to identify expressions which we thought of as purely fixed expressions. We aim to answer two questions:

- Can data-driven models be useful to infer the syntactic behavior and appropriate linguistic description of Dutch collocational PPs?
- Are standard association measures good enough to identify collocational PPs in corpora?

The results confirm previous claims in literature. From a linguistic perspective, a uniform class of these expressions does not exist (Paardekooper, 1973). In addition, collocational PPs should not be formalized as multi-word fixed units because they may allow adverbial or adjectival modification in between constituent lexemes. From a different perspective, using linguistic information (POS-tags, sentential and phrasal boundaries) to extract datasets diminishes noise in datasets; secondly, log-likelihood and χ^2 tests are less sensitive to sparse data than mutual information. Results of log-likelihood and χ^2 are qualitatively speaking, better than raw frequency, although recall is higher if we just look at raw frequency. Evaluating the extraction and identification model proved to be a hard task given the lack of a gold standard list of collocational PPs.

4.4.1 Dutch CPPs

Collocational prepositional phrases (CPPs) exhibit the syntactic pattern [Prep NP Prep]. Examples of these phrases are *in tegenstelling tot*, *ten opzichte van*, etc., phrases that, in principle, appear to be totally fixed.

CPPs share properties with collocations and fixed phrases. Among these properties the relevant ones are: (i) head noun in NP cannot be replaced by a synonym; (ii) noun admits restricted modification and quantifiers; (iii) idiosyncratic morphology; (iv) often non-compositional meaning; (v) limited functionality as complements and, (vi) NP complement of second Prep may be an R-pronoun. Due to these characteristics, these phrases are not syntactically regular, and for many NLP applications (e.g. machine translation, generation) they would pose problems. Breidt, Segond, and Valetto (1996) suggested that such irregular phrases are best included as multi-word lexemes in the lexicon.

We used a data-driven model to determine whether collocational PPs can indeed be treated as fixed units given the evidence of their behavior found in corpora.

4.4.2 Extraction and Identification Model

Datasets extraction

The chosen methodology requires a little a priori grammatical knowledge encoded in the extraction data and used by the extraction tools. A CD-ROM version of the newspaper *de Volkskrant 97* was used to extract the datasets. The corpus had been previously tagged by a part-of-speech tagger with a rather large tagset and this tagging was performed automatically, using a Brill-tagger for Dutch (Drenth, 1997). The accuracy of the tagger is around 95%.

Extraction of instances of the chosen pattern was done by using a corpus query tool called Gsearch. Gsearch (Corley et al., 2001) allows the extraction of part of speech tagged corpus strings matching a user's query. The tool uses a small context-free grammar defined by the user, a left-corner parser and the user's query to search through the tagged corpora. Preprocessing with a tool like Gsearch filters out unwanted data to extract a more reliable dataset. Note that erroneous instances may still surface in the datasets due to tagging mistakes.

The output of Gsearch are all found instances of the pattern [prep NP prep] in the corpus. Further filtering was done to remove all instances that contained a proper name or a numeral as head of the NP. The remaining instances were sorted and assigned frequencies. Once the frequency of the candidate collocates was computed, we used Ted Pedersen's Bigram Statistic Package in order to compute the log-likelihood, mutual information and χ^2 score of each candidate in the dataset.[22] These scores measure the lexical association between the words in the candidate strings.

CPP identification models

The length of the extracted strings varies from 2 to 4 or sometimes more words. Standard statistical tests needed to be adjusted because they are usually applied to bigrams. We experiment with two different setups: a bigram model and a trigram model.

Bigrams We treated each string as a bigram $(w_1,\ w_2)$. As an example, *in tegenstelling tot* allows two possible bigram combinations ($(w_1,\ w_2)$ and $(w'_1,\ w'_2)$) where either (w_1=in, w_2=tegenstelling_tot) or (w'_1=in_tegenstelling, w'_2=tot). Each string is represented by two possible bigrams: (w_1, w_2) and $(w'_1,\ w'_2)$. The statistical tests used are: mutual information, log-likelihood and χ^2.

- Mutual Information:

$$\mathrm{MI}(w_1, w_2) = \log_2 \frac{P(w_1, w_2)}{P(w_1)P(w_2)}$$

[22]The Bigram Statistic Package provides Perl scripts to (i) extract n-grams and their frequencies from raw corpora, (ii) compute the mutual information, log-likelihood, Pearson's χ^2, Dice coefficient and Fisher's scores assigned to a given n-gram given their frequency in the corpus. This package is available at http://www.d.umn.edu/ tpederse/code.html.

- Log-likelihood:[23]
  - H_1 (independence) : $P(w_2|w_1) = P(w_2|\neg w_1)$,
  - H_2 (dependence): $P(w_2|w_1) \neq P(w_2|\neg w_1)$.

$$\log\lambda = \log\frac{L(H_1)}{L(H_2)}$$

- χ^2 for $(i, j) \in \{w_1 w_2, w_1 \neg w_2, \neg w_1 w_2, \neg w_1 \neg w_2\}$:

$$\chi^2 = \sum_{i,j} \frac{(O_{ij} - E_{ij})^2}{E_{ij}}$$

  where E_{ij} is based on the assumption of independence.

For each extracted string, we applied the statistical tests to both bigram combinations ((w_1, w_2) and (w'_1, w'_2)). The final rank assigned to a string (*in tegenstelling tot*) is the result of adding up the ranks assigned to each bigram combination ((w_1, w_2) and (w'_1, w'_2)).

Trigrams Collocational candidates are treated as 3 word units. Continuing with the same example, the collocational candidate *in tegenstelling tot* is represented as (w_1 = in, w_2 = tegenstelling,w_3 = tot). In this model, the statistical tests applied to the datasets are mutual information and Pearson's χ^2.[24]

4.4.3 Results and evaluation

Evaluation methodology

Two parameters were taken into account for the evaluation of the models:

Frequency threshold: Collocation density is typically lower among low-frequency data than among high-frequency data.We chose two different frequency cutoffs: $f \geq 10$ and $f \geq 40$. Our assumption is that word combinations that occur less than 10 times in the corpus are not likely to yield reliable data. However, we want to keep low-frequency data to make the task a bit harder for the statistical model.

Varying n-best lists: Once we applied the frequency thresholds, we are left with 2084 collocation candidates showing a frequency bigger than 10 and 317 candidates with a frequency bigger than 40. N-best lists include the N highest ranked candidates result of applying a statistical test. Three different n-best lists are chosen so that we evaluate almost all the strings with a frequency bigger than 10.

[23]This test compares the probability of two hypotheses: H_1 and H_2. The first hypothesis assumes that the constituent words in the bigram are independent. In contrast, H_2 assumes that the words are dependent. The log-likelihood score computes the log value of the result of dividing the likelihood that H_1 is correct given our background assumptions (H_2).

[24]The log-likelihood test was not computed because of the difficulty to state the conditionalization between the trigram constituents.

Evaluation data To the best of our knowledge, the only existing list of *voorzetzel uitdrukkingen* available is that proposed in Paardekooper (1973) and Haeseryn and et al. (1997). We refer to this list as the ANS list. The ANS list contains 86 voorzetsel uitdrukkingen; however, a few expressions are not used in current Dutch (e.g. *naarmate van, omwille van, ter fine van*). Thus, we manually compiled another list of CPP's.

We created another list of *cpp*s which were manually extracted from the monolingual Van Dale dictionary (Geerts and Heestermans, 1992). 200 candidate expressions proposed as true CPP's by the log-likelihood test were looked up in (Geerts and Heestermans, 1992). If under the lexical entry of the head noun inside a candidate expression (e.g. *tegenstelling*) a special phrase is listed (e.g. *in tegenstelling tot*) then, we take this 'special phrase' as a true CPP. Our validation list consisted of 88 CPP's that are listed in Van Dale's dictionary as 'special' phrases. The Van Dale's list is similar to the ANS list, only a few expressions vary.

Bigrams results

Table 4.1 gives the results of applying mutual information (mi), log-likelihood (ll) and χ^2 to the extracted collocation candidates when treated as bigrams. It also shows, in the final row, the results of comparing three n-best lists of collocational candidates retrieved on the basis of their frequency to the validation data.

			nbest		
test		n	100	300	all
mi	Freq $\geq$ 10	2084	23	39 (44.31%)	76
ll	Freq $\geq$ 10	2084	53	67 (76.1%)	77
χ^2	Freq $\geq$ 10	2084	52	69 (78.4%)	77
mi	Freq $\geq$ 40	317	47	67 (76.1%)	67
ll	Freq $\geq$ 40	317	53	65 (73.86%)	66
χ^2	Freq $\geq$ 40	317	55	65 (73.86%)	66
raw freq		248,683	50	65 (73.86%)	84

Table 4.1: Bigrams results of mutual information, log-likelihood, χ^2 and raw frequency

Discussion Mutual information, when used with a frequency threshold of 10, leads to a disproportional number of low frequency patterns among the highest scoring items, leading to poor results. It is well-known that the mutual information test performs poorly with sparse data even if large corpora are available and a frequency cut-off is used (Manning and Schütze, 1999, p.182). To correct for this effect, we used a higher frequency threshold of 40. This leads to the results in the 4th row in Table 4.1. More positive candidates are found among the smaller nbest lists in comparison to the previous frequency threshold. Considering an nbest list larger than 300 does not improve results because there are only 317 candidates that occur more than 40 times in the dataset.

The performance of the log-likelihood and Pearson's χ^2 tests is fairly similar in the bigram model. Comparing the results of these two tests for the two different thresholds becomes apparent how log-likelihood and Pearson's χ^2 are not so sensitive to low-frequency data as mutual information.

Expressions included in the validation data and not found by any statistic either have a frequency value that is lower than the frequency threshold or, the expressions are not present in the extracted datasets (missing in the corpus) (refer to the last line in Table 4.1).

Frequency counts leads to better results than MI and χ^2, and for higher *n-best* lists its accuracy comes closer to the log-likelihood and χ^2 accuracies. Similar conclusions have been reported about the extraction of Adj-Noun combinations by Evert and Krenn (2001).

Trigrams results

Results and discussion Refer to Table 4.2. For a low frequency threshold, mutual information results improve with a trigram setup of the statistical model. However, looking at larger nbest lists no difference exists between the bigram and the trigram setups.

			nbest		
test	Freq	n	100	300	all
mi	≥ 10	2,084	23	45 (51.1%)	77
χ^2	≥ 10	2,084	45	61 (69.3%)	77
mi	≥ 40	317	46	67 (76.1%)	67
χ^2	≥ 40	317	51	66 (75%)	67
raw freq		248,683	50	65 (73.86%)	84

Table 4.2: Trigrams: Results of mutual information, χ^2 and raw frequency

Comparing bigram and trigram models

From a quantitative perspective, the result of comparing the best statistic test (Pearson's χ^2) to raw frequency counts shows that χ^2 does slightly better than raw frequency in the bigram setup. In contrast, raw frequency proves to be a better test than χ^2 in the trigram setup.

From a qualitative perspective, we believe that either χ^2 or log-likelihood prove to be more appropriate for extraction of CPP's than raw frequency given the significant difference between the corresponding extracted lists. Among the CPP's extracted by the two-best statistical tests, the expressions are more fixed or less compositional than the expressions retrieved by raw frequency. Furthermore, given our purpose, applying the statistical tests has the advantage of pruning many very frequent non-collocational candidates out of the datasets. Raw frequency considers such candidates as collocational, since only their frequency is taken into account.

Human evaluation

We carried a small experiment to get insights about the difficulty of the task of finding collocations in human beings. A list with the defining properties of CPP's was given to the judges who had to assign a 1 to those expressions that satisfied all the properties. The list given to the judges included 180 candidate collocation expressions randomly selected from the output result of applying the log-likelihood test. Less than 10% of the expressions were classified as good CPP's by at least two judges.

4.4.4 Conclusions

The background study revealed that CPPs cannot be formalized as a multi-word-lexeme inserted as a fixed string in a lexicon. Variation needs to be allowed and therefore, internal structure is required.

Evaluation of the identification model proved to be a hard task because:

- gold standard list does not exist
- manually compiled Van Dale list of CPPs is limited and only covers a rather small number of these expressions
- human evaluation proved also complex due to the difficulty to state a uniform description of the properties of CPPs.

Furthermore, our results are not directly comparable to state-of-the-art extraction models. We cannot easily estimate precision and recall values given the scarce evaluation data.

To conclude, although the identification models used in the case study did not work remarkably well, they helped us to extract a reasonable list of *cpp*'s. We hope that in future experiments with different models, the list will be expanded.

4.5 Research plan

The Phd project being described here aims to contribute to the development of the Alpino wide-coverage grammar. In particular, we will concentrate on computational aspects of a syntactic property of (semi-)fixed phrases in Dutch: modification. This should also prove beneficial for a future theory of modification within fixed idiomatic expressions.

Questions we seek to answer are:

- What data-driven models are useful to automatically acquire fixed expressions?
- Applying corpus-based techniques, can we identify what modification is allowed within fixed phrases and infer their linguistic description?
- In order to handle modification within 'fixed' expressions what features and linguistic constraints need to be specified in a computational grammar?

The main objectives of the project are:

- the development of a model that performs automatic lexical acquisition of fixed phrases. The resulting model will provide a lexicon of fixed phrases and will help to infer their linguistic description.
- report on what salient features of modification within fixed expressions need to be considered in the development of the lexical representation of fixed phrases in a computational grammar.

4.5.1 Automatic acquisition of fixed phrases

A bottom-up approach will be pursued, such that we search for fixed expressions in a corpus given specific syntactic constraints. In principle, we will focus on 4 types of fixed expressions. These four types all involve (at least) a verbal lexeme and a fixed phrasal constituent: (i) [PP copular verb], (ii) [NP copular verb], (iii) [*laten* infinitive VP] and (iv) support verb constructions (verbs: *hebben*, *maken*, *houden* and *doen*).

Our goal is to find instances of fixed expressions that exhibit such patterns. Bearing this in mind, automatic extraction of fixed expressions from corpora can be decomposed into three tasks: (i) identification of fixed expressions given specific syntactic constraints, (ii) finding subcategorization frame of fixed expression and (iii) determining variation in fixed expressions.

Identification of fixed expressions given syntactic constraints

Rather homogenous (input) datasets to the identification models can be extracted by taking into account syntactic constraints of fixed expressions (Krenn, 2000; Pearce, 2002). We follow the assumption that the verbal lexeme and the 'fixed' constituent mutually select each other. Lexical selection between collocates within the fixed expression should be mirrored by high likelihood of finding them together within the same sentence.

Following Krenn (2000), we impose the following lexico-syntactic constraints: (i) word category of potential constituent lexemes in fixed expression is specified and, (ii) both constituents should occur within sentence boundaries. The extracted datasets will consist of all instances found in the extraction corpus such that the constituent lexemes exhibit a specific word category. For example, if we attempt to extract fixed expressions that involve the support verb *houden* and a fixed PP, our target expressions will include the verb *houden* and each PP instance that co-occurs with *houden* within the same sentence. If the extractor comes across the sentence *Zij houden het belang van het land in de gaten* two different candidates will be included in the datasets because they satisfy the requirement that the potential (fixed) complement is a PP: *van het land houden* and *in de gaten houden*.

Extraction from corpus Previous approaches report better performance of models that make use of fully parsed input data (Lin, 1998; Blaheta and Johnson, 2001; Pearce, 2002). Large fully-parsed corpora are not available in Dutch. The input data will be POS-tagged data. To compensate for the lack of fully parsed data, we will use Gsearch (Corley et al., 2001) to extract particular patterns out of the extraction corpus. Given that Gsearch requires a context-free grammar to search through instances of a syntactic pattern in the corpus, we will expand this context-free grammar with non-recursive phrase structure rules that reflect structure of PPs, AdjPs, NPs and all required terminals. The context-free grammar terminals match POS-tags used to annotate the corpus. Two difficulties we need to tackle are:

- extraction of non-contiguous word combinations that are instances of the pre-specified pattern.
- multiple occurrences of verbs and PPs/NPs within a sentence

In the beginning, we will focus on *subordinate structures* where it is most expected that the PP/NP (fixed chunk) is adjacent to the verbal head. This move may temporarily solve the two previous difficulties, however it may not be a good solution especially if the available corpus is not large enough.

Expanding the context-free grammar to deal with non-contiguous word combinations may have as consequence that the data in datasets becomes less reliable. An alternative experiment will try to use the Alpino parser on unseen data. Rather than using fully parsed data, we will extract sentences with syntactic annotation at the phrasal level, not at the sentence level. From that we will build the datasets.

Building datasets One of our goals is to determine the allowable modification within fixed expressions. To construct the datasets we do not want to remove information about (adjectival/adverbial) modification and quantification within the candidate fixed expressions. We will experiment with different datasets; candidate expressions may be represented by (i) all lexemes instantiating the extracted pattern; (ii) only head lexemes n-tuples ⟨Prep,Noun,Verb⟩, ⟨Verb_1=laten,Verb_2⟩ or ⟨Prep,Noun,Verb_1=support_verb⟩ and therefore intervening modification is discarded; (iii) intervening words between head lexemes replaced by their POS-tags thus, still considering the flexibility of the fixed expression and (iv) verbal lexemes within n-tuples reduced to their base form (lemmatized).

An example will illustrate the differences between datasets. Suppose we want to extract all instances of the support verb construction [PP houden]. Candidates within a dataset type (i) (according to specification given above) include all the word tokens within the PP, (e.g. *in de gaten houden, van het land houden*). Candidates within a dataset type (ii) will be *in gaten houden* or *van land houden*; type (iii) datasets list candidates such that quantifiers and modifiers within the complement PP are replaced by their part-of-speech (e.g. *in DET gaten houden, van DET land houden*); finally, datasets type (iv) replace the full form of verb lexemes (*houd,houdt,houden,gehouden,etc.*) by the base form of the verb (i.e. its lemma *houden*). The purpose of considering the 4 variants is to aleviate a potential sparse data problem while still considering variation within candidate expressions.

Following Krenn (2000) and Blaheta and Johnson (2001) we want to experiment how the different datasets influence the performance of the identification model.

Identification models We will investigate how a model that uses standard association measures such as the ones used in the case study (see Section 4.4) performs on this new task for the four different datasets described above. Second, we will apply Krenn's phrasal entropy and the log-linear models (Blaheta and Johnson, 2001) described in Section 4.3.2 to the same datasets. After the comparison and evaluation of the performance of the three models we will assess ways to improve the identification coverage.

After applying the standard association measures (trigrams model), if some tests overestimate the significance of low-frequency data we will treat low-frequency data ($2 \leq f \leq 5$) separately by applying Fisher's test (Weeber, Vos, and Baayen, 2000).

Evaluation data Automatic extraction of a collocations list from machine readable dictionaries is desired. We would like to explore the feasibility of compiling a list of specific collocations (e.g. with support verbs) from the electronic version of *Van Dale Groot Woordenboek der Nederlandse Taal* Geerts and Heestermans (1992) for evaluation purposes.

For earlier experiments a random sample of collocations will be selected out of varying *N*-best candidates list and manually evaluated by native speakers.

Finding variation and subcategorization frame of fixed expression

Provided we have a list of fixed expressions, we will semi-automatically explore all instances of each fixed expression in corpora. By doing this, we aim at collecting evidence from corpora to determine: (i) which lexemes allow modification in fixed phrases, (ii) what type of variation is allowed (nouns, adjectives, suffixes, determiners, adverbial intensifiers) and finally, (iii) under which conditions can variation be overtly realized?. Secondly, we will determine the required subcategorization frame of the verb lexeme. This should provide syntactic valency information of the verbal head within the fixed expression that needs to be annotated in the lexicon.

4.5.2 Lexical representation of flexible 'fixed' expressions

The second concrete objective of the project is to report on salient features of fixed expressions that ought to be considered during the development of the lexical representation of modification within fixed expressions in a computational grammar.

The task involves first, understanding the grammar of modification phenomena within fixed expressions and reporting how current formalizations of fixed expressions under the HPSG framework handle modification and second, informing the grammar developers about which features and linguistic constraints are needed to handle modification within fixed expressions in a computational grammar.

Syntactico-semantic properties Our primary focus is the linguistic motivation of an adequate lexical representation of fixed phrases, together with the identification of constraints that determine the type as well as the location of modification.

Section 4.2.2 described the main syntactico-semantic properties of fixed expressions.

Given the unpredictability of syntactic and semantic behavior of fixed expressions, the properties more difficult to formalize are:

- lexical selection of fixed lexemes that may allow restricted modification
- prevention of idiomatic words from occurring outside the fixed expression
- exceptional: agreement between determiners within fixed constituent and the subject; subject idioms

Three questions need to be answered:

- Where should these properties be encoded in a constraint-based lexicalist grammar?
  - lexicon component
  - grammar component
- How to set restrictions on fixed expressions that allow modification?
- What changes in the current Alpino grammar are required to handle the data?

To answer these questions we will investigate recent proposals to analyse idiomatic expressions in HPSG that combine two types of approaches: word-level and phrasal-level approaches (Sailer, 2000; Riehemann, 2001). In the future we will report the potential of Riehemann (2001) proposal in theoretical linguistics to account for the Dutch data. We will pursue a comparative study between Riehemann's and Sailer's proposals exploring which approach handles modification within fixed expressions in a more efficient way. In the end we aim at a characterization of modification within the 4 types of Dutch fixed phrases, by proposing the required lexical representation and grammatical description in the lexicalist Alpino grammar.

Part III

Annotation Efforts

Chapter 5

The Alpino Dependency Treebank

5.1 Introduction

In this section we present the Alpino Dependency Treebank and the tools that we have developed to facilitate the annotation process. Annotation typically starts with parsing a sentence with the Alpino parser. The number of parses that is generated is reduced through interactive lexical analysis and constituent marking. A tool for on line addition of lexical information facilitates the parsing of sentences with unknown words. The selection of the best parse is done efficiently with the parse selection tool. At this moment, the Alpino Dependency Treebank consists of about 6,000 sentences of newspaper text that are annotated with dependency trees. The corpus can be used for linguistic exploration as well as for training and evaluation purposes.

A syntactically annotated corpus is needed to train disambiguation models for computational grammars, as well as to evaluate the performance of such models, and the coverage of computational grammars. For this purpose we have started to develop the Alpino Dependency Treebank.

The treebank consists of sentences from the newspaper (`cdbl`) part of the Eindhoven corpus (Uit den Boogaard 1975). The sentences are each assigned a dependency structure, which is a relatively theory independent annotation format. The format is taken from the corpus of spoken Dutch (CGN)[1] (Oostdijk 2000), which in turn based its format on the Tiger Treebank (Skut 1997). In section 5.2 we go into the characteristics of dependency structures and motivate our choice for this annotation format.

Section 5.3 is the central part of this paper. Here we explain the annotation method as we use it, the tools that we have developed, the advantages and the shortcomings of the system. It starts with a description of the parsing process that is at the beginning of the annotation process. Although it is a good idea to start annotation with parsing (building dependency trees manually is very time consuming and error prone), it has one main disadvantage: ambiguity. For a sentence of average length typically a set of hundreds or even thousands of parses is generated. Selection of the best parse from this large set of possible parses is time intensive.

The tools that we present in this paper aim at facilitating the annotation process

[1]http://lands.let.kun.nl/cgn

and making it less time consuming. We present two tools that reduce the number of parses generated by the parser and a third tool that facilitates the addition of lexical information during the annotation process. Finally a parse selection tool is developed to facilitate the selection of the best parse from the reduced set of parses.

The Alpino Dependency Treebank is a searchable treebank in an XML format. In section 5.4 we present examples illustrating how the standard XML query language XPath can be used to search the treebank for linguistically relevant information. In section 5.5 we explain how the corpus can be used to evaluate the Alpino parser and to train the probabilistic disambiguation component of the grammar. We end with conclusions and some pointers to future work in 5.7.

5.2 Dependency Trees

The meaning of a word or a sentence is represented in standard HPSG by semantic representations that are added to lexical entries and phrases. Semantic principles define the construction of a semantic structure from these representations. In Alpino we have added the DT features with which we build a dependency tree instead.

Dependency structures represent the grammatical relations that hold in and between constituents. On the one hand they are more abstract than syntactic trees (word order for example is not expressed) and on the other hand they are more explicit about the dependency relations. Indices denote that constituents may have multiple (possibly different) dependency relations with different words. Fig. 5.1 shows the dependency tree for the sentence *Kim wil weten of Anne komt*. The dependency relations are the top labels in the boxes. In addition, the syntactic category, lexical entry and string position are added to each leaf. The index **1** indicates that *Kim* is the subject of both *wil* (wants) and *weten* (to know).

The main advantage of this format is that it is relatively theory independent, which is important in a grammar engineering context. A second advantage is that the format is similar to the format CGN uses (and that they in turn based on the Tiger Treebank), which allowed us to base our annotation guidelines on theirs (Moortgat, Schuurman and van der Wouden 2001). The third and last argument for using dependency structures is that it is relatively straightforward to perform evaluation of the parser on dependency structures: one can compare the automatically generated dependency structure with the one in the treebank and calculate statistical measures such as F-score based on the number of dependency relations that are identical in both trees (Carroll, Briscoe, and Sanfilippo, 1998).

5.3 The annotation process

The annotation process is roughly divided into two parts: we first parse a sentence with the Alpino parser and then select the best parse from the set of generated parses. Several tools that we have developed and implemented in Hdrug, a graphical environment for natural language processing (van Noord and Bouma 1997), facilitate the two parts of the annotation process. In section 5.3.1 we present an

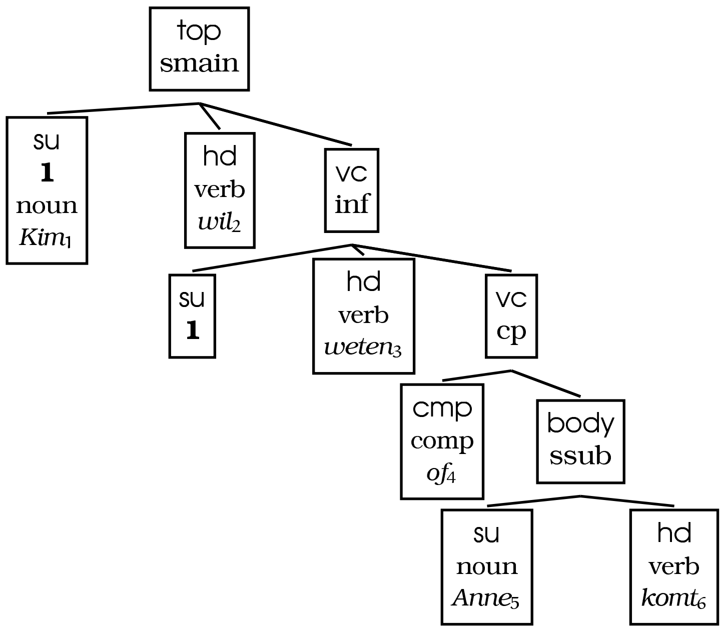


Figure 5.1: Dependency tree voor de zin *Kim wil weten of Anne komt*

interactive lexical analyzer, a constituent marker and a tool for temporary addition of lexical information. The parse selection tool is described in in section 5.3.5.

5.3.1 Parsing

The annotation process typically starts with parsing a sentence from the corpus with the Alpino parser. This is a good method, since building up dependency trees manually is extremely time consuming and error prone. Usually the parser produces a correct or almost correct parse. If the parser cannot build a structure for a complete sentence, it tries to generate as large a structure as possible (e.g. a noun phrase or a complementizer phrase). The main disadvantage of parsing is that the parser produces a large set of possible parses (see fig.5.2). This is a well known problem in grammar development: the more linguistic phenomena a grammar covers, the greater the ambiguity per sentence. Because selection of the best parse from such a large set of possible parses is time consuming, we have tried to reduce the set of generated parses. The interactive lexical analyzer and the constituent marker restrict the parsing process which results in reduced sets of parses. A tool for on line addition of lexical information makes parsing of sentences with unknown words more accurate and efficient.

5.3.2 Interactive lexical analysis

The interactive lexical analyzer is a tool that facilitates the selection of lexical entries for the words in a sentence. It presents all possible lexical entries for all words in the sentence to the annotator. He or she may mark them as correct, good or bad.

- Correct Parse must include it

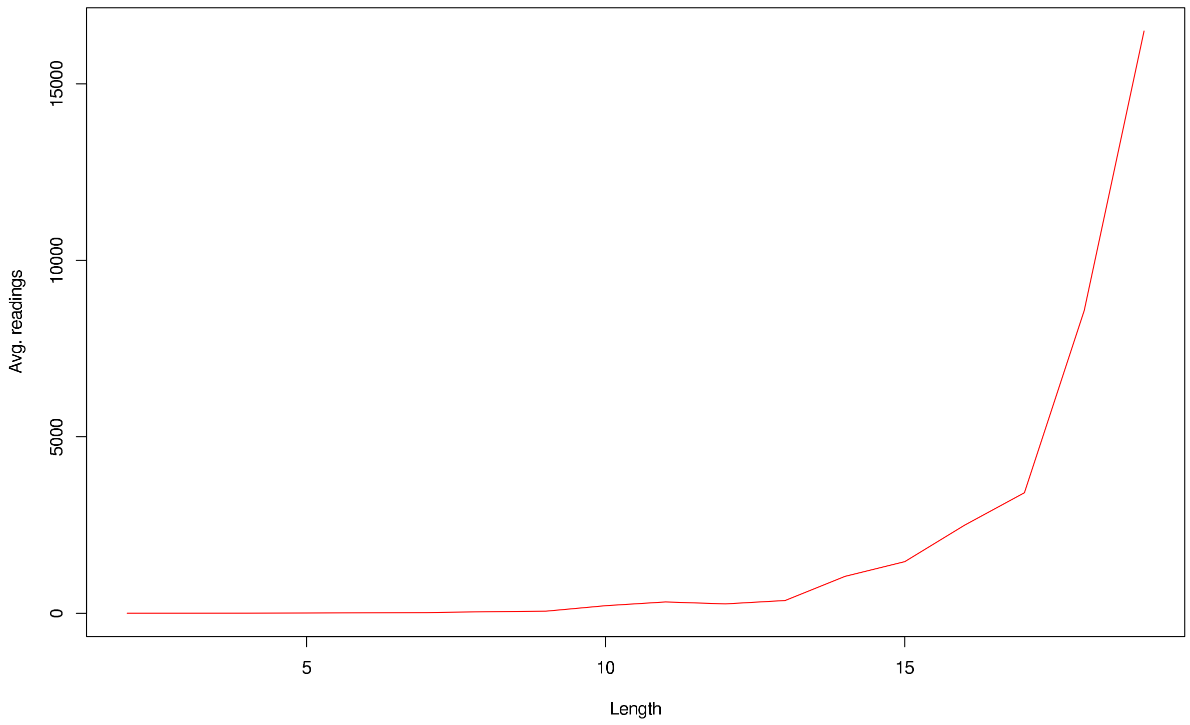


Figure 5.2: Number of parses generated per sentence by the Alpino parser

- Good Parse may include it
- Bad Parse may not include it

One *correct* mark for a particular lexical entry automatically produces *bad* marks for all other entries for the same word. The parser uses the reduced set of entries to generate a significantly smaller set of parses in less processing time.

5.3.3 Constituent Marking

The annotator can mark a piece of the input string as a constituent by putting square brackets around the words. The type of constituent can be specified after the opening bracket. The parser will only produce parses that have a constituent of the specified type at the string position defined in the input string. Even if the parse cannot generate the correct parse, it will produce parses that are likely to be close to the best possible parse, because they do oblige to the restrictions posed on the parses by the constituent marker.

Constituent marking has some limitations. First, the specified constituent borders are defined on the syntactic tree, not the dependency tree (dependency structures are an extra layer of annotation that is added to the syntactic structure). Using the tool therefore requires knowledge of the Alpino grammar and the syntactic trees that it generates.

Second, specification of the constituent type is necessary in most cases, especially for disambiguating prepositional phrase attachments. As shown in fig. 5.3, a noun phrase and a prepositional phrase can form a constituent on different levels.

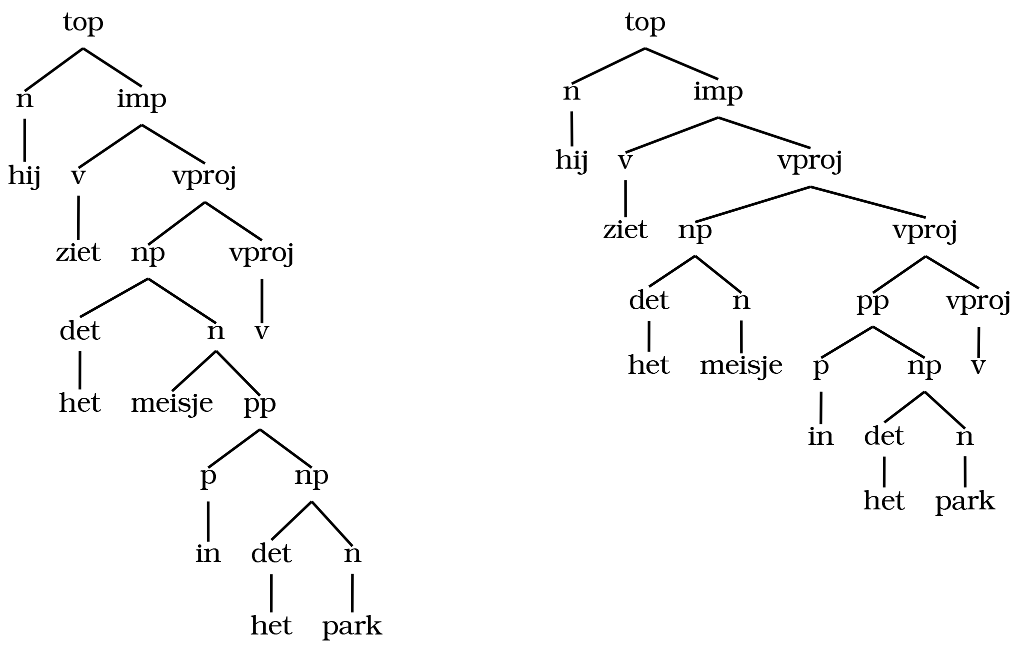


Figure 5.3: PP attachment ambiguity in Alpino

The two phrases can form either a noun phrase or a verbal projection with an empty verb (which is used in the grammar to account for verb second). The first structure corresponds to a dependency structure with a noun phrase internal prepositional modifier, the second corresponds to a dependency tree in which the prepositional phrase is a modifier on the sentence level. Marking the string `het meisje in het park` as a constituent without further specification does not disambiguate between the two readings: in both readings the string is a constituent. One has to specify that the string should be a noun phrase, not a verbal projection. This specification of the constituent type requires even more knowledge of the grammar. If one specifies a constituent type that cannot be formed at the denoted string position, the parser treats the specification as an illegal character, skips it and generates partial parses only.

5.3.4 Addition of lexical information

Alpino is set up as a broad coverage parser. The goal is to build an analyzer of unrestricted text. Therefore a large lexicon has been created and extensive unknown word heuristics have been added to the grammar. Still, it is inevitable that the parser will come across unknown words that it cannot handle yet. Verbs are used with extra or missing arguments, Dutch sentences are mingled with foreign words, spelling mistakes make common words unrecognizable. In most cases, the parser will either skip such a word or assign an inappropriate category to it. The only way to make the system correctly use the word, is to add a lexical entry for it in the lexicon.

Adding new words to the lexicon costs time: one has to write the entry, save the new lexicon and reload it. It would be far more efficient to add all new words one

top:hd	=	v *wil*
top:su	=	n *Kim*
top:vc:hd	=	v *weet*
top:vc:su	=	n *Kim*
top:vc:vc:cmp	=	comp *of*
top:vc:vc:body:hd	=	v *kom*
top:vc:vc:body:su	=	n *Anne*

Figure 5.4: Set of dependency paths for the sentence *Kim wil weten of Anne komt*

comes across during an annotation session at once, avoiding spurious reloadings. Furthermore, not all unknown words the parser finds should be added to the lexicon. One would want to use misspelled words and verbs with an incorrect number of arguments only once to build a parse with.

Alpino has temporary, on line addition of lexical information built in for this purpose. Unknown words can temporarily be added to the lexicon with the command `add_tag` or `add_lex`. Like the words in the lexicon, this new entry should be assigned a feature structure. `add_tag` allows the user to specify the lexical type as the second argument. However types may change and especially for verbs it is sometimes hard to decide which of the subcategorization frames should be used. For that reason the command `add_lex` allows us to assign to unknown words the feature structure of a similar word, that could have been used on that position. The command `add_lex stoel tafel` for instance assigns all feature structures associated with *tafel* to the word *stoel*. The command `add_lex zoen slaap` assigns *zoen* all feature structures of *slaap*, including imperative and 1st person singular present for all sub-categorization frames of *slapen*. The lexical information is automatically deleted when the annotation session is finished.

5.3.5 Selection

Although the number of parses that is generated is strongly reduced through the use of different tools, the parser usually still produces a set of parses. Selection of the best parse (i.e. the parse that needs the least editing) from this set of parses is facilitated by the parse selection tool. This design of this tool is based on the SRI Treebanker (Carter 1997).

The parse selection takes as input a set of dependency paths for each parse. A dependency path specifies the grammatical relation of a word in a constituent (e.g. head (hd) or determiner (det)) and the way the constituent is embedded in the sentence. The representation of a parse as a set of dependency paths is a notational variant of the dependency tree. The set of dependency triples that corresponds to the dependency tree in fig. 5.1 is in fig. 5.4.

From these sets of dependency paths the selection tool computes a (usually much smaller) set of maximal discriminants. This set of maximal discriminants consists of the triples with the shortest dependency paths that encode a certain difference between parses. In example 5.5 the triples *s:su:det = det het* and *s:su = np het meisje* always co-occur, but the latter has a shorter dependency path and

s:hd	=	v *zag*	s:hd	=	v *zag*
*s:su	=	np *jan*	*s:su	=	np *het meisje*
*s:obj1	=	np *het meisje*	s:su:det	=	det *het*
s:obj1:det	=	det *het*	s:su:hd	=	n *meisje*
s:obj1:hd	=	n *meisje*	*s:obj1	=	np *jan*

Figure 5.5: Two readings of the sentence *Jan zag het meisje* represented as sets of dependency paths. An '*' indicates a maximal discriminant

is therefore a maximal discriminant. Other types of discriminants are lexical and constituent discriminants. Lexical discriminants represent ambiguities that result form lexical analysis, e.g. a word with an uppercase first letter can be interpreted as either a proper name or the same word without the upper case first letter. Constituent discriminants define groups of words as constituents without specifying the type of the constituent.

The maximal discriminants are presented to the annotator, who can mark them as either good (parse must include it) or bad (parse may not include it). The parse selection tool then automatically further narrows down the possibilities using four simple rules of inference. This allows users to focus on discriminants about which they have clear intuitions. Their decisions about these discriminants combined with the rules of inference can then be used to make decisions about the less obvious discriminants.

1. If a discriminant is bad, any parse which includes it is bad
2. If a discriminant is good, any parse which doesn't include it is bad
3. If a discriminant is only included in bad parses, it must be bad
4. If a discriminant is included in all the undecided parses, it must be good

The discriminants are presented to the annotator in a specific order to make the selection process more efficient. The highest ranked discriminants are always the lexical discriminants. Decisions on lexical discriminants are very easy to make and greatly reduce the set of possibilities.

After this the discriminants are ranked according to their power: the sum of the number of parses that will be excluded after the discriminant has been marked *bad* and the number of parses that will be excluded after it has been marked *good*. This way the ambiguities with the greatest impact on the number of parses are resolved first.

The parse that is selected is stored in the treebank. If the best parse is not fully correct yet, it can be edited in the Thistle (Calder 2000) tree editor and then stored again. A second annotator checks the structure, edits it again if necessary and stores it afterwards.

```
<node rel="top" cat="smain" start="0" end="6" hd="2">
  <node rel="su" pos="noun" cat="np" index="1"
        start="0" end="1" hd="1" root="Kim" word="Kim"/>
  <node rel="hd" pos="verb"
        start="1" end="2" hd="2" root="wil" word="wil"/>
  <node rel="vc" cat="inf" start="2" end="6" hd="3">
    ....
  </node>
</node>
```

Figure 5.6: XML encoding of dependency trees.

5.4 Querying the treebank

The results of the annotation process are stored in XML. XML is widely in use for storing and distributing language resources, and a range of standards and software tools are available which support creation, modification, and search of XML documents. Both the Alpino parser and the Thistle editor output dependency trees encoded in XML.

As the treebank grows in size, it becomes increasingly interesting to explore it interactively. Queries to the treebank may be motivated by linguistic interest (i.e. which verbs take inherently reflexive objects?) but can also be a tool for quality control (i.e. find all PPs where the head is not a preposition).

The XPath standard[2] implements a powerful query language for XML documents, which can be used to formulate queries over the treebank. XPath supports conjunction, disjunction, negation, and comparison of numeric values, and seems to have sufficient expressive power to support a range of linguistically relevant queries. Various tools support XPath and can be used to implement a query-tool. Currently, we are using a C-based tool implemented on top of the LibXML library.[3]

The XML encoding of dependency trees used by Thistle (and, for compatibility, also by the parser) is not very compact, and contains various layers of structure that are not linguistically relevant. Searching such documents for linguistically interesting patterns is difficult, as queries tend to get verbose and require intimate knowledge of the XML structure, which is mostly linguistically irrelevant. We therefore transform the original XML documents into a different XML format, which is much more compact (the average file size reduces with 90%) and which provides maximal support for linguistic queries.

As XML documents are basically trees, consisting of elements which contain other elements, dependency trees can simply be represented as XML documents, where every node in the tree is represented by an element `node`. Properties are represented by attributes. Terminal nodes (leaves) are nodes which contain no daughter elements. The XML representation of (the top of) the dependency tree given in figure 5.1 is given in figure 5.6.

The transformation of dependency trees into the format given in figure 5.6 is not only used to eliminate linguistically irrelevant structure, but also to make explicit

[2]www.w3.org/TR/xpath
[3]www.xmlsoft.org/

information which was only implicitly stored in the original XML encoding. The indices on root forms that were used to indicate their string position are removed and the corresponding information is added in the attributes `start` and `end`. Apart from the root form, the inflected form of the word as it appears in the annotated sentence is also added. Words are annotated with part of speech (`pos`) information, whereas phrases are annotated with category (`cat`) information. A drawback of this distinction is that it becomes impossible to find all NPs with a single (non-disjunctive) query, as phrasal NPs are `cat="np"` and lexical NPs are `pos="noun"`. To overcome this problem, category information is added to non-projecting (i.e. non-head) leaves in the tree as well. Finally, the attribute `hd` encodes the string position of the lexical head of every phrase. The latter information is useful for queries involving discontinuous constituents. In those cases, the start and end positions may not be very informative, and it can be more interesting to be able to locate the position of the lexical head.

We now present a number of examples which illustrate how XPath can be used to formulate various types of linguistic queries. Examples involving the use of the `hd` attribute can be found in Bouma and Kloosterman (2002).

Objects of prepositions are usually of category NP. However, other categories are not completely excluded. The query in (1) finds the objects within PPs.

(1) `//node[@cat="pp"]/node[@rel="obj1"]`

The double slash means we are looking for a matching element anywhere in the document (i.e. it is an ancestor of the top element of the document), whereas the single slash means that the element following it must be an immediate daughter of the element preceding it. The `@`-sign selects attributes. Thus, we are looking for nodes with dependency relation `obj1`, immediately dominated by a node with category `pp`. In the current state of the dependency treebank, 98% (5,892 of 6,062) of the matching nodes are regular NPs. The remainder is formed by relative clauses (*voor wie het werk goed kende, for who knew the work well*), PPs (*tot aan de waterkant, till on the waterfront*), adverbial pronouns (see below), and phrasal complements (*zonder dat het een cent kost, without that it a penny costs*).

The CGN annotation guidelines distinguish between three possible dependency relations for PPs: complement, modifier, or 'locative or directional complement' (a more or less obligatory dependent containing a semantically meaningful preposition which is not fixed). Assigning the correct dependency relation is difficult, both for the computational parser and for human annotators. The following query finds the head of PPs introducing locative dependents:

(2) `//node[@rel="hd" and ../@cat="pp" and ../@rel="ld"]`

Here, the double dots allow us to refer to attributes of the dominating XML element. Thus, we are looking for a node with dependency relation *hd*, which is *dominated* by a PP with a *ld* dependency relation. Here, we exploit the fact that the mother node in the dependency tree corresponds with the immediately dominating element in the XML encoding as well.

Comparing the list of matching prepositions with a general frequency list reveals that about 6% of the PPs are locative dependents. The preposition *naar* (*to, towards*) typically introduces locative dependents (50% (74 out of 151) of its usage), whereas the most frequent preposition (i.e. *van, of*) does introduce a locative in only 1% (15 out of 1496) of the cases.

In PPs containing an impersonal pronoun like *er* (*there*), the pronoun always precedes the preposition. The two are usually written as a single word (*eraan, thereon*). A further peculiarity is that pronoun and preposition need not be adjacent (*In Delft wordt* **er** *nog* **over** *vergaderd* (*In Delft, one still talks about it*)). The following query finds such discontinuous phrases:

(3)
```
//node[@cat="pp" and
./node[@rel="obj1"]/@end < ./node[@rel="hd"]/@start ]
```

Here, the '<'-operator compares the value of the end position of the object of the PP with the start position of the head of the PP. If the first is strictly smaller than the second, the PP is discontinuous. The corpus contains 133 discontinuous PPs containing an impersonal pronoun vs. almost 322 continuous pronoun-preposition combinations, realized as a single word, and 17 cases where these are realized as two words. This shows that in almost 25% of the cases, the preposition + impersonal pronoun construction is discontinuous.

5.5 Evaluation Metrics

One of the applications of the Alpino treebank is the evaluation of the performance of the parser. Evaluation is done by comparing the dependency structure that the parser generates for a given corpus sentence, to the dependency structure that is stored in the treebank. For the purpose of comparison, we do not use the representation of dependency structures as trees, but the alternative notation as sets of dependency paths that we already saw in the previous section. Comparing these sets, we can count the number of relations that are identical in the parse that the system generated and the stored structure. From these counts precision, recall and F-score can be calculated. In our experience, the following metric, *concept accuracy* (CA), is a somewhat more reliable indicator of the quality of the system.[4] This metric is defined for a given sentence s:

$$\text{CA}(s) = 1 - \frac{D_f(s)}{max(D_g(s), D_p(s))}$$

Here, $D_p(s)$ is the set of dependency relations of the parse for sentence s (generated by the parser). $D_g(s)$ is the set of dependency relations of the gold parse that is stored in the treebank for s. $D_f(s)$ is the number of incorrect or missing relations in $D_p(s)$.

For a corpus of sentences S, we often report the *per sentence* mean CA score. In addition, it is useful to consider the overall CA score:

$$\text{CA}(S) = 1 - \frac{\Sigma_{s \in S} D_f(s)}{max(\Sigma_{s \in S} D_g(s), \Sigma_{s \in S} D_p(s))}$$

[4]This metric is a variant of the metric introduced in (Boros et al., 1996); we adapted the metric to ensure that the score is between 0 and 100.

In chapter 11, a number of disambiguation models is described. Such disambiguation models aim to pick out the best parse from a set of possible parses. It is therefore natural to think of the upper and lower bounds of parse selection, on the basis of the potential scores of each of the parses in the set from which a choice has to be made. The lower bound *baseline* CA is defined as the CA of a random parse from the set of parses. The upper bound *best* CA is defined as the maximum CA of any of the parses. *best* CA is 100 just in case the correct parse is among the set of parses generated by the grammar. The *best* CA score reflects the accuracy of the grammar. Based on these bounds, we define an adjusted concept accuracy:

$$\mathrm{CA}_\kappa = 100 \times \frac{\mathrm{CA} - \textit{baseline}\ \mathrm{CA}}{\textit{best}\ \mathrm{CA} - \textit{baseline}\ \mathrm{CA}}$$

The adjusted concept accuracy CA_κ allows the models to be more broadly compared to others by incorporating not only the concept accuracy of the model but also the lower and upper bound accuracy.

5.6 Annotator Agreement

This section describes a small experiment that we performed in order to investigate the agreement among annotators that can be expected for treebanks of this sort. The setup of the experiment was as follows. Two annotators were asked to annotate the same set of sentences, independently from each other. These two annotators were trained to annotate according to the CGN guidelines. The set consisted of the last one hundred sentences of nineteen words of the `cdbl` corpus. Perhaps it would have been better to perform the experiment on sentences of varying length, but all shorter sentences of this corpus were already annotated. Note that the length of nineteen words is a rather typical sentence length for this corpus (mean sentence length is about twenty words).

For 42 sentences, the two annotators produced the same set of dependency relations. In table 5.1 we list the annotator agreement, expressed in terms of the *concept accuracy* metric defined in the previous section. In a few cases, the differences in annotation were due to simple mistakes. In most cases, the differences were due to a true difference in linguistic analysis. In the table we quantify what types of disagreement occurred often. The most frequent cause of disagreement were differences in attachment of modifiers. Another frequent cause of disagreement is related to the choice of the dependency relation: typical disagreements involve the labels MOD, LD, PC for modifiers, directional complements, and prepositional complements, respectively. In some cases, the disagreement is about whether or not a fixed phrase should be treated as a multi-word unit, or not. Disagreements of this sort are counted rather heavily in the concept accuracy measure, because each relation that involve the multi-word unit will be treated as incorrect. The final somewhat larger class of mistakes were related to the identification of discourse units (for instance for the analysis of certain spoken language constructs that sometimes occur in written texts too).

agreement	93.1 %
mistakes	1.5 %
agreement after correction	94.6 %
attachment of modifiers	1.5 %
different dependency labels	1.3 %
multi word units	1.1 %
discourse units	0.7 %
misc	0.7 %

Table 5.1: Results of Annotator Agreement Experiment

5.7 Conclusions

A treebank is very important for both evaluation and training of a grammar. For the Alpino parser, no suitable treebank existed. For that reason we have started to develop the Alpino Dependency Treebank by annotating a part of the Eindhoven corpus with dependency structures. As the treebank grows in size, it becomes more and more attractive to use it for linguistic exploration as well, and we have developed an XML format which supports a range of linguistic queries.

To facilitate the time consuming annotation process, we have developed several tools: interactive lexical analysis and constituent marking reduce the set of parses that is generated by the Alpino parser, the tool for addition of lexical information makes parsing of unknown words more efficient and the parse selection tool facilitates the selection of the best parse from a set of parses. In the future, constituent marking could be made more user friendly. We could also look into ways of further reducing the set of maximal discriminants that is generated by the parse selection tool.

The treebank currently contains over 6,000 sentences, and is available on http://www.let.rug.nl/ vannoord/trees. Much effort will be put in extending the treebank to at least the complete cdbl newspaper part of the Eindhoven corpus, which contains more than 7,100 sentences.

Part IV

Finite State Language Processing

Chapter 6

Compact and Efficient Finite Automata for NLP

6.1 Introduction

There are two main reasons for which finite state automata are used in natural language processing: small size and great speed compared to other, alternative representations. In the framework of the present project, memory-efficient representations with finite state automata were investigated in the following aspects:

- impact of combinations of various compression techniques on the size of the resulting automaton was examined (Section 6.2);

- compressed language models using finite state perfect hashing techniques and finite state compression methods (Section 6.2) were developed (Section 6.3);

- an incremental minimization algorithm (Watson, 2001) was improved so that it can run in polynomial instead of exponential time (Section 6.4).

Although a finite state automaton is small and fast, building one may require considerable resources. Acyclic finite automata are frequently used as dictionaries. A comparison of existing algorithms for constructing acyclic finite state automata from sets of strings was conducted (Section 6.5). This research lead to a new algorithm that proves to be the fastest for word lists when sufficient memory is available.

In some applications cyclic automata are required. Although it is usually possible to separate the cyclic and acyclic parts, one automaton for all cases is faster than two separate ones. In a recent paper (Carrasco and Forcada, 2002) proposed a generalization of one of algorithms for constructing acyclic finite state automata to the case when strings are added to a cyclic automaton. A related algorithm constructing automata from sorted lists of strings can also be generalized in a similar way (Section 6.6).

6.2 Compression of Automata

Finite-state automata are used in various applications. One of the reasons for this is that they provide very compact representations of sets of strings. However, the size of an automaton measured in bytes can vary considerably depending on the storage method in use. Most of them are described in (Kowaltowski, Lucchesi, and Stolfi, 1993), a primary reference for all interested in automata compression. However, (Kowaltowski, Lucchesi, and Stolfi, 1993) does not provide sufficient data on the influence of particular methods on the size of the resulting automaton. We investigate that in this paper. We used only deterministic, acyclic automata in our experiments. However, the methods we used do not depend on that feature. The automata we used were minimal (otherwise the first step in compression should be the minimization).

Our starting point is as follows. An automaton is stored as a sequence of transitions (fig. 6.1). The states are represented only implicitly. A transition has a label, a pointer to the target state, the number of transitions leaving the target state (*transition counter*), and a final marker. We use the transition counter to determine the boundaries of states, instead of finding them by subtracting addresses of states in a large vector of addresses of states as in (Kiraz, 1999), because we can get rid of the vector.

	L	F	#	→
1	s		2	2
2	t		1	4
3	a		1	5
4	a		1	5
5	y	•	0	0

Figure 6.1: Starting point storage method. *L* is a label, F marks final transitions, # is the number of outgoing transitions in the target state, and → is a pointer to the target state. The automaton recognizes words *say* and *stay*.

We store the final marker as the most significant bit of the transition counter. We use non-standard automata, *automata with final transitions* (see (Daciuk, 1998)), because they have less states and less transitions than the traditional ones. But since the storage methods are the same, the results are also valid for traditional automata.

6.2.1 Compression Techniques

Compression techniques fall into three main categories:

- coding of input data,
- making some parts of an automaton share the same space,
- reducing the size of some elements of an automaton.

Some techniques may contribute to the compression in two ways, e.g. changing the order of transitions in a state can both make sharing some transitions possible and reduce the size of some pointers. The first category depends on the kind of data that is stored in the automaton. The techniques we use apply to natural language dictionaries.

We define a deterministic finite-state automaton as $A = (\Sigma, Q, i, F, E)$, where Q is a finite set of states, $i \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, and $E \subseteq Q \times \Sigma \times Q$ is a set of transitions. We also define a function *bytes* that returns the number of bytes needed to store its argument: $bytes(x) = \lceil \log_{256} x \rceil$. Total savings (in percents) achieved by using a particular method M on the starting point automaton are $\eta^M(A) = 100\% \cdot \tau^M(A) \cdot \pi^M(A)/sizeof(A)$, where $\tau^M(A)$ is the number of transitions affected by the compression method, $\pi^M(A)$ is the saving in bytes per affected transition, and *sizeof(A)* is the size of the automaton in bytes. The size of an automaton in the starting point representation is $|E|(2 \cdot bytes|\Sigma| + bytes(|E|))$. In all those calculations, we assume that additional one-bit flags they require fit into the space taken by a pointer without the need to enlarge it.

Coding of Input Data

This subsubsection applies to natural language morphological dictionaries. Entries in such dictionaries usually contain 3 pieces of information: the inflected form, the base form, and the categories associated with the inflected form. It is common (e.g. in INTEX (Silberztein, 1999), (Silberztein, 1997), and in systems developed at the University of Campinas (Kowaltowski, Lucchesi, and Stolfi, 1998)) to represent that information as one string, with the base form coded. The standard coding consists of one character that says how many characters should be deleted from the end of inflected form so that the rest could match the beginning of the base form, and the string of characters that should be appended to the result of the previous operation to form the base form.

Such solution works very well for languages that do not use prefixes or infixes in their flectional morphology, e.g. French. However, in languages like German and Dutch, prefixes and infixes are present in many flectional forms. So to accommodate for this feature, we need 2 additional codes. The first one says what is the position from the beginning of a prefix or infix, the second code - the length of the prefix or infix. For languages that do not use infixes, but do use prefixes, it is possible to omit the position code.

Eliminating Transition Counters

There are two basic ways to eliminate transition counters. One uses a very clever sparse matrix representation (see (Tarjan and Yao, 1979), (Lucchiesi and Kowaltowski, 1993), and (Revuz, 1991)). Apart from eliminating transition counters, it also gives shorter recognition times, as transitions no longer have to be checked one by one – they are accessed directly. However, that method excludes the use of other compression methods, so we will not discuss it here.

The other method (giving the same compression) is to see a state not as a set of transitions, but as a list of transitions ((Kowaltowski, Lucchesi, and Stolfi, 1993)).

We no longer have to specify the transition count provided that each transition has a 1-bit flag indicating that it is the last transition belonging to a particular state (see fig. 6.2). That bit can be stored in the same space as the pointer to the target state, along with the final marker. We can combine that method with others.

	L	F	S	$\rightarrow$
1	s		●	2
2	t			4
3	a		●	5
4	a		●	5
5	y	●	●	0

Figure 6.2: States seen as lists of transitions. *S* is the marker for the last transition in the state.

$$\tau^{sb}(A) = |E|,\ \pi^{sb}(A) = bytes(|\Sigma|)$$

Transition Sharing

If we look at the figure 6.1, we can see that we have exactly the same transition twice in the automaton. However, once it is part of a state with 2 different transitions, and another time it is part of a state that has only 1 transition. As the information about state boundaries is not stored in the transitions belonging to the given state, we can share transitions between states (on the left on fig. 6.3). More precisely, a smaller state (with a smaller number of outgoing transitions) can be stored in a bigger one. It is also possible to place transitions of a state so that part of them falls into one different state, and the rest into another one. This is possible only when we keep the transition counters, so we will not discuss that further.

	L	F	#	$\rightarrow$
1	s		2	2
2	t		1	3
3	a		1	4
4	y	●	0	0

	L	F	S	$\rightarrow$
1	s		●	2
2	t			3
3	a		●	4
4	y	●	●	0

Figure 6.3: Two transitions (number 3 and 4 from figures 6.1 and 6.2) occupy the same space. Version with counters on the left, with lists – on the right.

In the version that uses lists of transitions, exactly one of the transitions belonging to a state holds information about one state boundary. The other boundary is defined by the pointer in transitions that lead to the state. If all transitions of a smaller state **A** are present as the last transitions of a bigger state **B**, then we can still store **A** inside **B** (on the right on fig. 6.3).

	L	F	N	#	→
1	s		•	2	
2	t			1	3
3	a		•	1	
4	y	•		0	0

	L	F	S	N	→
1	s		•	•	2
2	t				3
3	a		•	•	4
4	y	•	•		0

Figure 6.4: The next flag with sharing of transitions. Version with counters on the left, with lists – on the right. *N* represents the next flag.

Next Pointers

Tomasz Kowaltowski *et al.* ((Kowaltowski, Lucchesi, and Stolfi, 1993)) note that most states have only one incoming and one outgoing transition, forming chains of states. It is natural to place such states one after another in the data structure. We call a state placed directly after the current one the *next state*. It has been observed in (Kowaltowski, Lucchesi, and Stolfi, 1993) that if we add a flag that is on when the target state is the next one, and off otherwise, then we do not need the pointer for transitions pointing to the next states. In case of the target being the next state, we still need a place for the flags and markers, but they take much less space (not more than one byte) than a full pointer. In case of the target state not being the next state, we use the full pointer. We need one additional bit in the pointer for the flag. Usually, we can find that space. In our implementation, we used the *next flag* only on the last transition of a state. Therefore, the representation of our example automaton looks like that given on figure 6.4.

The maximum number of transitions that can use next pointers is equal to the number of states in the automaton minus one, i.e. the initial state. The reason for this is that only one transition leading to a given state may be placed immediately in front of it in the automaton.

The transitions in states having more than one outgoing transition can be arranged in such a way that a transition leading to the next state in the automaton may not be the last one. However, if for a given transition its source state has exactly one outgoing transition, and its target state has exactly one incoming transition, the transition must use the next pointer.

Assuming $p, q, r \in Q$, and $a, b \in \Sigma$, we have:

$$|\{(p,a,q) : ((\forall_{(p,b,r)\in E} b = a, r = q) \wedge (\forall_{(r,b,q)\in E} b = a, r = s))\}| \leq \tau^{np}(A)$$

$$\tau^{np}(A) < |Q|, \quad \pi^{np}(A) = bytes(|E|) - 1$$

Tails of States

In subsubsection 6.2.1 we assumed that only entire states can share the same space as some larger states. When using the list representation (subsubsection 6.2.1), we can share only parts of states. We can have two or more states that share some but not all of their last transitions.

Let us consider a more complicated example (fig. 6.5). The transition number 4 holds in fact 3 identical transitions. The states reachable from the start state have both 2 transitions. One of those transitions is common to both states (the one with label *a*). The second one is different (either labeled with *l* or *t*). To avoid confusion, we did not use the *next* flag.

	L	F	S	$\rightarrow$	$\hookrightarrow$
1	p			3	
2	s		•	5	
3	l			4	
4	a		•	6	
5	t			4	4
6	y	•	•	0	

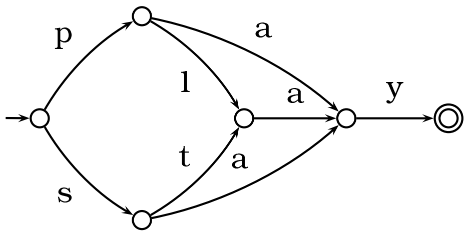


Figure 6.5: Automaton recognizing words *pay*, *play*, *say*, and *stay*, and sharing last transitions of states. $\hookrightarrow$ is a pointer to the tail of the state.

To implement tail sharing we need two things: a new flag (we call it the *tail flag*, not shown on the figure 6.5 as its value is implied), and an additional pointer occurring only when the tail flag is set. When the flag is set, then only the first transitions are kept, and the additional pointer points to the first transition of the tail shared with some other state. We need 1 bit for the flag, and we allocate a place for it in the bytes of transition pointers.

Changing the Order of Transitions

In the examples we showed so far, nothing was told about the order of transitions in a state. Techniques of transition sharing depend on the order of transitions. Automata construction algorithms (e.g. (Daciuk et al., 2000)) may impose initial ordering, but it may not be the best one. Kowaltowski *et al.* ((Kowaltowski, Lucchesi, and Stolfi, 1993)) propose sorting the transitions on increasing frequency of their labels. They also propose to change the order in each state individually. The order of transitions also influences the number of states that are to be considered as *next*. To increase the number of *next pointers*, we try to change the order of transitions in states that do not already have that compression. To increase transition sharing, we put every possible set of n transitions (starting from the biggest n) of every state into a register, and then look for states and tails of states that match them.

Other Techniques

There are other techniques that we have not experimented with. They include local pointers and indirect pointers. Local pointers are only mentioned in (Lucchiesi and Kowaltowski, 1993). We can only stipulate that what they propose is 1-byte pointers for states that are located close in the automaton, and full-length pointers for other states. A flag is needed to differentiate among the two. Indirect pointers are proposed in US patent 5,551,026 granted August 27, 1996 to Xerox. By putting pointers to frequently referenced locations into a vector of full-length pointers, and replacing those pointers in transitions with (short) indexes in the vector, one can gain more space.

6.2.2 Experiments

Data

Our experiments were carried out on morphological dictionaries for German and Dutch. The German morphological dictionary by Sabine Lehmann contains 3,977,448 entries. The automaton for the version with coded suffixes had 307,799 states (of which 62,790 formed chains), and 436,650 transitions. The version with coded suffixes, prefixes, and infixes had 107,572 states (of which 14,213 formed chains), and 176,421 transitions.

Results

		O	N	NM	NO	NMO
sg	2,178,268	2,117,628	1,747,210	1,733,730	1,706,512	1,694,405
sgi	882,123	822,133	731,713	718,439	692,514	681,007
cd	3,028,893	2,914,528	2,369,485	2,331,985	2,287,664	2,257,565
cdi	2,873,143	2,758,473	2,255,753	2,221,047	2,183,489	2,147,982

	S	SO	SMO	SN	SNM	SNO
sg	1,742,616	1,720,460	1,703,376	1,311,558	1,298,078	1,300,592
sgi	705,700	682,932	668,084	555,290	542,016	544,696
cd	2,423,116	2,382,936	2,350,584	1,763,708	1,726,208	1,737,200
cdi	2,298,516	2,257,728	2,225,540	1,681,126	1,646,420	1,663,904

	SNMO	STO	STMO	SNTO	SNMT	SNMTO
sg	1,282,648	1,703,292	1,690,229	1,507,949	1,511,461	1,495,909
sgi	528,814	666,202	654,613	531,578	542,016	523,299
cd	1,697,742	2,939,005	2,904,985	1,985,335	1,983,531	1,959,747
cdi	1,619,334	2,779,723	2,745,299	1,902,887	1,894,999	1,876,403

Table 6.1: Size of automata built with various options, Sabine Lehmann's German morphology , and CELEX Dutch morphology, in bytes. In the table, *sg* means Sabine Lehmann's German morphology with coded suffixes, *cd* – CELEX Dutch, *i* – coded prefixes and infixes, *O* – shared transitions, *S* - stop bit (lists of transitions), *N* – next pointers, *T* – tails of states, and *M* – changing the order of transitions.

Table 6.1 gives the size of automata built with various options. The *sg* automata with *SNTO*, *SNMT*, and *SNMTO* are bigger than expected because there was no space for one more flag in the pointer.

Conclusions

In case of German morphology, we managed to compress the initial automaton more than fourfold. With coded infixes and prefixes, we compressed the input data more than 696 times. Gzip compressed the input data (with coded infixes and prefixes) to 16,342,908 bytes. All automata for given input data could be made smaller by using compression by over 40% (43.9% for Dutch). The smallest automaton we obtained could still be compressed with gzip by 27.77%. The best compression method for German turned out to be a good preparation of the input data. It gave savings from 57.66% to 65.02%. For Dutch, those savings were only 4.15-5.50%, as words with prefixes and infixes constituted 3.68% of data, and not 22.93% as in case of German. As predicted, elimination of transition counters gave 20% on average. The figure was higher (up to 25.13% for German, 25.98% for Dutch) when next pointers were also used, as counters took proportionally more space in transitions. The figure was lower (16.93%) when only transition sharing was in use, because distributing a state over two other states was no longer possible. For German, next pointers gave savings from 15.77% to 24.70%, i.e. within the predicted range (3.22% –28.26%). For Dutch: 20.84% – 34.51%. The savings were bigger when the stop bit option was used. Surprisingly, transition sharing is less effective (0.84% – 3.01% on sg, and 1.91% – 7.24% on sgi, 0.98% – 4.45% on Dutch), and works better on sgi because sg contains many chains of states. Compression of tails of states adds only 0.71% to 2.45% for German, and does not work for Dutch. Changing the order of transitions is a solution only for those desperately in need to squeeze out a few bytes more – the method gives small results (up to 2.92%), but is extremely time-consuming.

6.3 Compact Representation of Language Models in NLP

6.3.1 Introduction

An important practical problem in Natural Language Processing (NLP) is posed by the size of the knowledge sources that are being employed. For NLP systems which aim at full parsing of unrestricted texts, for example, realistic electronic dictionaries must contain information for hundreds of thousands of words. In recent years, *perfect hashing* techniques have been developed based on finite state automata which enable a very compact representation of such large dictionaries without sacrificing the time required to access the dictionaries (Lucchiesi and Kowaltowski, 1993; Roche, 1995; Revuz, 1991). A freely available implementation of such techniques is provided by one of us (Daciuk, 2000b; Daciuk, 2000a)[1].

A recent experience in the context of the Alpino wide-coverage grammar for Dutch (Bouma, van Noord, and Malouf, 2001) has once again established the importance of such techniques. The Alpino lexicon is derived from existing lexical resources. It contains almost 50,000 stems which give rise to about 200,000 fully inflected entries in the compiled dictionary which is used at runtime. Using a standard representation provided by the underlying programming language (in this case Prolog), the lexicon took up about 27 Megabytes. A library has been constructed (mostly implemented in C++) which interfaces Prolog and C with the tools provided by the `s_fsa` (Daciuk, 2000b; Daciuk, 2000a) package. The dictionary now contains only 1,3 Megabytes, without a noticeable delay in lexical lookup times.

However, dictionaries are not the only space consuming resources that are required by current state-of-the-art NLP systems. In particular, *language models* containing statistical information about the Co-occurrence of words and/or word

[1]http://www.pg.gda.pl/ jandac/fsa.html`http://www.pg.gda.pl/~jandac/fsa.html`

meanings typically require even more space. In order to illustrate this point, consider the model described in chapter 6 of (Collins, 1999); a recent, influential, dissertation in NLP. That chapter describes a statistical parser which bases its parsing decisions on bigram lexical dependencies, trained from the Penn Treebank. Collins reports:

> All tests were made on a Sun SPARCServer 1000E, using 100% of a 60Mhz SuperSPARC processor. The parser uses around 180 megabytes of memory, and training on 40,000 sentences (essentially extracting the co-occurrence counts from the corpus) takes under 15 minutes. Loading the hash table of bigram counts into memory takes approximately 8 minutes.

A similar example is described in (Foster, 2000). Foster compares a number of linear models and maximum entropy models for parsing, considering up to 35,000,000 features, where each feature represents the occurrence of a particular pair of words.

The use of such data-intensive probabilistic models is not limited to parsing. For instance, (Malouf, 2000) describes a method to learn the ordering of prenominal adjectives in English (from the British National Corpus), for the purpose of a natural language generation system. The resulting model contains counts for 127,016 different pairs of adjectives.

In practice, systems need to be capable to work not only with bigram models, but trigram and fourgram models are being considered too. For instance, an unsupervised method to solve PP-attachment ambiguities is described in (Pantel and Lin, 2000). That method constructs a model, based on a 125-million word newspaper corpus, which contains counts of the relevant $\langle V, P, N_2 \rangle$ and $\langle N_1, P, N_2 \rangle$ trigrams, where P is the preposition, V is the head of the verb phrase, N_1 is the head of the noun phrase preceding the preposition, and N_2 is the head of the noun phrase following the preposition. In speech recognition, language models based on trigrams are now very common (Jelinek, 1998).

For further illustration, a (Dutch) newspaper corpus of 40,000 sentences contains about 60,000 word types; 325,000 bigram types and 530,000 trigram types. In addition, in order to improve the accuracy of such models, much larger text collections are needed for training. In one of our own experiments we employed a Dutch newspaper corpus of about 350,000 sentences. This corpus contains more than 215,000 unigram types, 1,785,000 bigram types and 3,810,000 trigram types. A straightforward, textual, representation of the trigram counts for this corpus takes more than 82 Megabytes of storage. Using a standard hash implementation (as provided by the `gnu` version of the C++ standard library), will take up 362 Megabytes of storage during run-time. Initializing the hash from the table takes almost three minutes. Using the technique introduced below, the size is reduced to 49 Megabytes; loading the (off-line constructed) compact language model takes less than half a second.

All the examples illustrate that the size of the knowledge sources that are being employed is an important practical problem in NLP. The runtime memory requirements become problematic, as well as the CPU-time required to load the required knowledge sources. In this paper we propose a method to represent huge language models in a compact way, using finite-state techniques. Loading compact models is much faster, and in practice no delay in using these compact models is observed.

6.3.2 Formal Preliminaries

In this paper we attempt to generalize over the details of specific statistical models that are employed in NLP systems. Rather, we will assume that such models are composed of various functions from tuples of strings to tuples of numbers. Each such *language model function* $T^{i,j}$ is a finite function $(W_1 \times \ldots \times W_i) \rightarrow (Z_1 \times \ldots \times Z_j)$. The word columns typically contain words, word meanings, the names of dependency relations, part-of-speech tags and so on. The number columns typically contain counts, the cologarithm of probabilities, or other numerical information such as *diversity*.

For a given language model function $T^{i,j}$, it is quite typical that some of the dictionaries $W_1 \ldots W_i$ may in fact be the same dictionary. For instance, in a table of bigram counts, the set of first words is the same as the set of second words. The technique introduced below will be able to take advantage of such shared dictionaries, but does not require that the dictionaries for different columns are the same. Naturally, more space savings can be expected in the first case.

6.3.3 Compact Representation of Language Models

A given language model function $T^{i,j} : (W_1 \times \ldots \times W_i) \rightarrow (Z_1 \times \ldots \times Z_j)$ is represented by (at most) i perfect hash finite automata, as well as a table with $i + j$ rows. Thus, for each W_k, we construct an acyclic finite automaton out of all words found in W_k. Such an automaton has additional information compiled in, so that it implements perfect hashing ((Lucchiesi and Kowaltowski, 1993),(Roche, 1995),(Revuz, 1991)). The perfect hash automaton (fig. 6.6) converts between a word $w \in W_k$ and a unique number $0 \leq |W_k| - 1$. We write $N(w)$ to refer to the *hash key* assigned to w by the corresponding perfect hash automaton.

If there is enough overlap between words from different columns, then we might prefer to use the same perfect hash automaton for those columns. This is a common situation in n-grams used in statistical natural language processing.

We construct a table such that for each $w_1 \ldots w_i$ in the domain of T, where $T(w_1 \ldots w_i) = (z_1 \ldots z_j)$, there is a row in the table consisting of $N(w_1), \ldots, N(w_i)$, $z_1, \ldots, z_j$. Note that all cells in the table contain numbers. We represent each such number on as few bytes as are required for the largest number in its column. The representation is not only compact (a number is typically represented on 2 instead of 8 bytes on a 64 bit architecture), but it is machine-independent (in our implementation, the least significant byte always comes first). The table is sorted. So a language model function is represented by a table of packed numbers, and at most i perfect hash automata converting words into the corresponding hash keys.

The access to a value $T(w_1 \ldots w_n)$ involves converting the words $w_1 \ldots w_n$ to their hash keys $N(w_1) \ldots N(w_n)$ using perfect hashing automata; constructing a query string from the hash keys by packing these hash keys; and using a binary search

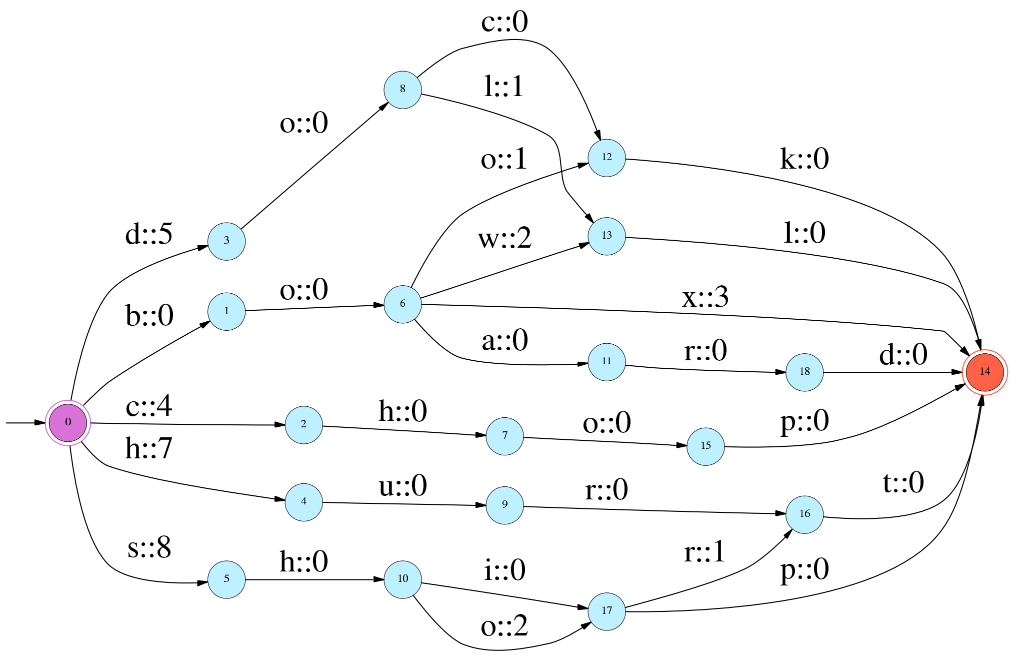


Figure 6.6: Example of a perfect hash automaton. The sum of numbers along transitions recognizing a given word give the word number (hash key). For example, *doll* has number 5+0+1+0=6.

for the query string in the table; $T(w_1 \ldots w_n)$ is then obtained by unpacking the values found in the table.

There is a special case for language model functions $T^{i,j}$ where $i = 1$. Because the words are unique, their hash keys are unique numbers form $0 \ldots |W_1| - 1$, and there is no need to store the hash key of the words in the table. The hash key just serves as an index in the table. Also the access is different than in the general case. After we obtain the hash key, we use it as the address of the numerical tuple.

6.3.4 Preliminary Results

We have performed a number of preliminary experiments. The results are summarized in table 6.2. The *text* method indicates the size required by a straightforward textual representation. The *old* methods indicate the size required for a straightforward Prolog implementation (as a long list of facts) and a standard implementation of hashes in C++. It should be noted that a hash would always require at least as much space as the *text* representation. We compared our method with the hash-map datastructure provided by the gnu implementation of the C++ standard library (this was the original implementation of the knowledge sources in the bigram POS-tagger, referred to in the table).[2]

[2]The sizes reported in the table are obtained using the Unix command `wc -c`, except for the size of the hash. Since we did not store these hashes on disk, the sizes were estimated from the increase

test set	text	old		concat dict	new
		Prolog	C++ hash		
Alpino tuple	9,475	44,872	NA	4,636	4,153
20,000 sents trigram	5,841	32,686	27,000	6,399	2,680
40,000 sents trigram	11,320	61,672	52,000	11,113	4,975
20,000 sents fourgram	8,485	45,185	33,000	13,659	3,693
40,000 sents fourgram	16,845	88,033	65,000	20,532	7,105
POS-tagger	15,722	NA	45,000	NA	4,409

Table 6.2: Comparison of various representations (in Kbytes)

The *concat dict* method indicates the size required if we treat the sequences of strings as words from a single dictionary, which we then represent by means of a finite automaton. No great space savings are achieved in this case (except for the Alpino tuple) , because the finite automaton representation is able only to compress prefixes and suffixes of words; if these 'words' get very long (as you get by concatenating multiple words) then the automaton representation is not suitable. The final *new* column indicates the space required by the new method introduced in this paper.

We have compared the different methods on various inputs. The *Alpino tuple* contains tuples of two words, two part-of-speech tags, and the name of a dependency relation. It relates such a 5-tuple with a tuple consisting of three numbers. The rows labeled n *sents trigram* refer to a test in which we calculated the trigram counts for a Dutch newspaper corpus of n sentences. The n *sents fourgram* rows are similar, but this case we computed the fourgram counts. Because all words in n-gram tests came from the same dictionary, we needed only one automaton instead of 3 for trigrams and 4 for fourgrams. The automaton sizes for trigrams accounted for 11.84% (20 000 sentences) and 9.33% (40 000 sentences) of the whole new representation, for fourgrams – 8.59% and 6.53% respectively. The automata for the same input data size were almost identical. Finally, the *POS-tagger* row presents the results for an HMM part-of-speech tagger for Dutch (using a tag set containing 8,644 tags), trained on a corpus of 232,000 sentences. Its knowledge sources are a table of bigrams of tags (containing 124,209 entries) and a table of word/tag pairs (containing 209,047 entries).

As can be concluded from the results in table 6.2, the new representation is in all cases the most compact one, and generally uses less than half of the space required by the textual format. Hashes, which are mostly used in practice for this purpose, consistently require about ten times as much space.

6.3.5 Variations and Future Work

We have investigated additional methods to compress and speed-up the representation and use of language model functions; some other variations are mentioned here as pointers to future work.

In the table, the hash key in the first column can be the same for many rows. For

of the memory size reported by `top`. All results are obtained on a 64bit architecture.

trigrams, for example, the first two hash keys may be identical for many rows of the table. In the trigram data set for 20,000 sentences, 47 rows (out of 295,303) have hash key 1024 in the first column, 10 have 0, 233 – 7680. The same situation can arise for other columns. In the same data set, 5 rows have 1024 in the first column, and 29052 in the second column, 16 – 7680 in the first column, and 17359 in the second one. By representing them once, and providing a pointer to the remaining part, and doing the same recursively for all columns, we arrive at a structure called *trie*. In the trie, edges going out from root are labeled with all the hash keys from the first column. They point to vertices with outgoing edges representing tuples that have the same two words at the beginning, and so on. By keeping only one copy of hash keys from the first few columns, we hope to economize the storage space. However, we also need additional memory for pointers. A vertex is represented as a vector of edges, and each edge consists of two items: the label (hash key), and a pointer. The method works best when the table is dense, and when it has very few columns. We construct the trie only for the columns representing words; we keep the numerical columns intact (obviously, because it is “output”).

0	2	4	1
0	15	4	3
20	7	50	1
20	7	53	2
20	15	4	2

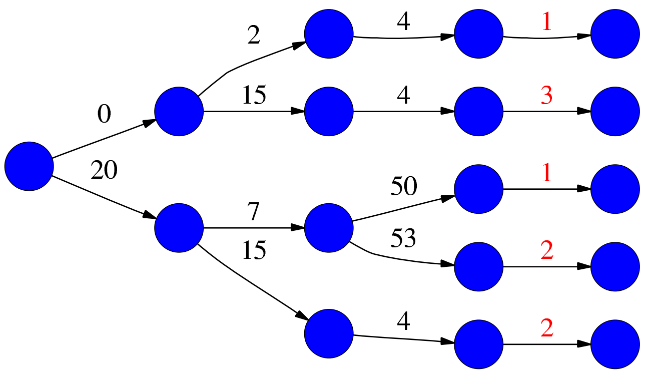


Figure 6.7: Trie (right) representing a table (left). Red labels represent numerical tuples. Numbers 0 and 20 from the first column, and 7 from the second column, are represented only once.

For dense tables, we may perceive the trie as a finite automaton. The vertices are states, and the edges – transitions. We can reduce the number of states and transition in the automaton by minimizing it. In that process, isomorphic subtrees of the automaton for the word columns are replaced with single copies. This means that additional sharing of space takes place. However, we need to determine which paths in the automaton lead to which sequences of numbers in the numerical columns. This is done, again, by means of *perfect hashing*. This implies that each transition in the automaton not only contains a label (hash key) and a pointer to the next state, but also a number which is required to construct the hash key. Although we share more transitions, we need space for storing those additional numbers.

We use a sparse matrix representation to store the resulting minimal automaton. The look-up time in the table for the basic model described in the previous subsubsection is determined by binary search. Therefore, the time to look-up a

0	2	4	1
0	15	4	3
20	7	50	1
20	7	53	2
20	15	4	2

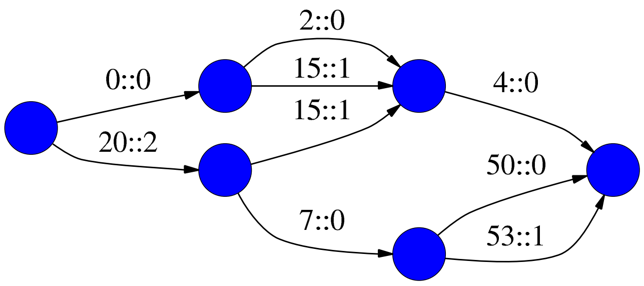
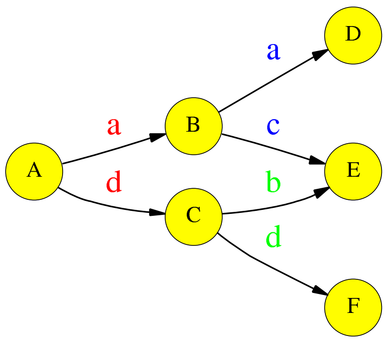


Figure 6.8: Perfect hash automaton (right) representing a table (left). Only word columns are represented in the automaton. Numerical columns from the table are left intact. They are indexed by hash keys (sums of numbers after "::" on transitions). The first row has index 0.

tuple is proportional to the binary logarithm of the number of tuples. It may be possible to improve on the access times by using interpolated search instead of binary search. In an automaton, it is possible to make the look-up time independent from the number of tuples. This is done by using the sparse matrix representation ((Tarjan and Yao, 1979)) applied to finite-state automata ((Revuz, 1991)). A state is represented as a set of transitions in a big vector of transitions for the whole automaton. We have a separate vector for every column. This allows us to adjust the space taken by pointers and numbering information. The transitions do not have to occupy adjacent space; they are indexed with their labels, i.e. the label is the transition number. As there are gaps between labels, there are also gaps in the representation of a single state. They can be filled with transitions belonging to other states, provided that those states do not begin at the same point in the transition vector. However, it is not always possible to fill all the gaps, so some space is wasted.



0	a D		a D	
1	a B	a B		
2	c E		c E	
3	b E			b E
4	d C	d C		
5	d F			d F

Figure 6.9: Sparse table representation (right) of a part of an automaton (left). Node A has number 1, B – 0, C – 3. The first column is the final representation, column 2 – state A, column 3 – state B, column 4 – state C.

Results on the representation of language model functions using minimal automata for word tuples and sparse matrix representation are discouraging. If we take the word tuples, and create an automaton with each row converted to a string of transitions labeled with hash keys from successive columns, and then minimize that automaton, and compare the number of transitions, we get from 27% to 44% reduction. However, the transition holds two additional items, usually of the same size as the label, which means that it is 3 times as big as a simple label. In the trie representation, we don't need numbering information, so the transition is twice as big as the label, but the automaton has even more transitions. Also, the sparse matrix representation introduces additional loss of space. In the our experiments, 32% to 59% of space in the transition vector is not filled. This loss is due to the fact that the labels on outgoing transitions of a state can be any subset of numbers from 0 to over 50,000. This is in sharp contrast with natural language dictionaries, for instance, where the size of the alphabet is much smaller. We also tried to divide longer (i.e. more than 1 byte long) labels into a sequence of 1 byte long labels. While that led to better use of space and more transition sharing, it also introduced new transitions, and the change in size was not significant. The sparse matrix representation was in any case up to 3.6 times bigger than the basic one (table of hash keys), with only minor improvement in speed (up to 5%).

We thought of another solution, which we did not implement. We could represent a language model function $T^{i,j}$ as an i-dimensional array $A[1,\dots,i]$. As before, there are perfect hashing automata for each of the dictionaries $W_1 \dots W_n$. For a given query $w_1 \dots w_n$, the value $[N(w_1),\dots,N(w_n)]$ is then used as an index into the array A. Because the array is typically very sparse, it should be stored using a sparse matrix representation. It should be noted that this approach would give very fast access, but the space required to represent A is at least as big (depending on the success of the sparse matrix representation) as the size of the table constructed in the previous method.

6.3.6 Conclusions

We have presented a new technique for compact representation of language models in natural language processing. Although it is a direct application of existing technology, it has great practical importance (numerous examples are quoted in the introduction), and we have demonstrated that our solution is the answer to the problem. We also show that a number of more sophisticated and scientifically appealing techniques are actually inferior to the basic method presented in the paper.

6.4 Incremental Minimization

6.4.1 Introduction

An algorithm ((Watson, 2001)) presented by Bruce Watson at FSMNLP 2001 workshop in Helsinki minimizes an automaton in such a way that the intermediate results are usable. This is in contrast to other known minimization algorithms. Although it has worse computational complexity than the best known methods, introduction of full memoization and a few other techniques make it not as bad as reported in (Watson, 2001).

We will start with the sketch of the original algorithm, then we will introduce our improvements. We will end the paper with a discussion of the results.

6.4.2 The Original Algorithm

A deterministic finite automaton (DFA) is defined as a quintuple $(Q, \Gamma, \delta, q_0, F)$, where Q is a set of states, Γ is the alphabet (we use non-standard Γ instead of Σ for compatibility with (Watson, 2001)), $\delta \in Q \times \Gamma \longrightarrow Q \cup \{\perp\}$ is the transition function ($\perp$ designates the invalid state), $q_0 \in Q$ is the start state, and $F \subseteq Q$ is the set of final states. $\Gamma_q = \{a | a \in \Gamma \wedge \delta(q, a) \neq \perp\}$ is the set of labels on out-transitions from q. $\mathcal{L}(M)$ is the language of the automaton. The size $|M|$ of an automaton M is the number of states in it $|M| = |Q|$. δ^*, defined as $\delta^*(q, \epsilon) = q$, $\delta^*(q, aw) = \delta^*(\delta(q, a), w)$ is the extended transitions function.

For every DFA there is a minimal DFA M such that

$$\nexists_{M'} \mathcal{L}(M) = \mathcal{L}(M') \wedge |M'| < |M|$$

Provided the automaton does not contain useless states, the minimality can also be stated in terms of the right languages of states:

$$\forall_{p,q \in Q} \overrightarrow{\mathcal{L}}(p) = \overrightarrow{\mathcal{L}}(q) \Rightarrow p = q$$

where the right language is defined as:

$$\overrightarrow{\mathcal{L}}(q) = \{w \in \Sigma^* | \delta^*(q, w) \in F\}$$

The right language can also be defined recursively:

$$\overrightarrow{\mathcal{L}}(q) = \left[\bigcup_{a \in \Gamma_q} a \overrightarrow{\mathcal{L}}(\delta(q, a)) \right] \cup \begin{cases} \{\epsilon\} & \text{if } q \in F \\ \emptyset & \text{if } q \notin F \end{cases}$$

The recursive definition can be modified to compare the identity of states in out-transitions rather than their right languages, provided that we can make sure that they are the only states in the automaton that have the same right language. This can easily be done in algorithms dealing with acyclic automata, which leads to efficient algorithms ((Daciuk et al., 2000)). In cyclic automata, we cannot satisfy that condition. However, we can detect cycles, and if the structure of a pair of cycles is the same, they are equivalent. This leads to the algorithm in (Watson, 2001).

Initially, pairs $F \times (Q - F)$ and $(Q - F) \times Q$ are recognized as different (a final state cannot be equivalent to a non-final one), and only pairs $F \times F$ and $(Q - F) \times (Q - F)$ are checked. Variable k is initially set to $|Q| - 2$. Variable S keeps track of pairs compared in the current call.

6.4.3 Improvements

We describe three improvements to the original algorithm:

1. we pre-sort the states on their finality, the number of out-transitions, and labels on those transitions;

```
(1)    func equiv($p, q, k$) $\rightarrow$
(2)        if $k = 0 \rightarrow eq := true$
(3)        || $k \neq 0 \wedge \{p, q\} \in S \rightarrow eq := true$
(4)        || $k \neq 0 \wedge \{p, q\} \notin S \rightarrow$
(5)            $eq := (p \in F \equiv q \in F) \wedge (\Gamma_p = \Gamma_q)$;
(6)            $S := S \cup \{\{p, q\}\}$;
(7)            for $a : a \in \Gamma_p \cap \Gamma_q \rightarrow$
(8)                $eq := eq \wedge$ equiv($\delta(p, a), \delta(q, a), k - 1$)
(9)            rof;
(10)           $S := S \setminus \{\{p, q\}\}$
(11)       fi;
(12)       return $eq$
(13)   cnuf
```

Figure 6.10: Comparison of a pair of states in the incremental minimization algorithm in Watson 2001

2. we put into S only pairs of states that can potentially start a cycle;
3. we introduce full memoization and demonstrate that the complexity of the algorithm is $\mathcal{O}(|Q|^3)$.

Pre-sorting

The original algorithm divides states into 2 classes: final and non-final states. Only pairs of states each belonging to the same class are tested for equivalence.

We propose to divide the set of states into more (numbered) classes. The criterion includes not only the finality of states, but also the number of out-transitions, and their labels. This can be done in $\mathcal{O}(|Q|)$ time using bucket sort[3].

It brings several advantages:

- as we compare the states pairwise only in their classes (states from two different classes are not equivalent), there are fewer comparisons, e.g. if we divide the set into n roughly equally numerous classes, we perform n times fewer comparisons;
- instead of comparing the number of out-transitions and their labels in line 5 of the algorithm, we compare only classes of states (i.e. two integer numbers)[4];
- we need less memory to represent non-equivalent states, as there is no need to remember that states belonging to different classes are not equivalent.

[3]Thanks to Bruce Watson for pointing this out to me in personal correspondence. He also reminded me that Lauri Karttunen mentioned some kind of pre-sorting during discussion after Bruce's talk at FSMNLP 2001. Bruce also says he uses pre-sorting on the number of out-transitions in his implementation.

[4]This is impossible if we sort only on finality and number of out-transitions.

If all states in the automaton have the same number of transitions, and the transitions have the same labels (for example in a complete automaton), there is nothing to gain with pre-sorting.

Remembering Only Reentrant States on Stack

In line 6 of the original algorithm, the currently examined pair of states is added to variable S (the stack of pairs of states – predecessors of the currently examined pair), and in line 10, the pair is removed from S.

The purpose of variable S is to detect cycles. There are two ways in which those cycles can be started:

- a single chain of transitions may lead from the initial pair back to it – the cycle is started at the initial pair;
- the cycle starts later on, but it can only start from a state that has more than one in-transition; one in-transition leads from a path from a state in the initial pair (outside the cycle), so there must be at least another in-transition from within the cycle.

There is no need to add to the stack pairs of states that cannot start a cycle. It is guaranteed that we cannot revisit such a pair during the same call to *equiv* – we would have to revisit a pair that starts a cycle first (and then there would be no need to follow out-transitions, as the result would already be known).

The profit from the improvement is twofold:

- we use less memory to represent S in memory efficient implementations,
- in some memory-efficient implementations, searching S takes $\mathcal{O}(\log |S|)$ time, contributing that factor to the overall complexity, so shortening S means also shortening the time.

If (almost) all states of the automaton have more than one in-transition, this modification does not reduce the running time, nor memory requirements.

Full Memoization

In the original paper ((Watson, 2001)), Bruce Watson determines worst-case computational complexity as $\mathcal{O}(\Gamma^{(|Q|-2)max0})$ even when partial memoization is used. When full memoization is used, the complexity is actually $\mathcal{O}(|Q|^2 G(|Q|^2)$, where $G(n)$ is the inverse of Ackermann's function – far better than exponential. $G(n) \leq 5$ for all "practical" values of n, i.e. for all $n \leq 2^{65536}$ (see (Aho, Hopcroft, and Ullman, 1974b) for details).

A call to function *equiv* can return only two results: *true* or *false*. However, we can immediately memoize[5] only the non-equivalence. Storing and retrieving that

[5]We memoize the result only with respect to the pair of states; we ignore the depth of recursion. However, we call it *full* memoization here, as the depth of recursion is irrelevant for evaluating equivalence of a pair of states

information can be done in constant time in two-dimensional arrays indexed with states numbers.

If function *equiv* returns *true*, the result can either be conclusive, or inconclusive. If the result depends on equivalence of a certain pair of states that is still under evaluation, i.e. it is still in variable S, the result is inconclusive.

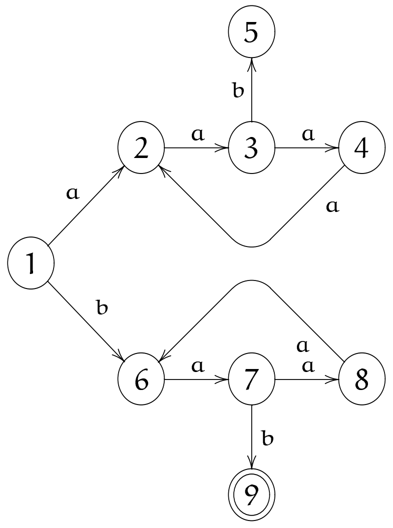


Figure 6.11: Inconclusive comparison of states 4 and 8, resulting from evaluation of the pair 2 and 6.

Consider the automaton on fig. 6.11. Function *equiv* has been called to determine the equivalence of states 2 and 6. It depends on the equivalence of the pair of states $\{3, 7\}$, so that pair is visited as well. Assuming that we traverse the transitions in lexicographical order, the algorithm goes on to examine the pair $\{4, 8\}$. From $\{4, 8\}$, it moves to $\{2, 6\}$ to see that that pair is already in S, so the call to *equiv*(2, 6, 3) with $S = \{\{2, 6\}, \{3, 7\}, \{4, 8\}\}$ returns *true*. However, neither in $\{2, 6\}$, nor in $\{4, 8\}$ can we say that those pairs are equivalent. They are only potentially equivalent. The pair $\{2, 6\}$ is still under investigation – we have not checked whether $\{3, 7\}$ is equivalent (the call to *equiv*(3, 7, 6) has not returned any value yet), and the pair $\{4, 8\}$ depends on that evaluation. In case of $\{2, 6\}$ we could simply wait until the nested calls return – we will know the result when it has been computed. However, in case of $\{4, 8\}$ we have no other call to *equiv* higher up.

We remember that a pair of states $\{p, q\}$ depends on another pair $\{r, s\}$ higher up in the hierarchy of recursive calls to function *equiv*. When the control returns to the first call with the $\{r, s\}$ as the argument, there are two possibilities. Either the pair does not depend on any other pair higher up in the hierarchy (it can depend on itself), or such dependence exists. In the first case, all pairs that depend on $\{r, s\}$ are either merged (if pair $\{r, s\}$ is equivalent), or remembered as non-equivalent (if pair $\{r, s\}$ is not equivalent). If the pair $\{r, s\}$ depends on another pair, a new dependency is stored. To prevent formation of chains of dependencies, we use the UNION-FIND algorithm ((Aho, Hopcroft, and Ullman, 1974b)).

A call to function *equiv* with $\{p, q\}$ as an argument may create various situations:

- $\{p, q\}$ is in fact $\{p, p\}$ – *equiv* returns conclusive true;
- $\{p, q\}$ are already remembered as non-equivalent – *equiv* returns false;
- $\{p, q\}$ is already under evaluation higher up in the call hierarchy (it is in variable S) – *equiv* returns inconclusive true, making all pairs up to and including initial $\{p, q\}$ dependent at least on $\{p, q\}$;
- $\{p, q\}$ is not in S, but it is on a list of dependent states for a pair of states $\{r, s\}$ in S – *equiv* returns inconclusive true, making all pairs up to and including $\{r, s\}$ depend on at least $\{r, s\}$. Dependencies for pairs are stored in two-dimensional arrays as pointers to structures containing the representative (the pair highest in the hierarchy), and a list of dependant pairs.

The UNION-FIND algorithm is also used to solve the problem of chains of equivalent pairs of states. To efficiently implement full memoization, we also need to represent non-equivalence relation, and the variable S – pairs under investigation. The non-equivalence relation can simply be stored in a two dimensional array. To save space, the array can be divided in sub-arrays for each class, or vectors for each state created on demand. We need variable S to know the level of recursion of function *equiv* for a pair of states. Therefore, the variable can be implemented as a two dimensional array, with values being the depth of recursion for respective pairs.

6.4.4 Final algorithm

Figure 6.12 shows the final version of the algorithm. Due to inadequacy of Dijkstra's notation to express larger chunks of code, we decided to use normal pseudocode. Depending on memory limitations and speed requirements, certain operations can be implemented in different ways. We have provided two implementations: one making use of two-dimensional arrays (achieving desired complexity), and another using tree structures (with $\log|Q|$ overhead). It is possible to use hash tables in an implementation. This would combine the speed of the first solution with memory efficiency of the other one.

When memoization is used, function *equiv* can still be called more than once with the same pair of states. If it has already been called with the same pair, lines 2-7 check that and the control never reaches line 8. In line 2, states are compared for identity. Although at the top level, *equiv* is never called with a pair of identical states, the states might be identical on other levels. The check is done in constant time.

Line 3 tests whether the states are either both final, or both non-final, and whether they have the same number of transitions, and the same labels on those transitions. It is equivalent to line 5 of the original algorithm. However, because of pre-sorting, there is no need to compare so many features each time; only class numbers need to be compared. The check is done in constant time.

Line 4 is the same as in the original algorithm, except for variable rl that is set to 0. The variable indicates the level of recursion at which a pair of states can be found that the current pair depends upon. If S is seen as stack, then variable rl is an index in that stack, indicating a position of the pair the current pair depends on. If the current pair depends on nothing, variable rl is set to $|Q|$. If the comparison has done a full cycle, it means that the current pair of states depends on the initial pair of states (the one from the top level call). This check is done in constant time. Line 5 belongs to the same category – it also tests for cycles. If a cycle has been detected, the level of the pair on which the current pair depends is stored in global variable *rl*. That variable propagates the dependency upwards. This check can be done in constant time .

In line 6, it is checked whether the pair is already known not to be equivalent. Variable G is adopted for that purpose from the original paper. The check can be done in constant time.

In line 7, it is checked whether the pair has already been found to depend on another pair higher up in the hierarchy. While searching for the pair, the dependency chains are updated (see subsubsection 6.4.3). This can be done in $\mathcal{O}(G(|Q|))$ time ($|Q|$ is the maximal depth of recursion, so also the maximal height of dependency tree). This concludes memoization checking.

Lines 9-11 update the value of global variable S. Only the pair of states from the top level calls, and pairs where at least one state has more than one in-transition, need to be put into S. The update can be done in constant time. The pair is removed in lines 18-20, which can also be done in constant time.

In lines 12-17, it is checked whether the states are equivalent by checking the equivalence of the targets of their out-transitions. This is the central part of the original algorithm. As variable rl is global, so it is used to convey any dependency upwards in the call hierarchy. Variable rl' is local, so it can store the local value between the nested calls.

If the current pair of states has been found to be equivalent, then the result can be conclusive or not. If the result is positive and conclusive, i.e. the equivalence of $\{p, q\}$ does not depend on equivalence of any other pair of states, then the states are merged (line 23). It is done using the UNION-FIND algorithm with complexity $O(G(|Q|))$. If the result is positive, but not conclusive, then the result must be stored along with the pair it depends on (line 25). It is appended to the list for appropriate level, and a mapping from $\{p, q\}$ to that other pair is stored. If the pair is not equivalent, it is added to the set of non-equivalent pairs G in line 28. If G is a two-dimensional array, this can be done in constant time.

There can be pairs of states that depend on the current pair of states $\{p, q\}$. They are stored in $P[level]$, where $level$ is the recursion level (or the number of items in S) for the current pair. If $\{p, q\}$ are found to be equivalent, then if the result is conclusive, the pairs are merged. There can be more than one pair of states in $P[level]$, but the operation cannot be invoked more than $|Q|^2$ times across all calls. The operation can be performed in time proportional to the number of items in $P[level]$. If the result is not conclusive, the pairs are merged with the pairs at $P[rl]$. Appending a list to another list (moving it to the end of another list – without copying) can be done in constant time. If $\{p, q\}$ are found not to be equivalent, the contents of $P[level]$ is stored in G – again it is possible to do that in constant time for each pair. There can be at most $|Q| \cdot (|Q| - 1)/2$ pairs across all calls, so the cost per pair, or per average call, is constant.

Function *equiv* is called at most $|Q|^2 \max_{q \in Q} |\Gamma_q|$ times. One call (without nested calls) can be completed in constant time, except for certain operations on lists stored in P. However, all of these operations take at most $\mathcal{O}(|Q|^2)$ time across all calls, except for the UNION-FIND algorithm that applied to both dependencies and equivalent states takes $|Q|^2 G(|Q|^2)$ across all calls, so the final complexity is $\mathcal{O}(|Q|^2 G(|Q|^2))$.

6.4.5 Results

The performance was checked for various implementations of the algorithm. Not only the speed of execution was measured, but also some additional characteristics, like the maximal depth of recursion of function *equiv*, the maximal number of items in variable S, and the number of calls to function *equiv* excluding memoization. They were measured for several versions of the algorithm: the standard (final) one, one where two-dimensional arrays were replaced with tree structures, one without pre-sorting, and one without full memoization (i.e. the original version).

Input Data

Experiments were performed on four different data sets. The first one contained determinized versions of automata used in (van Noord, 2000b). They originally come from experiments on finite-state approximation of context-free grammars – a real life task. The automata have varied characteristics. However, they are of moderate size, so only the biggest ones are suitable for measuring the effects of various improvements on the speed of the algorithm. Examining the table 6.4, we can see that a version of the algorithm that uses tree structures is usually faster than the one using two-dimensional arrays. A quick look at table 6.3 solves the mystery. The automata have relatively many classes, so the number of items in variable S is low, and initialization for two-dimensional arrays takes more time than the use of tree structures.

The other three sets were generated automatically. The first generated set consisted of large automata with a large number of classes (in the sense defined in the subsubsection 6.4.3). This means that most of the job of minimization was done in the sorting phase. Also for these automata, tree structures were faster than two-dimensional arrays.

The second generated data set consisted of large automata with a fixed low number of classes. However, the automata were already minimal. For this kind of job, most of the work was done by the function *equiv*.

The third generated data set consisted of automata from the second generated set inflated to a larger size. A program was used to create additional states in an automaton without changing its language. The automata still had the same, low number of classes, but they were not minimal. The ratio of the number of states in the non-minimized automaton to the number of states in the minimized automaton was fixed.

Ignoring states that cannot start cycles

This improvement relies on density of automata to be minimized, and more precisely on the proportion of states with less than two in-transitions. In all sets of data, such states were not abundant. We have also measured only the longest sequence put into the stack S; perhaps measuring the average length could give different results.

Results for our data show some improvement, but it is not much. Only in one case, our method produced a stack 41.7% of its original size (10 instead of 24). However, even small improvement is worth considering, and there may exist applications where states with fewer than 2 in-transitions are more common.

Pre-sorting

Introduction of pre-sorting decreased the number of calls to function *equiv()* in a noticeable way. The average improvement was 16.95% for the first data set. As expected, the biggest improvements were for automata with a large number of classes. For those automata (see figure 6.13), the incremental algorithm was faster than Aho-Sethi-Ullman algorithm, and even faster than Hopcroft algorithm! Only in case of automata with only a few classes, pre-sorting increased the execution time. In all other cases, pre-sorting was beneficial.

Full Memoization

The original algorithm used only partial memoization. When comparison of a pair of states was inconclusive, the result was lost, and the same pair could be evaluated many times. This lead directly to exponential times for some automata. Note that the exponential growth is the worst case; it is not obligatory. However, when it does occur, it makes the original version of the algorithm useless.

Because of the exponential complexity of the original algorithm, its performance was measured only for moderate-size real-life automata.

Execution times for the biggest among the small real-life automata are shown in table 6.4. Results for large minimal automata with a limited number of classes (figure 6.14, and results for non-minimal large automata with limited number of classes (figure 6.15) show that the execution time grows more slowly than the worst case computational complexity.

6.5 Constructing Acyclic Finite Automata from Sets of Strings

6.5.1 Motivation

During last 12 years, one could see emergence of construction methods specialized for minimal, acyclic, deterministic, finite-state automata. However, there are various opinions about their performance, and how they compare to more general methods. Only partial comparisons are available. What has been compared so far was complete programs, which performed not only construction, but computation of a certain representation (e.g. space matrix representation, or various forms of compression).

The aim of this paper is to give answers to the following questions:

- What is the fastest construction method?
- What is the most memory-efficient method?
- What is the fastest method for practical applications?
- Do incremental methods introduce performance overhead?

The third question is not the same as the first one, because one has to take into account the size of data and available main memory. Even the fastest algorithm may become painfully slow during swapping.

6.5.2 Construction Methods

Due to lack of space, the construction methods under investigation have only been sketched here. The reader is referred to the bibliography for proper descriptions.

Some of the methods use a structure called the *register* of states. In those algorithms, states in an automaton are divided into those that have been minimized, i.e. they are unique in that part, and other states, i.e. those that are to be minimized. The register is a hash table, and two states are considered equivalent if they are either both final or both non-final, and they have the same transitions (the same number, labels, and targets).

The following methods have been investigated:

1. Incremental construction for sorted data ((Daciuk et al., 2000)). Input strings are lexicographically sorted. For each string, the longest common prefix with the last string added to the automaton is found. If the last state in the path recognizing the common prefix has outgoing transitions, a path consisting in last transitions of subsequent states is followed, and then, starting from the end of that path up to, but excluding, the last state in the common prefix path, if an equivalent state is found in the register, it replaces the current state, and the current state is deleted. Afterwards, the remaining part of the string is added, creating a chain of states attached to the last state of the path of the common prefix. After the last string has been added, the path of that string is also compared against states in the register and replaced if necessary.

2. Incremental construction for unsorted data ((Daciuk et al., 2000)). Input strings come in arbitrary order. For each string, the longest common prefix with any string in the automaton is found. If any state in the common prefix path is reentrant, that state and all states following it in the path are cloned, the first state before any cloned state, or the last state in the common prefix path, is removed from the register. The remaining part of the string is added, creating a chain of states attached to the last state of the path of the common prefix. Afterwards, starting from the end of the chain, if an equivalent state is found, it replaces the current state. The process continues past

the appended chain towards the initial state removing states from the register and adding new states to it, until no replacement is made.

3. Semi-incremental construction by Bruce Watson ((Watson, 1998)). Input strings are sorted on decreasing length. For each string, the longest common prefix with any string in the automaton is found. If the string is a prefix of any string already in the automaton, all states reachable from the end of the prefix path (including the last state in that path) and not in the register are put onto stack so that states with smallest height are on top. Then for successive states popped from the stack, if the register contains en equivalent state, it replaces the current state. After the last word has been added, the remaining states are put onto stack and then replaced if needed.
4. Semi-incremental construction by Dominique Revuz ((Revuz, 1991)). Input strings are lexicographically sorted on their *reversal*. A trie-like structure is built, but the longest common suffix between the current string, and the last string added to the automaton is computed. Whenever possible (states have only one outgoing transition, they are not final), the path (or its part) recognizing the common suffix is shared. That phase does not lead to the minimal automaton, so it is followed by true minimization. States are sorted on their height, and then states of the same height are sorted using bucket sort, and redundant states are removed.
5. Building a trie and minimizing it using the Hopcroft algorithm ((Hopcroft, 1971), (Aho, Hopcroft, and Ullman, 1974a)).
6. Building a trie and minimizing it using the minimization phase from the incremental construction algorithms (postorder minimization).
7. Building a trie and minimizing it using the minimization phase from the semi-incremental algorithm by Dominique Revuz (lexicographical sort).

6.5.3 Data for Evaluation

Data sets for evaluation were taken from the domain of Natural language Processing (NLP). Acyclic automata are widely used as dictionaries. Both word lists, and morphological dictionaries, for German, French, and Polish, as well as a word list for English were used. Word lists and morphological dictionaries have different characteristics. Strings in word lists are usually short, sharing short suffixes. Strings in morphological dictionaries are much longer, with long suffixes shared between entries. The data is summarized in Table 6.5.

Because of huge amounts of memory needed to represent a trie, only sufficiently small parts of data were used for non-incremental methods.

6.5.4 Experiments

In the experiments, the hash function had 10001 possible values. The register was implemented with an overflow area for each hash value being a set class from the C++ standard template library - a tree-like structure.

Several characteristics were measured. They included: execution time, number of states during construction: (maximal number of states, number of states as a function of the number of words during construction), use of the register: (number of calls to find an equivalent state in the register, number of calls to remove a state from the register (only for the incremental construction from unsorted data), number of states with the same hash value per call, register fill ratio).

6.5.5 Results

Memory Requirements

As it is difficult to trace the exact memory requirements of programs under Unix, the number of states of an automaton during construction was measured. The states were represented by the same structure for all algorithms, although certain fields were specific to some algorithms, like i.e. height was specific to Revuz's algorithm. Memory was also used to store transitions. Programs that used register required a constant amount of memory for an empty register, and an amount proportional to the number of states. In addition, Watson's algorithm required memory for a stack of states. Revuz's algorithm did not use the register, but it needed memory for sorting (at least proportional to the number of states). Using Hopcroft's algorithm for minimization of a trie also required memory proportional to the number of states, but much more than a register of states.

Figure 6.16 is representative for memory requirements for various methods. The points labeled "trie" represent non-incremental methods. Memory requirements for the incremental method for unsorted data are identical to those for the sorted method on the same data, and only slightly higher for data sorted for other methods. Initial memory requirements for Watson's algorithm are higher than for a trie made from data sorted lexicographically, as longer words come first. They become lower towards the end of word data, as shorter words trigger minimization. That effect is hard to be noticed for morphological dictionaries.

Execution time

Due to memory memory limitations, and sometimes extremely long execution time, it was impossible to measure the speed of some algorithms on all data. To test the relation between the speed, and the size of the data, each algorithm was tested on 0.1, 0.2, ... 0.9, and on the whole data. In case of Revuz's algorithm, and Watson's algorithm, those parts were sorted accordingly, instead of taking the same number of words from the beginning of the whole file sorted according to the requirements. This is different than measurements of memory requirements, because they were all taken during a single run on the whole appropriate file. Also, due to multiuser, multi-task Unix environment, only processor times were measured, not the elapsed "real" time. It means that the effects of swapping do not show up on diagrams. Only initial values for trie + Hopcroft minimization algorithm are shown to underline differences between other algorithms.

For most data, the trie + postorder minimization method was the fastest. It was slightly faster than the algorithm for sorted data, and in some cases their values are not distinguishable on diagrams. For morphological dictionaries, Revuz's algorithm was faster. This happens because in that data very long common suffixes were present. The INTEX program (Silberztein, 1999) uses Revuz's algorithm without pseudo-minimization phase to save both time and disk space, but annotations are kept short, and their expansions are kept elsewhere.

6.5.6 The Fastest Algorithm

Surprisingly, the fastest construction algorithm is not yet described in literature. This is probably due to its simplicity. We define a deterministic finite state automaton as $M = (Q, \Sigma, \delta, q_0, F)$, where Q is the set of states, Σ is the alphabet, $\delta \in Q \times \Sigma \longrightarrow Q \cup \{\perp\}$ is the transition function, q_0 is the initial state, and $F \subseteq Q$ is the set of final states. A somewhat formally awkward notation of assignment to the delta function in the algorithm below means creating or modifying a transition. This algorithm has exactly the same complexity as both incremental algorithms from (Daciuk et al., 2000).

```
func trie_plus_postorder_minimization;
    start_state := construct_trie;
    Register := $\emptyset$; postorder_minimize(start_state);
    return start_state;
cnuf

func construct_trie();
    start := new state;
    while file not empty
        word := next word form file;
        i := 0; s := start;
        while i <¡ length(word)
            if $\delta(s, word_i) \neq \perp$
                $\delta(s, word_i) :=$ new state;
            fi
            s := $\delta(s, word_i)$; i := i + 1;
        elihw
        $F := F \cup \{s\}$;
    elihw
cnuf

proc postorder_minimize(s);
    foreach $a \in \Sigma : \delta(s, a) \neq \perp$
        postorder_minimize($\delta(s, a)$);
        if $\exists_{q \in Register}\ \delta(s, a) \equiv q$
            $\delta(s, a) = q$;
        else
            Register := Register $\cup\{s\}$;
        fi
    hcaerof
```

corp

6.5.7 Conclusions

- The incremental algorithm for sorted data, and the trie + postorder minimization algorithm are the fastest. The incremental algorithm should always be used for sorted data. For unsorted data, the trie + postorder minimization algorithm is the fastest provided enough memory is available. Revuz's algorithm is the fastest for morphologies.

- Both fully incremental algorithms are the most economical in their use of memory. They are orders of magnitude better than other, even semi-incremental methods.

- Both incremental algorithms are the fastest in practical applications, i.e. for large data sets. The algorithm for sorted data is the fastest, but if data is not sorted, sorting it (and storing it in memory) may be more costly than using the algorithm for unsorted data.

- It seems that incremental algorithms do not introduce overhead when compared to non-incremental methods. The differences between the incremental sorted data algorithm and its non-incremental counterpart are minimal. The non-incremental version of the semi-incremental Revuz's algorithm (trie + lexical sort) is faster than the original version for words, and slower for morphologies.

- Trie + Hopcroft minimization is the slowest algorithm. While all other algorithms are linear, this one has an additional $\mathcal{O}(\log(n))$ overhead, and it is quite complicated compared to register-based algorithms.

- There is no real reason to use Revuz's algorithm, Watson's algorithm, trie + lexicographical sort (Revuz's algorithm without suffix compression), or trie + Hopcroft minimization algorithm. They are slower, and they have higher memory requirements, than incremental algorithms.

6.6 Incremental Addition of Strings to Cyclic Finite Automata

6.6.1 Introduction

(Carrasco and Forcada, 2002) present an algorithm for incremental addition of strings into a minimal, cyclic, deterministic, finite-state automaton. The algorithm can be seen as an extension of the algorithm for sorted data, the second algorithm in (Daciuk et al., 2000), for cyclic automata. It turns out that not only the second algorithm in (Daciuk et al., 2000), but also the first one can be extended in the same way. That extension is presented in Subsubsection 6.6.3.

The title of Carrasco and Forcada's paper refers to incremental *construction* and maintenance of cyclic automata. The algorithms they provide do not *construct* cyclic automata; they only add words to them. They do not provide a method to *incrementally* construct a cyclic automaton from scratch.

6.6.2 Mathematical Preliminaries

We define a deterministic finite-state automaton as $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite set of symbols called the alphabet, $q_0 \in Q$ is the start (or initial) state, and $F \subseteq Q$ is a set of final (accepting) states. As in (Carrasco and Forcada, 2002), we define $\delta : Q \times \Sigma \rightarrow Q$ a total mapping. In other words, if the automaton is not complete, i.e. if $\exists q \in Q \wedge \exists a \in \Sigma : \delta(q, a) \notin Q$, then an absorption state $\perp \notin F$ such that $\forall a \in \Sigma : \delta(\perp, a) = \perp$ must be added to Q. A complete acyclic automaton always has an absorption state. The extended mapping is defined as:

$$\begin{aligned} \delta^*(q, \epsilon) &= q \\ \delta^*(q, ax) &= \delta^*(\delta(q, a), x) \end{aligned} \tag{6.1}$$

The right language of a state q is defined as:

$$\overrightarrow{\mathcal{L}}(q) = \{x \in \Sigma^* : \delta^*(q, x) \in F\}$$

The language of the automaton $\mathcal{L}(M) = \overrightarrow{\mathcal{L}}(q_0)$. The right language can be defined recursively:

$$\overrightarrow{\mathcal{L}}(q) = \bigcup_{a \in \Sigma : \delta(q,a) \neq \perp} a \cdot \overrightarrow{\mathcal{L}}(\delta(q, a)) \cup \begin{cases} \{\epsilon\} & \text{if } q \in F \\ \emptyset & \text{otherwise} \end{cases}$$

The length of a string $w \in \Sigma^*$ is denoted $|w|$, and the i-th symbol in the string w as w_i.

6.6.3 Incremental Addition of Sorted Strings

Clarification of the Role of the Register

(Carrasco and Forcada, 2002) derive their algorithm from the union of an automaton $M = (q, \Sigma, \delta, q_0, F)$ with a single string automaton $M_w = (Q_w, \Sigma, \delta_w, q_{0w}, F_w)$. In a single string automaton, $Q_w = \mathrm{Pr}(w) \cup \{\perp_w\}$, where $\mathrm{Pr}(w)$ is the set of all prefixes of w, which also serve as names of states, $\perp_w$ is the absorption state, $F_w = \{w\}$, and $q_{0w} = \epsilon$.

States in the automaton $M' = M \cup M_w$ that is the result of the union can be divided into 4 groups:

- **Intact** states of the form $(q, \perp_w)$ with $q \in Q - \{\perp\}$ – states that are not affected by the union.
- **Cloned** states of the form (q, x) with $q \in Q - \{\perp\}$ and $x \in \mathrm{Pr}(w)$ such that $\delta^*(q_0, x) = q$. All other states in $(Q - \{\perp\} \times \mathrm{Pr}(x)$ can be safely discarded. The new initial state (q_0, ϵ) is a cloned state.

- **Queue** states of the form $(\perp, x)$, with $x \in \mathrm{Pr}(w)$.
- The new **absorption** state $\perp' = (\perp, \perp_w) \notin F$. It is present only if M has an absorption state.

In (Carrasco and Forcada, 2002), the algorithm proceeds by minimizing the queue states and cloned states, arriving at the minimal automaton. All states of M are put into a set called a *register* of states, which holds all unique states in the automaton. States unreachable from the new initial state are removed from the automaton and from the register. Then, starting from the states that are the most distant from the initial state, queue states and cloned states are compared against those in the register. If an equivalent state is found in the register, it replaces the state under investigation. If not, the state under investigation is added to the register.

Before we go further, we have to clarify the role of the register of states. It is explained in (Daciuk et al., 2000), but taken for granted in (Carrasco and Forcada, 2002). Incremental construction consists in two synchronized processes: one that adds new states, the other one that minimizes the automaton. In minimization, it is important to check whether two states are equivalent. The Myhill-Nerode theorem tells us that two states are equivalent when they have the same right languages. Computing right languages can take much time. However, what we need to check is whether two states have the same right language, and not what that language actually is. We can use the recursive definition of the right language. If the target states of all outgoing transitions are unique in the automaton, i.e. they are already in the register, then instead of comparing their right languages, we can compare their identity (i.e. e.g. their addresses). The assumption in the previous statement can be made true by enforcing a particular order in which states are compared against those in the register. When states are on a path representing a finite string, they should be processed from the end of the string towards the beginning.

The queue states should be processed in that order. If an equivalent state is found in the register, it replaces the current state. Otherwise, the current state is added to the register.

The register can be organized as a hash table. Finality of the state, the number of transitions, labels on transitions, and targets of transitions are treated together as a key – an argument to a hash function. The register does not store right languages. It stores pointers to states. If the right language of a state changes, the key of that state does not have to.

Necessary modifications

We divide the set of cloned states into two groups: **prefix** states – up to, but excluding the first state with more than one incoming transition, and the **proper cloned** states, which will simply be called the **cloned** states. Cloned states are modified copies of other states. They are new states; they were created by adding a new string. In (Carrasco and Forcada, 2002), the prefix states are also cloned. However, it is usually not necessary to clone them. They all change their right languages as the result of adding a new string, but only the last prefix state (the most distant from the initial state) is sure to change its transitions. Therefore, it should be removed from the register **before** adding a new string. Other prefix states should be removed from the register only if they change their key features. This can only happen if the next prefix state in the path is replaced by another state. In that case, the current prefix state is removed from the register, and reevaluated. If an equivalent state is found in the register, it replaces the current state, and the previous prefix state should be considered. Otherwise the state is put back into the register, and no further reevaluation is necessary.

If strings are added in an ordered way, the minimization process can be optimized in the same way as in the “sorted data algorithm”, the first algorithm described in (Daciuk et al., 2000). We introduce two changes to the string addition algorithm in (Carrasco and Forcada, 2002):

- prefix states are not cloned when not necessary,
- states are never minimized (i.e. compared against the register, and either put there or replaced by other states) more than once.

The first modification is described above. The second one requires more explanation. Let us consider an automaton where no minimization takes place after a new string has been added. That automaton has form of a trie. If a set of strings is lexicographically sorted, then the paths in the automaton recognizing two consecutive strings w and w' share some prefix states (at least the initial state, or the root of the trie). We call the longest initial part of two strings w and w' that is identical for both of them the **longest common prefix** of w and w' – $lcp(w, w')$. If w is a prefix of w', then all states in the path recognizing w are also in the path of w'. Otherwise, there will be states in the path recognizing w that are not shared with the path recognizing w'. Note that no subsequent words will have them in the common prefix path either, as the shared initial part of paths of w and subsequent words can only become shorter. Therefore, those states will never change their right language, so they can be minimized without any further need of reevaluation. As soon as we add w', we know which states in the path of w can be minimized. Instead of a try, we keep a minimal automaton except for the path of the last string added to the automaton.

If we start from scratch, and add strings in the manner just described, cloned states will never be created. Cloned states are created only when the common prefix of two words contains states with more than one incoming transitions. Additional transitions coming to states are created when the states are in the register, and they are found to be equivalent to some other states. But the states can be put into the register only when they are no longer in the common prefix path.

In case of a cyclic automaton, we do not start from scratch. There must be an initial (minimal) automaton that contains cycles. No new cycles are created by adding simple strings. As the automaton already contains some strings, and it can contain states with more than one incoming transition, cloned states can be created. However, no cloned states will be created in the common prefix path, because the path recognizing the previous string does not contain any states with more than one incoming transition.

The algorithm that makes special use of the fact that strings come in lexicographical order is different from the general algorithm in yet one aspect. After a string has been added, the automaton is not completely minimized. It is "almost" minimized. It is minimal except for the path representing the last string added.

The Algorithm

```
{1}  $R \leftarrow Q$;
{2}  **if** $(\text{fanin}(q_0) > 0)$ **then**
{3}     $q_0 \leftarrow \text{clone}(q_0)$;
{4}  **fi**;
{5}  $w' \leftarrow \epsilon$;
{6}  **while** $((w \leftarrow \text{nextword}) \neq \epsilon)$ **do**
{7}     $p \leftarrow \text{lcp}(w, w')$;
{8}     $M \leftarrow \text{minim\_path}(M, w', p)$;
{9}     $M \leftarrow \text{add\_suffix}(M, w, p)$;
{10}    $w' \leftarrow w$;
{11} **end**;
{12} $\text{minim\_path}(M, w', p)$;
{13} **if** $\exists r \in R : \text{equiv}(r, q_0) \rightarrow$
{14}    delete $q_0$; $q_0 \leftarrow r$;
{15} **fi**

{16} **func** $\text{lcp}(M, w, w')$;
{17}    $j \leftarrow \max(i : \forall_{k \leq j} w_k = w_{k'})$;
{18}    **return** $(\delta^*(q_0, w_1 \ldots w_j), j)$;
{19} **cnuf**

{20} **func** $\text{minim\_path}(M, w, p)$;
{21}    $q \leftarrow \delta^*(q_0, p)$;
{22}    $i \leftarrow |p|; j \leftarrow i$;
{23}    **while** $i \leq |w|$ **do**
{24}      $\text{path}[i - j] \leftarrow q$;
{25}      $q \leftarrow \delta(q, w_i)$;
{26}      $i \leftarrow i + 1$;
{27}    **end**;
{28}    $\text{path}[i - j] \leftarrow q$;
{29}    **while** $i > j$ **do**
{30}      **if** $\exists r \in R : \text{equiv}(r, q)$ **then**
{31}         $\delta(\text{path}[i - j - 1], w_{i-1}) \leftarrow r$;
{32}         delete $q$;
{33}      **else**
{34}         $R \leftarrow R \cup \{q\}$;
{35}      **fi**;
{36}      $i \leftarrow i - 1$;
{37}    **end**;
{38}    **return** $M$;
```

```
{39} cnuf;

{40} func add_suffix($M, w, p$);
{41}   $q \leftarrow \delta^*(q_0, p)$;
{42}   $i \leftarrow |w|$;
{43}   while $i \leq |w|$ and $\delta(q, w_i) \neq \perp$ and fanin($\delta(q, w_i)$) $\leq 1$ do
{44}     $q \leftarrow \delta(q, w_i)$; $R \leftarrow R - \{q\}$;
{45}     $i \leftarrow i + 1$;
{46}   end;
{47}   while $i \leq |w|$ and $\delta(q, w_i) \neq \perp$ do
{48}     $q \leftarrow$ clone($\delta(q, w_i)$);
{49}     $q \leftarrow \delta(q, w_i)$;
{50}     $i \leftarrow i + 1$;
{51}   end;
{52}   while $i < |w|$ do
{53}     $\delta(q, w_i) \leftarrow$ newstate;
{54}     $q \leftarrow \delta(q, w_i)$;
{55}     $i \leftarrow i + 1$;
{56}   end;
{57}   $F \leftarrow F \cup \{q\}$;
{58}   return M;
{59} cnuf
```

Function *funin*(q) returns the number of incoming transitions for a state q. If the initial state has more than one incoming transitions, it must be cloned (lines 2–4) to prevent prepending of unwanted prefixes to words to be added. Function *nextword* simply returns the next word in lexicographical order from the input. Function *lcp* (lines 16–19) returns the longest common prefix of two words. It is called with the last string added to the automaton, and the string to be added to the automaton as the arguments. For the first string, the previous string is empty. Function *minim_path* minimizes that part of the path recognizing the string previously added to the automaton that is not in the longest common prefix. This is done by going to the back of the path representing the string (lines 21–28) and checking the states one by one starting from the last state in the path (lines 29–37). The register is represented as variable R.

While function *minim_path* is not much different from an analogical function for the acyclic case, function *add_suffix* (lines 40–59) does introduce some new elements. It resembles more closely a similar function from the algorithm for unsorted data ((Daciuk et al., 2000)). The longest common prefix of the string to be added and the last string to be added to the automaton is not necessarily the same as the longest common prefix of the string to be added to the automaton and all strings already in the automaton. The latter can be longer, and the path recognizing it may contain states with more than one incoming transition. Those states have to be cloned (lines 47–51).

6.6.4 Analysis

The algorithm correctly adds new strings to the automaton, while maintaining its minimality. We assume that all states in the initial automaton are in the register, there are no pairs of states with the same right language, all states are reachable from the initial state, and there is a path from every state to one of the final states. The absorption state and transitions that lead to it are not explicitly represented.

If the initial state has any incoming transitions, it is cloned, and the clone becomes the new initial state. That operation does not change the language of the automaton – the right language of the new initial state is exactly the same as of the old one. The old initial state is still reachable, because it has incoming transitions from either the new initial state (the old initial state had a loop) or other states that reachable. The cloning creates a new state that is not in the register and that is equivalent to another state in the automaton. Lines 13–15 of the algorithm check whether after addition of new strings the new initial state is equivalent to some other state in the automaton. If it is the case, the new initial state is replaced with the equivalent state.

When strings are ordered in lexicographical order, the longest common prefix of two subsequent strings is never shorter than the longest common prefix of the first string of those two and any string after then second one. The right language of a state can be changed either by making the state final (in line 57), or by adding a transition to it (line 54), or by changing any state reachable from it. The path of the last string added to the automaton does not contain any states that have more than one incoming transition, and the initial state has no incoming transitions. If the initial state has any incoming transitions, it is cloned in lines 2–4, and if states with more than one transition are encountered during addition of a new string, they are cloned in lines 47–51. Strings added after a certain string cannot reach states not in their common prefix by following the path of that string. They cannot make the states final, and they cannot add new transitions to them. This means that their right language is not going to change. The only other way to reach them is via transitions created in line 31 of function *minim_path*. Those transitions are created only after a new string has been added – they are not followed by new strings.

States in the path of the last string added to the automaton are not in the register. They were not registered when they were created, or they were removed from the register in line 44. They are the only states in the automaton that are not in the register and that can have equivalent states elsewhere in the automaton. States in the path of the last string added to the automaton but not in the longest common prefix path (queue states), are minimized. The minimization is performed starting from the end of the string, so that all states reachable from a state under minimization are already in the register. Therefore, it is sufficient to compare only transitions and finality of two states to check if they are equivalent. When queue states of the last string have been minimized, queue states for the new string are created. After the new string has been added, its path contains states that are not in the register – the only such states in the automaton. We return to the start situation.

The algorithm hash the same asymptotic complexity as algorithms in (Carrasco and Forcada, 2002) and (Daciuk et al., 2000). However, it is faster than algorithms for unsorted data because it does not have to reprocess the states over and over again.

6.7 Incremental Construction with Continuation Classes

6.7.1 Introduction

Acyclic deterministic finite-state automata (ADFA) provide compact and fast representation for morphological dictionaries. Traditional methods for constructing ADFA require considerable amounts of memory. They are also slower then new, incremental algorithms. However, the incremental methods currently in use construct minimal ADFA from sets of strings. They do not use additional information contained in morphological descriptions. A recent study ((Daciuk, 2002)) showed that the semi-incremental construction algorithm by Dominique Revuz ((Revuz, 1991)) was the fastest for morphological dictionaries with long morphological annotations. The advantage of the algorithm in that case was a search for common suffixes. Information about common suffixes is already encoded in morphological descriptions. Therefore, we present an algorithm that constructs minimal ADFA using that information.

6.7.2 Formal Preliminaries

We define a deterministic finite-state automaton as $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite set of symbols called the alphabet, $q_0 \in Q$ is the start (or initial) state, and $F \subseteq Q$ is a set of final (accepting) states. δ is a partial mapping $Q \times \Sigma \mapsto Q$. If for a certain $q \in Q$ and $a \in \Sigma$, $\delta(q, a) \notin Q$, we write $\delta(q, a) = \perp$. The extended mapping is defined as:

$$\begin{aligned} \delta^*(q, \epsilon) &= q \\ \delta^*(q, ax) &= \delta^*(\delta(q, a), x) \end{aligned}$$

The right language of a state q is defined as:

$$\overrightarrow{\mathcal{L}}(q) = \{x \in \Sigma^* : \delta^*(q, x) \in F\}$$

The language of the automaton $\mathcal{L}(M) = \overrightarrow{\mathcal{L}}(q_0)$. The right language can be defined recursively:

$$\overrightarrow{\mathcal{L}}(q) = \bigcup_{a \in \Sigma : \delta(q,a) \neq \perp} a \cdot \overrightarrow{\mathcal{L}}(\delta(q, a)) \cup \begin{cases} \{\epsilon\} & \text{if } q \in F \\ \emptyset & \text{otherwise} \end{cases}$$

Equality of right languages is an equivalence relation over states, and it divides them into classes of abstractions. An automaton is minimal iff all its states are useful (they are reachable from the initial state, and there is a path from each state to any of the final states), and every abstraction class has only one representative.

An acyclic deterministic finite-state automaton (ADFA) is a deterministic finite-state automaton that does not contain loops. In other words, in an ADFA, $\forall_{q \in Q} \forall_{w \in \Sigma^*} \delta(q, w) \neq q$.

Equality of all outgoing transitions (equal number of transitions with the same labels and the same targets) and the same finality of states (i.e. the states are either both final or both non-final) is also an equivalence relation. That equivalence implies equality of right languages, but not vice versa. It means that two states may have the same right languages, but not the same set of transitions (they must be both final or both non-final). However, in an ADFA, if no pair of states has the same finality and set of transitions, no pair of states has the same right language. If we check equivalence of states in order of increasing length of the longest string in their right languages, then checking equality of transitions and of finality is sufficient. It can be proven by induction.

6.7.3 Continuation Classes

In morphological descriptions, the lexicon of morphemes can be divided into sub-lexica called *continuation classes* (See (Beesley and Karttunen, 2002), (Schulze et al., 1994)). An entry in a continuation class specifies a list of morphemes and the next continuation class. Morphemes from the current and the next continuation class are then concatenated (and are subject to further alternations using spelling rules).

We can see continuation classes as fixed points in a finite-state automaton. Morphemes are added between those points, forming chains of states and transitions labeled with subsequent letters from the morphemes. Unfortunately, due to phenomena like null morphemes and prefixes, continuation classes cannot be represented by single states, but such simplified view is still useful in explaining the concept. Common sets of suffixes are now represented by the same next continuation classes. There is no need to check for those suffixes in each string. To facilitate compilation, the traditional format of continuation classes has been slightly altered. We put the next continuation class in front of a list of morphemes, so that we know both the start and the end point of a chain of transitions to be added. An example is shown on Figure 6.30.

A predefined class *NULL* has no morphemes and no further continuation classes. All other classes should be defined before they are specified as the next continuation classes for some other classes. By defining a continuation class, we define a part of an ADFA.

6.7.4 Addition of New Morphemes

The algorithm for adding new morphemes is similar to the incremental construction algorithm for unsorted data presented in (Daciuk et al., 2000) and generalized in (Carrasco and Forcada, 2002). However, instead of adding strings between the initial state and a final state, we add it between two given states. Therefore, the last step in the algorithm – adding the last transition – is different. Also, we may encounter null morphemes.

Function *add_morpheme* (Figure 6.31) accepts three parameters: *start* – the initial state of the class, *target* – the state representing the next continuation class, and *m* – the morpheme. The function modifies the right language of *start*, and possibly some other states reachable from *start*. Conceptually, we can divide the morpheme into three parts: a prefix – a part (not containing the last symbol in the morpheme) such that there is a path in the automaton from the initial state with transitions labeled with subsequent symbols of the prefix, a suffix – the remaining part excluding the last symbol, and the last symbol. Any of those parts may be empty. The last symbol is empty only if the morpheme is empty. If the morpheme is an empty string, then the right language of the *start* must also contain the right language of *target*. This is done by creating a state with the right language being the union of the right languages of *start* and *target*.

If the morpheme is not empty, the longest common prefix of the morpheme with any string already in the automaton is followed (lines 5–15). That part of the morpheme is already in the automaton; we only need to add the second part (the suffix). However, if the path of the prefix contains a state with more than one incoming transition (a reentrant state), then by adding the suffix, we would also add the suffix to some other word. Therefore, all prefix states from the first reentrant state must be cloned (lines 12–15). Cloning means creating a copy of a state with exactly the same transitions (the same labels, the same targets) and the same finality.

The suffix is added in lines 16–19 by creating a chain of states linked with transitions with labels spelling out the suffix. A set of states with unique right languages in maintained in variable R. That set is called the *register* ((Daciuk et al., 2000)). Because adding the suffix changes the right language of the last non-reentrant state, we remove it from the register, unless it is the initial state of the continuation class (*start*). All prefix states up to the first reentrant state change their right language, but it is not necessary to remove them from the register. We will return to that issue later.

If the suffix is empty, then there may be a transition from the last prefix state labeled with the same symbol as the last symbol of the morpheme. In that case (lines 22-26), a new state is created with the right language being the union of the right language of the target of the transition and the right language of *target*. Function *fanin* returns the number of incoming transitions for a state. If by redirecting some transitions, a state has no more incoming transitions, it is deleted automatically. Variable *off* is set to zero to indicate that the newly created state is not yet registered. If the suffix is not empty, or if there is no transition from the last prefix state labeled with the same symbol as the last symbol of the morpheme, a transition is created from that state to *target*. Variable *off* is set to 1 to indicate, that *target* is already in the register.

In lines 30–42 add states to the register. This is done from the end of the morpheme towards the beginning. The last state in the path is either *target* – a state already in the register, or a state being a union of two other states. In the latter case, all transitions leaving that state go to states already in the register. It means that we can use equality of sets of transitions and of finality as the equivalence relation. All states added by *add_morpheme* are either put to the register (line 39), or replaced by some other states already in the register (line 37). Also, the last state in the prefix preceeding of any reentrant state is either put into the register or replaced with an equivalent state. If that state is replaced, then the state preceeding it is removed from the register (before the replacement), and then minimized again, and the process is repeated until no further changes are made or until the initial state is reached. The initial state is never put into the register in function *add_morpheme*. At the end of the function, all states except for the initial state of the class are in the register, and the initial state is returned.

Function *merge_states* (Figure 6.32) creates a state with the right language begin a union of the right languages of the two states – arguments of the function. A copy of the first state is created, and then transitions of the second state are added to it. If both states have transitions labeled with the same symbol, but leading to different targets, function *merge_states* is called recursively to create states accepting unions of the right languages of both targets (line 7). If a state created in such a way has a unique right language, it is put into the register (line 15). Otherwise, it is replaced with an equivalent state (lines 13 and 17) and deleted (line 12).

6.7.5 Problems and Solutions to Them.

Suppose we have a class C1 with morphemes “a” and “b” and the next continuation class *NULL*, and we are in the process of creating another class C2 with morphemes “a”, “b”, and “c”, and the next continuation class *NULL*. If we registered the initial state of class C2 after the addition of “a” and “b”, then we would have to replace it with the initial state of class C1. By adding morpheme “c” to it, we would also modify the language of C1. The problem is solved by putting the initial state of a class into the register only after the whole class has been defined (after the right brace ending the class definition has been parsed).

There is a related problem. Suppose we have the class C1 with the right language “a” and “b”, and we define a new class with the language “aa”, “ab”, and “ac”. After having added “aa” and “ab” to the class, we notice that we have created a state that is equivalent to the initial state of class C1, so we replace the newly created state with the old one. When we add “ac”, we add “c” to class C1. This problem is avoided by adding an additional incoming transition for the initial state of every finished class. The newly created state is still replaced with the initial state of class C1, but when we add “ac”, we have to clone it, because it has more than one incoming transitions (one from the initial state of the class we define, another one artificial). Actually, we do not need to create artificial transitions, we only increase the counter of incoming transitions.

At the end of processing, only one continuation class and its initial state is important. Some continuation classes may have initial states with only one, artificial incoming transition. They should be deleted to make the automaton minimal. A directive (“Forget:”) is provided to decrease incoming transition counters of known classes and deleting the corresponding states if they have no incoming transitions.

6.8 Future Work

The incremental construction algorithm from continuation classes (Section 6.7) can be extended to cyclic automata. The method is currently under development. Another possible extension is the inclusion of spelling rules into the construction process. In the current solution, the spelling rules have to be composed with lexicon at run time.

```
(1)  func equiv(p, q, k)
(2)      if $p = q$ then eq := true
(3)      elsif $class[p] \neq class[q]$ then eq :=false
(4)      elsif $k = 0$ then eq := true; rl := 0
(5)      elsif $\{p, q\} \notin S$ then eq := true; rl := index({p, q}, S); skip
(6)      elsif $\{p, q\} \in G$ then eq := false
(7)      elsif $\{p, q\} \in P$ then eq := true; rl := index({p, q}, P)
(8)      else
(9)        if $level = 0 \vee in(p) > 1 \vee in(q) > 1$ then
(10)         $S := S \cup \{\{p, q\}\}$; level := level +1; pushed := true;
(11)       fi
(12)       $rl' := |Q|$; eq :=true;
(13)       for $a : a \in \Gamma_p \cap \Gamma_q$ do
(14)           eq := eq$\wedge$ equiv($\delta(p, a), \delta(q, a), k - 1$);
(15)           $rl' := min(rl', rl)$;
(16)       rof;
(17)       $rl := rl'$;
(18)       if pushed then
(19)           $S := S \setminus \{\{p, q\}\}$; level := level −1
(20)       fi
(21)       if eq then
(22)           if $rl > level$ then
(23)             merge({p, q})
(24)           else
(25)             $P[rl] := P[rl] \cup \{\{p, q\}\}$
(26)           fi
(27)       else
(28)           $G := G \cup \{\{p, q\}\}$
(29)       fi
(30)       if rl = level then rl := |Q|; fi
(31)       if eq then
(32)         if rl = |Q| then
(33)           $\forall_{\{r,s\} \in P[level]}$ merge({r, s})
(34)         else
(35)           $P[rl] := P[rl] \cup P[level]$
(36)         fi
(37)       else
(38)         $\forall_{\{r,s\} \in P[level]}$ $G := G \cup \{\{r, s\}\}$;
(40)       fi
(41)       $P[level] := \emptyset$;
(42)     fi;
(43)     return eq
(44) cnuf
```

Figure 6.12: Comparison of a pair of states in the incremental minimization algorithm – final version

	$\|Q\|$	$\|\delta\|$	cl	min	S_o	S_n	e_{ns}	e_s	e_{nm}
church	14	21	5	6	2	4	30	17	35
g1	9	21	2	2	3	3	20	20	22
g10	13	18	2	2	3	5	17	17	20
g11	17	34	3	3	6	8	36	33	72
g13	337	673	4	5	32	32	726	692	$63 \cdot 10^6$
g14	137	273	4	5	16	16	297	283	9 852
g15	35	69	4	5	9	9	77	71	283
g4	103	206	2	34	17	20	867	867	3 911
g5	22	42	6	8	3	3	94	46	95
g5p	20	30	10	19	4	4	170	47	172
g6	49	98	2	45	4	4	1 297	1 297	1 318
g7	39	78	2	7	2	2	129	129	132
g8	288	1624	9	16	4	4	2 444	1 636	3 247
g9	232	463	4	5	14	14	482	478	3 407
g9a	16	32	3	3	5	7	33	30	55
griml	17	48	2	2	2	2	55	55	53
java	112	424	19	26	11	18	950	402	9 825
java16	3 186	12 077	19	46	94	94	13 507	12 267	too big
java19	1 971	24 633	24	26	129	129	26 295	24 630	too big
ovis4n	133	613	6	12	14	14	673	637	23 768
ovis5n	44	169	6	12	8	8	194	171	725
ovis6n	17	81	4	5	3	3	77	71	171
ovis7n	13	49	4	4	3	3	44	40	84
ovis9p	2 478	7 434	2	7	27	27	8 239	8 239	27 455
rene2	844	1380	47	193	10	24	40 969	16 434	31 726
ygrim	9	28	4	4	1	2	28	22	24

Table 6.3: Results of experiments on real examples. $|Q|$ – number of states, $|\delta|$ – number of transitions, cl – number of classes, min – number of states in minimal automaton, S_o – max depth of recursion of *equiv()*, S_n – S_o excluding the pairs that cannot start cycles, e_{ns} – calls to *equiv()* without pre-sorting with memoization, e_s – calls to *equiv()* with pre-sorting and memoization, e_{nm} – call to *equiv()* without pre-sorting and without full memoization.

	2D	tree	no sort	no mem		2D	tree	no sort	no mem
g13	4	4	3	27 298	ovis4n	2	1	1	12
java16	166	73	262	too big	ovis9p	89	109	81	78
java19	166	73	262	too big	rene2	22	36	28	28

Table 6.4: Execution times for real-life automata.

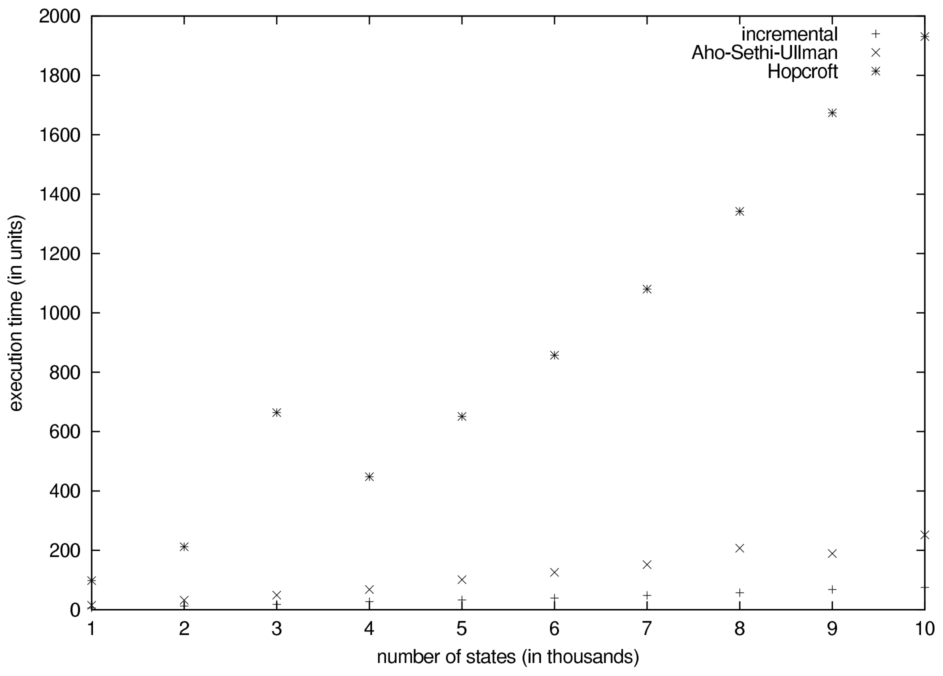


Figure 6.13: Execution times for large automata with a large number of classes.

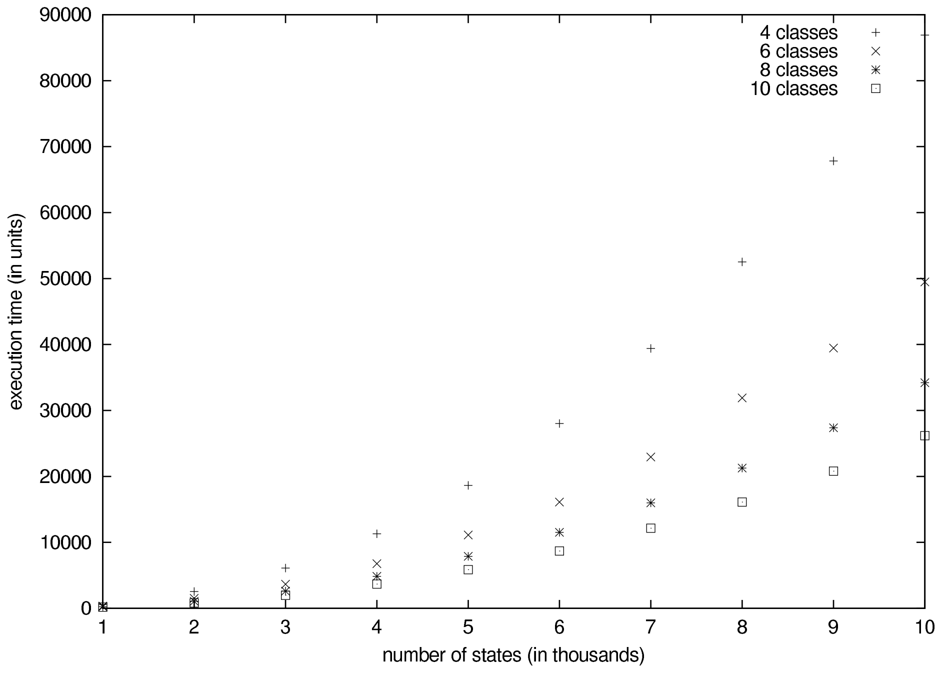


Figure 6.14: Execution times for large minimal automata with limited number of classes.

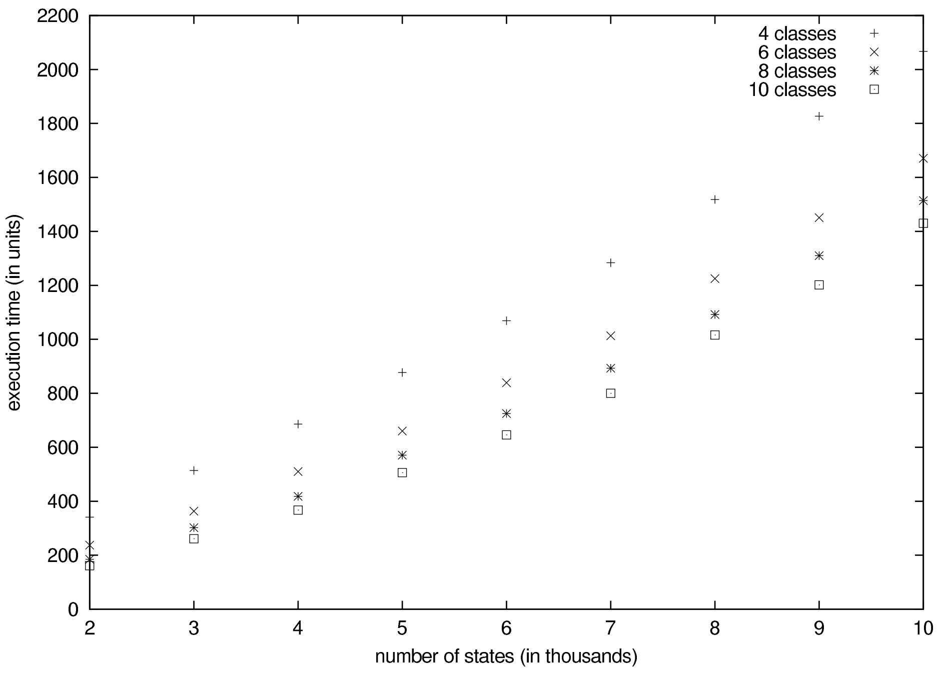


Figure 6.15: Execution times for inflated automata of 1000 states with various number of classes.

		strings			automaton	
		words	characters	av. len.	states	trans.
German	words	716 273	10 221 410	14.27	45 959	97 239
	morph.	3 977 448	364 681 813	91.69	107 198	435 650
French	words	210 284	2 254 846	10.72	16 665	43 507
	morph.	235 566	17 111 863	72.64	32 078	66 986
Polish	words	74 434	856 176	11.50	5 289	12 963
	morph.	92 568	5 174 631	55.90	84 382	106 850
`/usr/dict/words`		45 407	409 093	9.01	23109	47346

Table 6.5: Characteristics of data used in experiments

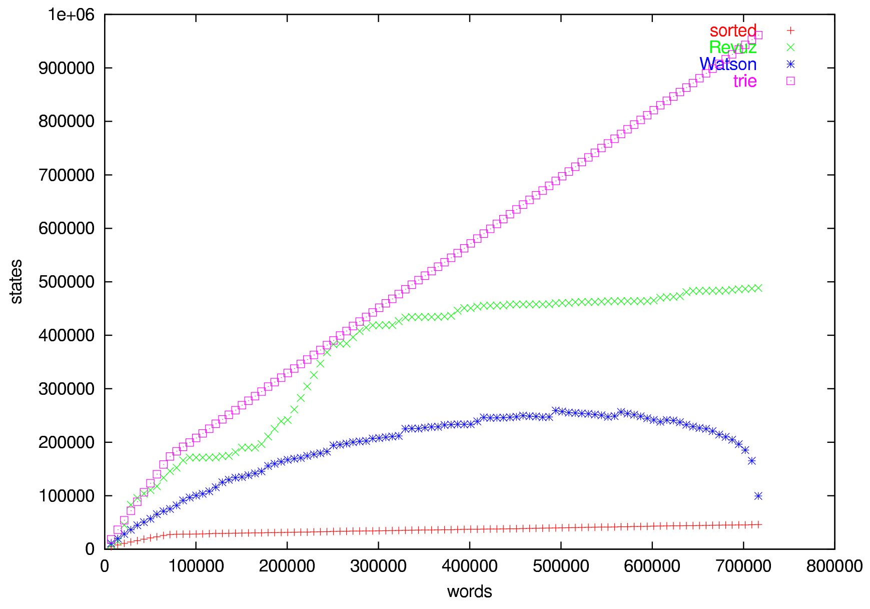


Figure 6.16: Memory requirements for German words.

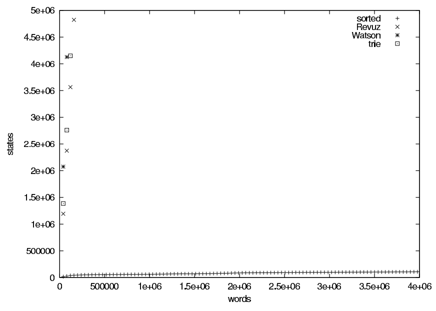


Figure 6.17: Memory requirements for German morphology

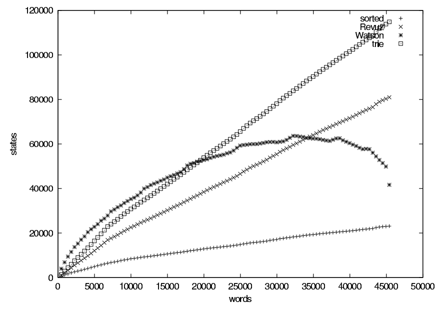


Figure 6.18: Memory requirements for English words from `/usr/dict/words`.

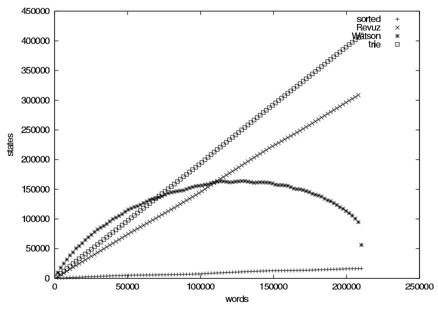


Figure 6.19: Memory requirements for French words.

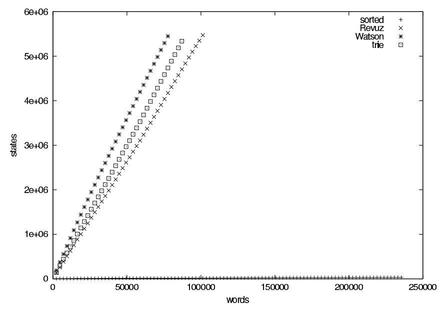


Figure 6.20: Memory requirements for French morphology

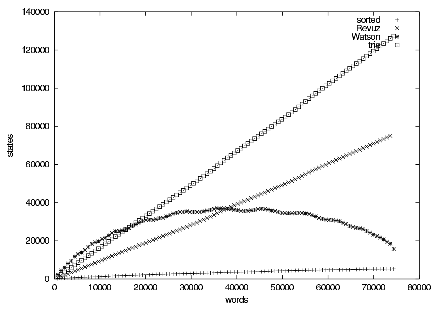


Figure 6.21: Memory requirements for Polish words.

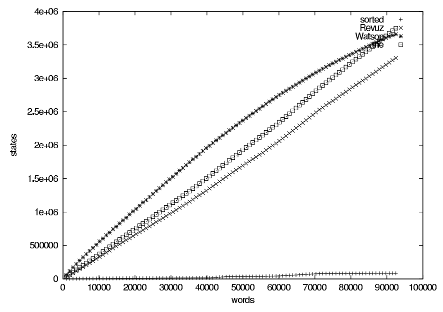


Figure 6.22: Memory requirements for Polish morphology.

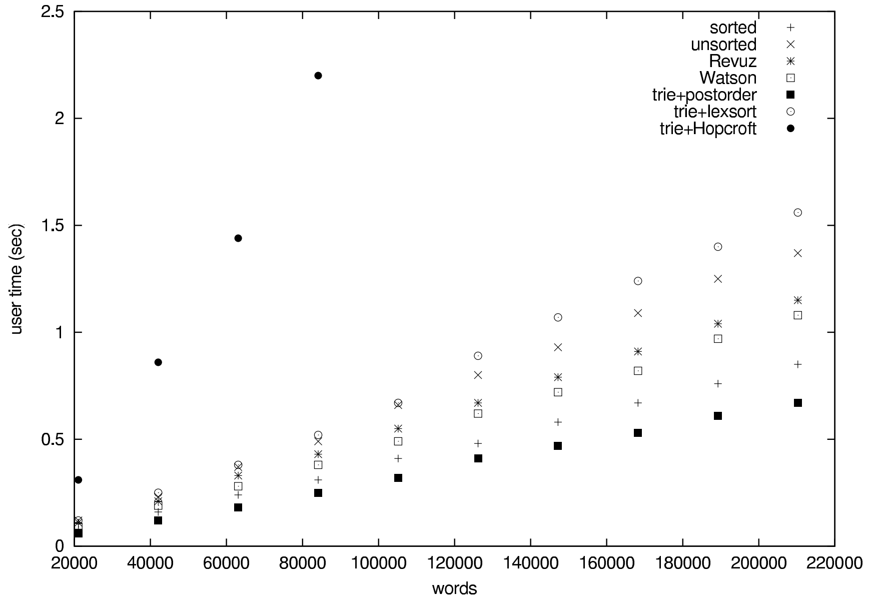


Figure 6.23: Execution time for Polish words

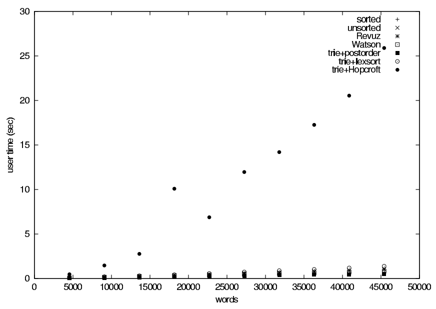


Figure 6.24: Execution time for English words.

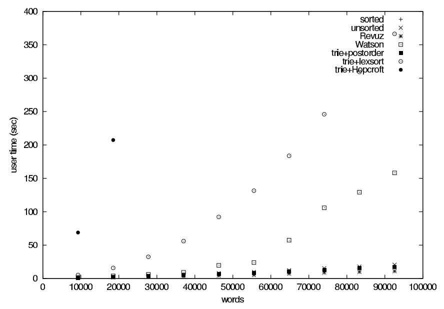


Figure 6.25: Execution time for Polish morphology.

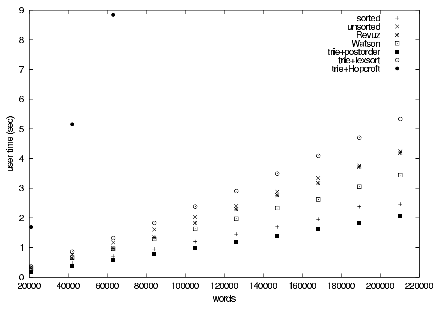


Figure 6.26: Execution time for French words.

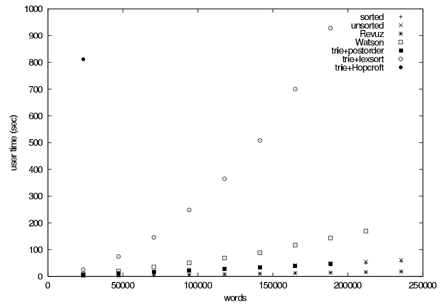


Figure 6.27: Execution time for French morphology

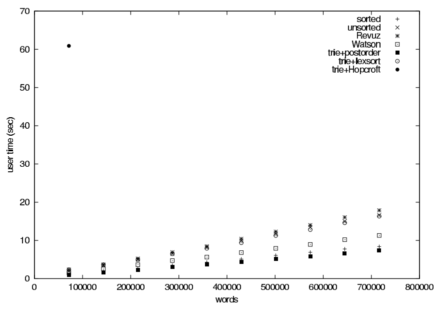


Figure 6.28: Execution time for German words.

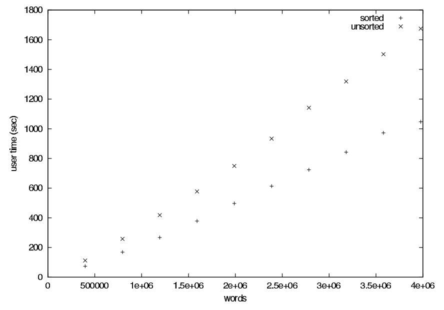


Figure 6.29: Execution time for German morphology.

```
Cl1: {
    Cl2     ``morph1'' ``morph2'';
    Cl3     ``morph3'' ``morph4'';
    NULL    ``morph5'';
}
```

Figure 6.30: Example of a continuation class in our format.

```
func add_morpheme(*start*, *target*, *m*)
  if $m$ = “” $\rightarrow$
    return merge_states(*start*, *target*);
  else
    $p \leftarrow$ *start*; $i \leftarrow 1$;
    while $i < |m|$ and $\delta(p, m_i) \neq \perp$ and $\mathrm{fanin}(\delta(p, m_i)) \leq 1)$ $\rightarrow$
      $p \leftarrow \delta(p, w_i)$; *path*[i] $\leftarrow p$;
      $i \leftarrow i + 1$;
    elihw
    if $p \neq$ *start* $\rightarrow R \leftarrow R - \{p\}$; fi
    *uf* $\leftarrow i$;
    while $i < |m|$ and $\delta(p, m_i) \neq \perp$ $\rightarrow$
      $\delta(p, m_i) \leftarrow \mathrm{clone}(\delta(p, m_i 8)$; $p \leftarrow \delta(p, m_i)$; *path*[i] $\leftarrow p$;
      $i \leftarrow i + 1$;
    elihw
    while $i < |m|$ $\rightarrow$
      $\delta(p, m_i) \leftarrow new$; $p \leftarrow \delta(p, m_i)$; *path*[i] $\leftarrow p$;
      $i \leftarrow i + 1$;
    elihw
    *off* $\leftarrow 1$;
    if $\delta(p, m_i) \neq \perp$ $\rightarrow$
      $q = \mathrm{merge\_states}(\delta(p, m_i), target)$;
      if *fanin*$(\delta(p, m_i)) \leq 1)$ $\rightarrow$
        $R \leftarrow R - \{\delta(p, m_i)\}$;
      fi
      $\delta(p, m_i) \leftarrow q$; *off* $\leftarrow 0$;
    else
      $\delta(p, m_i) \leftarrow$ *target*;
    fi
    $p \leftarrow \delta(p, m_i)$; *path*[i] $\leftarrow p$;
    $i \leftarrow |m| - off + 1$;
    while $i + off > 1$ $\rightarrow$
      if $\exists_{r \in R}\, r \equiv path[i]$ $\rightarrow$
        if $i <\ uf$ and $i + off > 2$ $\rightarrow$
          $R \leftarrow R - \{path[i]\}$;
        fi
        $\delta(path[i-1], m_{i-1}) \leftarrow r$;
      else
        $R \leftarrow R \cup \{\mathrm{path}[i]\}$;
        if $i <= uf$ $\rightarrow$ return *start*;
      fi
    elihw
  fi
  return *start*;
cnuf
```

Figure 6.31: Adding a string to a continuation class.

```
func merge_states($s_1, s_2$)
    $s \leftarrow$ clone($s_1$);
    foreach $a \in \Sigma : \delta(s_2, a) \neq \bot \rightarrow$
        if $\delta(s_1, a) = \bot \rightarrow$
            $\delta(s, a) \leftarrow \delta(s_2, a)$;
        elsif $\delta(s_1, a) \neq \delta(s_2, a) \rightarrow$
            $q \leftarrow$ merge_states($\delta(s_1, a), \delta(s_2, a)$);
            if fanin($\delta(s_2, a)$) $\leq 1 \rightarrow$
                $R \leftarrow R - \{\delta(s_2, a)\}$;
            fi
            if $\exists_{r \in R}\, r \equiv q \rightarrow$
                delete $q$;
                $q \leftarrow r$;
            else
                $R \leftarrow R \cup \{q\}$;
            fi
            $\delta(s, a) \leftarrow q$;
        fi
    hcaerof
    return $s$;
cnuf
```

Figure 6.32: Function *merge_states* creates a new state with the right language being the union of the right languages of its arguments

Chapter 7

Empirical Aspects of Finite-State Language Processing

7.1 Introduction

7.1.1 The larger project

This research is part of the PIONIER project “Algorithms for Linguistic Processing”, which focuses on problems of ambiguity and processing efficiency by investigating *grammar approximation* and *grammar specialization* techniques. As computational grammars grow larger and larger, taking more of linguistic theory into account and thus getting better at recognizing an increasing number of syntactical constructions, they tend to become slower as well. At the same time, and of course related to this, ambiguity increases: as the system is able to recognize more, it will assign more analyses to one and the same utterance, making it harder to find the ’correct’ one.

Ambiguity can be reduced by means of grammar specialization techniques. This amounts to the use of a large corpus of linguistic data in narrowing down the grammar to represent more closely the language as it is actually used. To put it another way, the grammar (or some other part of the analyzing system) is changed so that its output is closer to language *performance* of real speakers than to an exhaustive display of language *competence*. In addition to fighting ambiguity directly by making use of grammar specialization, the inefficiency caused by ambiguity as well as by the increasing size of the grammar in general can be addressed by means of grammar approximation techniques. The research “Empirical Aspects of Finite-State Language Processing” is about using finite-state techniques to this end. Before more is explained about this research, a short description is given of what grammar approximation is.

7.1.2 Grammar approximation

In *grammar approximation* a complex grammar is reduced to a much more simple grammar that is an approximation of the original grammar. Note that this is not the same as *grammatical inference*, which constitutes deducing a grammar from a sample (Dupont, 1997).

As an example, in (Nederhof, 2000) we find a discussion on different approaches to the case of *regular approximation*, where finite-state automata (which define regular languages) are constructed that approximate context-free languages. In figure 7.1 the general idea of approximation is shown. As we will see later, grammatical inference can be part of the process of grammar approximation.

$$\boxed{\text{grammar}} \rightarrow \text{calculations} \rightarrow \boxed{\text{simpler grammar}}$$

Figure 7.1: grammar approximation

7.1.3 Finite-state language processing

Why do we propose to use finite-state techniques? As described in (van Noord, 2000a), there are three arguments that support this decision:

1. Humans have a small (finite) amount of memory available for linguistic processing
2. Humans have trouble dealing with certain constructions, such as center-embedding, that are impossible to describe using finite-state techniques
3. Humans process language very efficiently, namely in linear time

This suggests that the human language processing machinery could be essentially regarded as a finite-state device: they share the finiteness of the available resources, the restrictions in dealing with certain linguistic phenomena, and the linear processing time. In the rest of this subsection I summarize the research questions and discussion as found in (van Noord, 2000a).

Methods of finite-state approximation

Many methods of finite-state approximation of grammars are based on the use of a parsing algorithm that is in some way restricted, so that it recognizes finite-state languages only. For instance, in (Johnson, 1998) a finite-state approximation technique is proposed based on a left-corner parser; the parser is compiled into a finite-state automaton as the size of the stack is restricted to a finite maximum depth. Approaches such as this one have not yet been applied in practice.

In contrast, another method know as *chunking* has been successfully used. This method divides the grammar up into separate levels of rules. Each level performs an analysis of its input, the result of which is then used as input to the next level, resulting at a certain moment in phrases, which are again used in later levels in more extensive analyses. Since rules do not apply recursively at a level, and the number of levels is finite, this can be viewed as a finite-state approximation. However, the rules are constructed by hand: we would prefer an approach in which the approximation is derived from the grammar in a more automatic way.

Evaluation of finite-state approximation

Once a method for approximation has been selected the resulting approximation can be evaluated, both qualitative and experimental. On a qualitative side, it can be seen that some methods always produce a *superset* of the language of the input grammar, others always produce a *subset*, and for some methods such a clear distinction can not be made.

Besides this, methods could be compared to each other by looking at specific characteristics when approximating certain sub-classes of grammars, such as regular, left-linear or right-linear grammars.

Another interesting area for questions is the relation to human language performance. How does the approximation handle constructions that are difficult for humans to deal with? For instance, what kind of center embedding is allowed in the approximation? And, at the same time, does the approximation properly recognize those cases that human beings *don't* have problems with?

Experimental evaluation can be performed to see if there is any such loss of accuracy, as well as to find out how the different methods compare with respect to the size of the resulting automaton and other computationally interesting aspects.

Approximation of constraint-based grammars

Most approximation techniques assume the original grammar is a context-free grammar. Certain techniques can be generalized to be used with feature-based grammars, and sometimes the idea is that the feature-based grammar is first transformed into a context-free grammar, which can then be approximated. In all of these cases, the approximation results in a grammar representing a *superset* of the original grammar: during approximation, distinct complex categories in the input grammar are mapped to the same category, making it possible to recognize sentences which were not recognized before. An important question in this research would be if it is possible to create an approximation that captures a *subset* of the language recognized by the original constraint-based grammar.

Approximation and interpretation

The finite-state automaton which is the result of approximating a more complex grammar usually does not share with this grammar the ability to assign structural descriptions to the utterances in the languages it recognizes. There have been work-arounds to this problem, but in all cases the resulting system has to return to the original grammar at a certain point in the process, in order to make language understanding possible. We would prefer a solution in which the approximation is implemented as a finite-state *transducer* that produces, upon recognizing a certain input sequence, a corresponding structural description. In this setup there would be no need to refer back to the original grammar in the semantic interpretation component.

7.2 Grammar Approximation through Inference

In section 7.1.2 it was described, though only in a few words, how approximation is done in general. In contrast to this approach, we have used a method of grammar approximation in which the procedure denoted as 'calculations' in figure 7.1 is replaced by a procedure of *grammatical inference*. A large corpus is first annotated by the grammar we want to approximate, and afterwards we use grammatical inference to derive a simpler grammar from the annotated corpus. This simple grammar is then an approximation of the grammar used to annotate the corpus, derived in an indirect way, as is also depicted in figure 7.2.

$$\boxed{\text{grammar}} \rightarrow \text{annotating} \rightarrow \boxed{\text{corpus}} \rightarrow \text{inference} \rightarrow \boxed{\text{simpler grammar}}$$

Figure 7.2: grammar approximation using inference

7.2.1 Learning a language from a sample

In the approach to approximation described here, we need a sample of the language that is defined by the grammar we want to approximate, for use in the process of inference. When the learning of a language from a sample of that language is concerned, many papers note the theorem by Gold (Gold, 1996) which states that regular languages (as well as context-free and context-sensitive languages) are impossible to learn from a sample containing only positive examples, that is examples of utterances that are part of the language. Instead, a *complete sample* would be needed, namely a sample containing both positive and negative examples with respect to the target language. In this case, the language can be *identified in the limit*. After a finite number of changes in the hypothesis regarding the language, the target language will have been identified.

However it seems that children have little or no access to negative examples of the language they are acquiring (Firoiu, Oates, and Cohen, 1998b; Marcus, 1993), and yet are successful in doing this. Chomsky (in 1975) suggested an explanation in which language is innate, instead of something that has to be learned. Another explanation would be the idea that it is somehow possible to learn a language from a positive sample, if this sample is based on a probability distribution. This is shown to be true: the stochastic information enclosed in such a sample makes up for the lack of negative data (Angluin, 1988).

7.2.2 Methods of inference

As described in (Dupont, 1997), the grammar that is the product of inference can be of different forms, such as regular, context-free or context-sensitive. In the case of a regular grammar, this can also be interpreted as creating a finite-state automaton; in the light of our project this would be what we are interested in for reasons mentioned in the introduction. There are a number of approaches possible when it comes to learning a finite-state automaton from a stochastic sample, including a few different algorithms. The following paragraphs present a number of these possibilities.

Learning a Hidden Markov Model

We can construct a probabilistic model of the language by *counting*. The model has to be restricted in some way to allow us to do this, since otherwise the number of different things to be counted would be very large, or infinite. Thus we restrict the model to work with a limited history: for every element in a sample sequence in the language we are considering, the model takes as the context of this element not the entire sequence, but only a number of *preceding* elements. When we consider n preceding elements, we can construct what is called an n-th order Hidden Markov Model. In this model the actual *states* are hidden, but we can observe certain corresponding output elements which are therefore called *observations*. In the case of a language model the observations are normally the words in a sentence, while the hidden states represent the corresponding syntactical categories or parts of speech. The idea is that the system traverses through the different states, where probabilities can be assigned to the event of the system changing from one state into another state, and in each state the system produces an observation, again with a certain probability.

The model is trained by counting the occurrences of sequences of states of length $n + 1$, as well as the number of times a certain observation is associated with a certain state, and using these counts to compute probabilities for these different events. The result can be seen as a (simple form of) probabilistic finite-state automaton. The model can be used to compute the likeliness of a certain sequence of observations, or to find the sequence of hidden states that best explains a given observed sequence.

Instead we could also use the simpler Markov Model in which the states are not hidden; in that case we only compute probabilities for traversing from a certain state to the next state. However in many situations the states are not directly accessible, as is also the case for applications related to language. These systems are better modeled using the Hidden Markov Model.

Using the ALERGIA algorithm

The previous paragraph described the HMM approach to inference, which delivers a restricted model of the language. Another method, while still resulting in a finite-state automaton, gives a better model by lifting the restriction on the size of the history used. The ALERGIA algorithm (Carrasco and Oncina, 1999) first constructs the *prefix tree automaton* (PTA) based on a sample of the target language. This is a stochastic automaton representing all prefixes found in the sample, where each transition is given a probability according to the number of times it is traversed during construction of the PTA. Through merging of states in the PTA, the algorithm generates an automaton that captures not only all the strings found in the sample, but hopefully also strings from the language that were *not* part of the sample: the algorithm finds the *canonical generator*. This is done in linear time with respect to the size of the sample set. States in the PTA are merged if they are *equivalent*: they are prefixes that lead to the same suffixes (or *subtrees* in the PTA). Since the PTA is a stochastic automaton, a number of statistical tests can be used in deciding if two states are indeed equivalent, which makes ALERGIA a class of algorithms.

Using the MDI algorithm

The Minimal Divergence Inference (MDI) algorithm (Thollard, Dupont, and de la Higuera, 2000) is in many ways similar to the ALERGIA class of algorithms. It uses a different learning criterion (or *merging* criterion) in which the Kullback-Leibler measure of divergence is used. It is noted in (Thollard, Dupont, and de la Higuera, 2000) that the ALERGIA algorithm does not provide the means to bound the divergence between the distribution as defined by the DFA produced by the algorithm and the training set distribution, as the merging operation operates locally on pairs of states. The learning criterion of the MDI algorithm does not have this problem. During construction of the PDFA, the algorithm constantly trades off between the divergence from the training sample (which should be as small as possible) and the difference in size between the current and the new automaton (which should be as large as possible, resulting in a smaller automaton). Empirical results show the MDI approach performs far better than the ALERGIA method.

Other methods

Using Bayesian analysis. The technique of first constructing a model that fits the sample data and afterwards merging parts of the model, resulting in a more general model, is also applied in (Stolcke and Omohundro, 1993) to Hidden Markov Models, which, as is stated in (Stolcke and Omohundro, 1993), "...can be viewed as a stochastic generalization of finite-state automata, where both the transitions between states and the generation of output symbols are governed by probability distributions". The initial HMM differs from the initial DFA used by (Carrasco and Oncina, 1999) in that it is not a prefix tree automaton, but an automaton in which every string from the sample set is stored: the start state has a many outgoing transitions as there are strings in the sample.

The process of state merging is in this case guided by *Bayesian* analysis. According to Bayes' rule the posterior model probability $P(M|x)$ is the product of the prior probability of the model, $P(M)$, and the likelihood of the data, $P(x|M)$. When states in the model are merged, the likelihood of the data is bound to decrease since the model moves away from the perfect representation of the sample data. However, the prior probability of the model increases, since one of the implications of the way in which $P(M)$ is computed is that there is a bias towards smaller models. As long as the loss in likelihood is outweighed by the prior probability of the model, merging continues. It is reported that the method is good at finding the generating model, using only a small number of examples.

Using a neural network. In (Firoiu, Oates, and Cohen, 1998a) we see how neural networks are used to learn finite automata. So called *Elman recurrent neural networks* are trained on positive examples created by an artificial, small grammar. Next, the DFA is extracted from the network to see what kind of automaton the network has learned. The network, when trained on the prediction task (predicting the word following the current input) tends to encode an approximation of the minimum automaton that accepts only the sentences in the training set. When trained on a small language, the training set DFA is indeed recovered. When the network is trained on a larger language, both correct and incorrect generalizations are introduced. In (Carrasco, Forcada, and Santamaria, 1996) a similar method is described. They show that the difference between the probability distribution as predicted by the extracted automaton and the true distribution is smaller than the difference between the true distribution and the distribution as predicted by the neural network itself. They also show that both these differences are smaller than the difference between the sample's distribution and the true distribution, which indicates that the inference method using neural networks has the ability to generalize.

7.3 Application of Grammar Approximation

While there were references in earlier sections to methods of grammar approximation that approximate and replace a grammar as a whole, we have implemented the technique in such a way that the resulting model forms an addition to the existing grammar: the original grammar is approximated and the result of this can be used to deal with a certain part of the total process of analyzing a sentence, hopefully in an efficient way, since that is the main reason for using an approximation.

In this fashion we have implemented a HMM part-of-speech tag filter for use in the Alpino system, using the approach to approximation as described in section 7.2. The approximation of the grammar is used to ascribe probabilities to sequences of POS tags, filtering out certain tags if they are considered unlikely. This reduces lexical ambiguity, which results in shorter parsing times and a slight increase in parsing accuracy. In the rest of this section the POS filter will be described, starting with a description of the Alpino system and the way this system works.

7.3.1 Lexical ambiguity

Full parsing of unrestricted texts on the basis of a wide-coverage computational HPSG grammar remains a challenge. In our recent experience in the development of the Alpino system, discussed in section 2, we found that even in the presence of various clever chart parsing and ambiguity packing techniques, lexical ambiguity in particular has an important effect on parsing efficiency.

In some cases, a category assigned to a word is obviously wrong for the sentence the word occurs in. For instance, in a lexicalist grammar the two occurrences of `called` in (1) will be associated with two distinct lexical categories. The entry associated with (1-a) will reflect the requirement that the verb combines syntactically with the particle '`up`'. Clearly, this lexical category is irrelevant for the analysis of sentence (1-b), since no such particle occurs in the sentence.

(1) a. I called the man up
    b. I called the man

An effective technique to reduce the number of lexical categories for a given input consists of the application of hand-written rules which check such simple co-occurrence requirements. Such techniques have been used in similar systems, e.g. in the English Lingo HPSG system (Kiefer et al., 1999).

We extend this filtering component using a part-of-speech (POS) filter. We consider the lexical categories assigned by the lexical analysis component as POS-tags, and use standard POS-tagging techniques in order to remove very unlikely tags.

In earlier studies, somewhat disappointing results were reported for using taggers in parsing (Wauschkuhn, 1995; Charniak et al., 1996; Voutilainen, 1998). Our approach is different from most previous attempts in a number of ways. These differences are summarized as follows.

Firstly, the training corpus used by the tagger is *not* created by a human annotator, but rather, the training corpus is labeled by the parser itself. Annotated data for languages other than English is difficult to obtain. Therefore, this is an important advantage of the approach. Typically, machine learning techniques employed in POS-tagging will perform better if more annotated data is available. In our approach, more training data can be constructed by simply running the parser on more (raw) text. In this sense, the technique is *unsupervised.*

Secondly, the HPSG for Dutch that is implemented in Alpino is heavily lexicalist. This implies that (especially) verbs are associated with many alternative lexical categories. Therefore, reducing the number of categories has an important effect on parsing efficiency.

Thirdly, the tagger is not forced to disambiguate all words in the input (of course, this has been proposed earlier, e.g. in (Carroll and Briscoe, 1996)). In typical cases the tagger only removes about half of the tags assigned by the dictionary. As we show below, the resulting system can be up to about twenty times as fast, while parsing accuracy does not drop. For somewhat less drastic efficiency gains, we observed an increase in parsing accuracy. Parsing accuracy drops considerably, however, if we only use the best tag for each word (this differs from the conclusion in (Charniak et al., 1996)).

Fourthly, whereas in earlier work evaluation was described e.g. in terms of the number of sentences which received a parse, and/or the number of parse-trees for a given sentence, we have evaluated the system in terms of lexical dependency relations, similar to the proposal in (Carroll, Briscoe, and Sanfilippo, 1998). This evaluation measure presupposes the availability of a treebank, but is expected to reflect much better the accuracy of the system.

We implemented a standard bigram HMM tagger in which the emission probabilities are directly estimated from a labeled training corpus. A standard POS-tagger attempts to find the best sequence of tags for the given input sentence, or perhaps the n-best sequences of tags for small n. As we discuss later, this is not appropriate for our purposes. Rather, we use an idea from chapter 5.7 of (Jelinek, 1998) by computing the *a posteriori* probability for each tag. We use a threshold in order to cut away for every position in the input string the most unlikely tags. The same idea is described in (Charniak et al., 1996).

7.3.2 Using a POS-tagger as a filter

Training and test data for the POS-tagger

Recall that the tagged training corpus is not annotated by humans, but rather by the parser. Thus, we have run the parser on a large training set, and collected the sequences of lexical categories that were used by the best parse (according to the parser).

Of course, the training set thus produced contains errors, but since the POS-tagger is used to select the best tags out of the tags suggested by the parser, it makes sense to train it on the parser's output as well.

In our experiments discussed below, we used as our corpus the first six months of 1997 of the Dutch newspaper 'de Volkskrant', except that we kept apart some 5783 sentences (91857 words) which are used for the stand-alone tests described below. The remaining 517492 sentences were fed to the parser, with a time-out of 60 CPU-seconds per sentence, and with the robustness component disabled (in such a way that only those sentences were used for which the parser found a complete parse). 324575 sentences (4879085 words) were parsed successfully; the corresponding tag sequences are used to train our bigram model.

We implemented a standard bigram HMM tagger, described e.g. in chapter 10.2 of (Manning and Schütze, 1999): an HMM in which each state corresponds to a tag, and in which emission probabilities are directly estimated from a labeled training corpus. For each sentence, the filter is given as input the set of tags found by the lexical analysis component of Alpino. The task of the filter consists of the removal of all unlikely tags. We have experimented with a few techniques to determine which tags are unlikely.

Using the most likely sequence

The optimal sequence of tags for a given sentence is defined as:

$$\hat{t}_{1,n} = \arg\max_{t_{1,n}} P(t_{1,n}|w_{1,n}) = \prod_{i=1}^{n} P(w_i|t_i)P(t_i|t_{i-1})$$

The Viterbi algorithm is used to compute this most probable tag sequence. In a first experiment, we simply assumed that a tag is removed if it is not part of the most probable tag sequence. This results in most of the tags being discarded, leading to a tagging accuracy of 84.3% (as can be seen on the first line of Table 7.1).

One might wonder why the results of this tagger are so poor, whereas in the literature tagging is supposed to obtain at least an accuracy level of 95%. This is caused by the size of the tag set (more than 25K different tags). If the tagger would simply use the most frequent tag for each given word (in isolation of its context), then we would obtain a tagger accuracy of about 50%. This should be contrasted with typical taggers in which this base-line is reported to be around 90%.

Using the n-best sequences

Since using only one sequence leads to low accuracy results, the set of accepted tags is extended to include the tags that make up the n-best sequences. For dif-

n	tags/word	accuracy (%)
1	1.00	84.3
5	1.15	88.9
15	1.36	91.3
25	1.50	92.2
50	1.69	93.3
100	1.90	94.2
200	2.11	94.9
300	2.23	95.3
∞	3.32	100

Table 7.1: Filter results using the n-best paths approach

τ	tags/word	accuracy (%)
0	1.00	86.5
1	1.10	89.8
2	1.23	92.6
3	1.37	94.6
4	1.51	96.1
5	1.66	97.0
6	1.81	97.7
7	1.97	98.3
8	2.11	98.7
∞	3.32	100

Table 7.2: Filter results using forward-backward method

ferent values of n Table 7.1 shows the results. A word is assumed to be tagged correctly if the correct tag is not filtered by the tagger. Increasing accuracy by considering more sequences leads to more ambiguity at the same time, and makes the Viterbi algorithm increasingly slower. Since the number of possible tag sequences increases exponentially with sentence length, we have also experimented with dynamically chosen values of n; these experiments were not very successful, and for longer sentences (i.e. larger values of n) the Viterbi algorithm itself becomes too slow.

Using forward and backward probabilities

In attempting to increase accuracy and decrease ambiguity, the idea of selecting tags based on the most likely sequences must be reconsidered. In an alternative approach, probabilities are computed for individual tags, allowing us to directly compare tags that are assigned to the same word. Thus, for each word in the sentence, we are interested in the probabilities assigned to each tag by the HMM. This is similar to the idea described in chapter 5.7 of (Jelinek, 1998) in the context of speech recognition. The same technique is described in (Charniak et al., 1996).

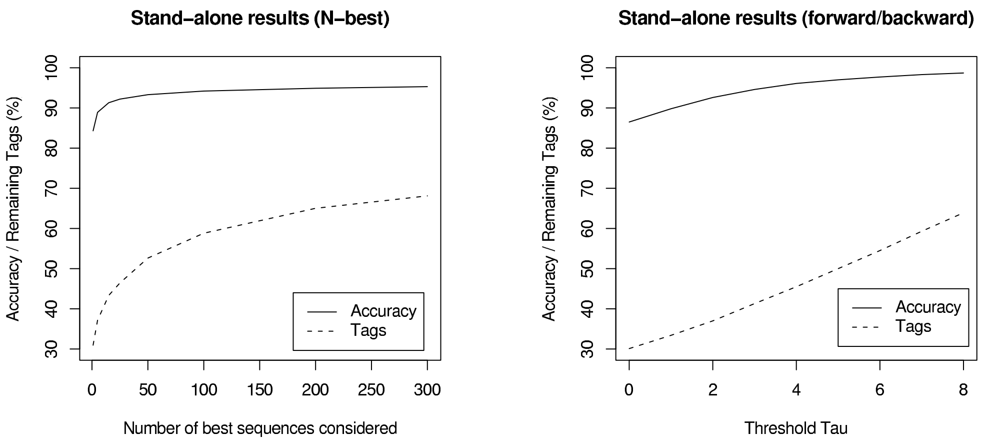


Figure 7.3: Stand-alone results for N-best and forward/backward techniques

The probability that t is the correct tag at position i is given by:

$$P(t_i = t) = \alpha_i(t)\beta_i(t)$$

where α and β are the forward and backward probabilities as defined in the forward-backward algorithm for HMM-training; $\alpha_i(t)$ is the total (summed) probability of all paths through the model that end at tag t at position i; $\beta_i(t)$ is the total probability of all paths starting at tag t in position i, to the end.

Once we have calculated $P(t_i = t)$ for all potential tags, we compare these values and remove tags which are very unlikely. Let $s(t,i) = -\log(P(t_i = t))$. A tag t on position i is removed, if there exists another tag t', such that $s(t,i) > s(t',i) + \tau$. Here, τ is a constant threshold value. We report on experiments with various values of τ.

The results using forward and backward probabilities to compute the likeliness of individual tags are given as Table 7.2, showing the average number of remaining tags per word and tagger accuracy percentages for various threshold levels τ. The method is a significant improvement over the n-best sequences approach, as can be seen in the comparison in Figure 7.3.

7.3.3 Incorporating the POS-tagger in Alpino

Evaluation procedure

The treebank used in the experiments with the Alpino parser is the CDBL (newspaper) part of the Eindhoven corpus (Uit den Boogaart, 1975), which we are currently annotating with dependency structures, according to the guidelines specified in (Moortgat, Schuurman, and van der Wouden, 2002). These dependency structures are similar to those used in the German Negra corpus (Skut, Krenn, and Uszkoreit, 1997). CDBL220 refers to the first 220 sentences of the CDBL part, which are all annotated. The average sentence length is 20 words.

τ	tags/word	CPU (msec)	Accuracy (%)
NONE	3.53	76620	74.79
0	1.05	1421	68.05
1	1.16	2341	71.88
2	1.32	4077	74.89
3	1.46	6620	75.63
4	1.62	10894	75.15
5	1.77	15702	74.94

Table 7.3: Incorporating the POS-filter in Alpino: results on cdbl220 corpus. (CPU times are averages per sentence)

Evaluation of coverage and accuracy of a computational grammar usually compares tree structures (such as recall and precision of (labeled) brackets or bracketing inconsistencies (crossing brackets) between test item and parser output). As is well-known, such metrics have a number of drawbacks. Therefore, (Carroll, Briscoe, and Sanfilippo, 1998) propose to annotate sentences with triples of the form ⟨*head-word, dependency relation, dependent head-word*⟩. For instance, for the example in (1) we obtain:

⟨zou su mercedes⟩	⟨zou vc hebben⟩
⟨hebben su mercedes⟩	⟨hebben vc aangekondigd⟩
⟨aangekondigd su mercedes⟩	⟨aangekondigd obj1 model⟩
⟨aangekondigd mod gisteren⟩	
⟨model mod nieuwe⟩	⟨model det haar⟩

Dependency relations between head-words can be extracted easily from the dependency structures in our treebank, as well as from the dependency structures constructed by the parser. It is thus straightforward to compute precision, recall, and f-score on the set of dependency triples. In the experiments described below, the accuracy of the Alpino parser is expressed in terms of this f-score.

Experiment with POS-filter in Alpino

In Table 7.3 we summarize the experiments in which the POS-filter is applied as a preprocessing component in Alpino. The POS-filter used the forward and backward probabilities to filter out unlikely tags, as described in the previous section. We experimented with various values of τ. In the table, the first row describes the reference system in which no POS-filter is applied. In that case, all tags are used by the parser, and parsing is very slow. The accuracy is 74.79%. As can be concluded from the table, a threshold value of $\tau = 2$ already performs (slightly) better than the reference system, with a sharp decrease in parsing times.

Ignoring subcategorization information

Inspection of the errors made by the filter indicated problems with subcategorization information. Since the bigram model uses only a limited history, this information is not always used properly. By removing the extra information from the tags, the algorithm can make a better decision in these cases. So, both in training and during the application of the filter, we map each verbal tag to its *class*, by removing the subcategorization specification. For instance, the verb *hebben* in (1) is assigned the following tags:

```
verb(inf,aux_psp_hebben) verb(inf,pred_transitive)
verb(inf,transitive)     verb(pl,aux_psp_hebben)
verb(pl,pred_transitive) verb(pl,transitive)
```

This set is mapped to two classes, `verb(inf)` and `verb(pl)`. If the tagger finds that a class is too unlikely, then all tags that were mapped to that class are removed. Similarly, if a class survives the filter, then all tags which were mapped to that class will be available during parsing. The transformation clearly makes tagging much easier by making the number of different tags much smaller (about 1400 tags remain). This class-based approach typically removes less tags than the previous approach. In Table 7.4 we show the results. In figure 7.4 we compare the system without a POS-filter with two systems including the POS-filter, either with or without subcategorization information. As can be seen from these figures, the accuracy of Alpino is improved if subcategorization from verbs is ignored. If $\tau = 2$ then Alpino is almost ten times faster than the reference system (without POS-tagger), and the corresponding accuracy is higher too: 76.40% vs. 74.79%.

7.4 Building a less restricted DFA

7.4.1 Another method for inference

The previous section showed how we implemented an approach to approximation using inference. The method of inference used was that of a Hidden Markov Model. It appeared to be the case that the HMM method had trouble dealing with subcategorization information when the model was applied in the part-of-speech filtering task. Since this model only uses bigram information, resulting in a simple form of finite-state automaton, this is not surprising; such long distance dependencies are out of the model's reach. A solution would be to build a less restricted form of

τ	tags/word	CPU (msec)	Accuracy (%)
NONE	3.53	76620	74.79
0	1.46	3125	73.33
1	1.58	5221	75.30
2	1.70	7846	76.40
3	1.81	11054	76.29
4	1.94	26415	76.09
5	2.09	34611	75.68

Table 7.4: POS-filter in Alpino, if subcategorization information is ignored; cdbl220 corpus. (CPU times are averages per sentence)

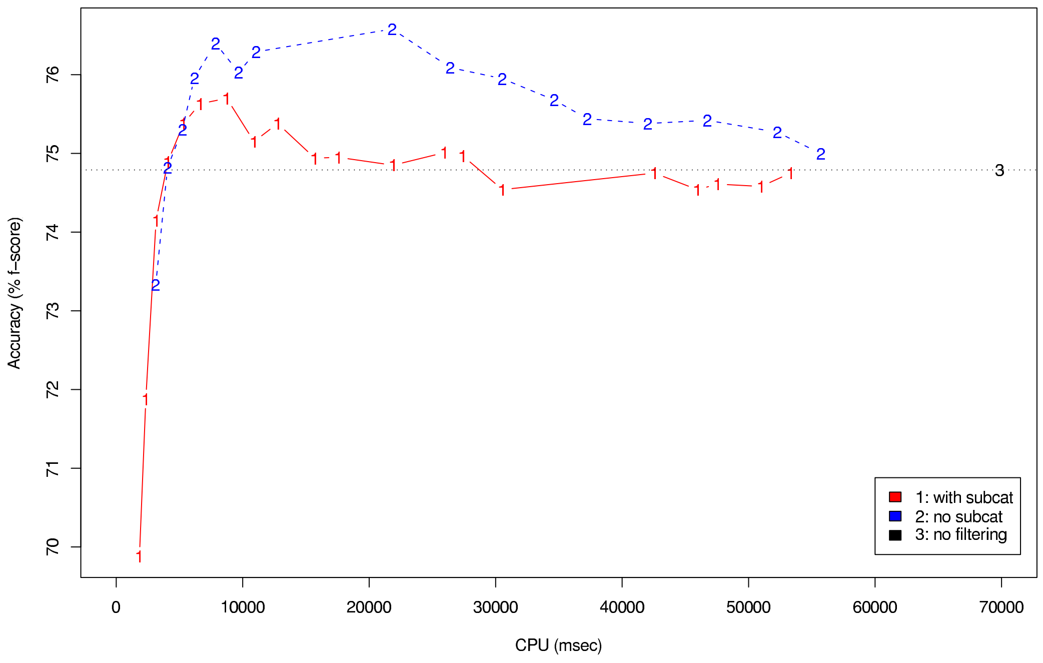


Figure 7.4: POS-filtering for Alpino

finite-state automaton, and this is possible using algorithms such as the ALERGIA and MDI algorithms which were already mentioned in section 7.2.2.

The idea behind both of these algorithms is to construct an automaton that models exactly a given positive sample of the language (this automaton is the probabilistic prefix tree automaton), and then to generalize this automaton in order to model a larger part of the language. Generalizing finite-state machines can be done by merging states. ALERGIA constructs the probabilistic prefix tree automaton from the sample. Then it compares pairs of nodes; the nodes are visited in the order as dictated by the automaton: the lexicographic order. Two nodes are merged (combined into one node) if they generate the same stochastic language.

7.4.2 MDI

The comparison of nodes in ALERGIA is a local event, meaning that the divergence of the more general automaton from the original automaton is not globally controlled. In the MDI algorithm the general idea is the same; states in the prefix tree are considered for merging in the same order. However, MDI uses a global criterion for equivalence of two states: the automaton resulting from a merge operation is compared to the automaton before merging, in terms of Kullback-Leibler divergence and reduction in size. Merging makes the model more general, and decreases its size. On the other hand, merging can lead to overgeneralization and a large divergence between the two successive models. MDI compromises between size and divergence; merging of two states takes place only if the divergence does not get the upper hand over the reduction in size. This is done by means of the following equation, in which the *Alpha* parameter regulates the amount of generalization that is allowed:

$$\frac{Divergence(A1, A2)}{|A1| - |A2|} < Alpha$$

Larger values for Alpha means allowing for more generalization (larger divergence). Smaller values mean divergence has to be smaller (in relation to the reduction in the number of states) in order for the new automaton ($A2$) to be accepted as a correct generalization from the previous automaton ($A1$).

7.4.3 Classification of subcategorization frames

We want to build a model that does not share the HMM's difficulties in working with long distance dependencies, or more specifically, with subcategorization frames. The question to be answered is whether a DFA constructed on the basis of a stochastic sample of a language using the MDI algorithm can be used to differentiate between subcategorization frames. (We choose to focus on the use of the MDI algorithm as this is based on the ALERGIA algorithm, and experimental results show that MDI outperforms ALERGIA.)

The sentences in our stochastic sample will consist of simplified POS-tags. This means that, for example, all information that is supplied with a verb on top of the fact that it is a verb, is removed. However, in each sentence the subcategorization information of one verb tag is not removed; the sentence acts as a training example for that particular subcategorization frame. (Note that the subcategorization information itself is simplified in the first round of the experiment; later on the experiment is repeated with somewhat more elaborate information in terms of expected prepositions and particles.)

A DFA is constructed using a fragment of the training data. Training sets of different sizes can be used, as well as different settings for the Alpha parameter regulating the amount of generalization.

Next, we take a single test sentence out of a set of unseen sentences. Just as with the training sentences, this sentence contains only one subcategorization frame; it may contain additional verbs, but these are reduced to a simple 'verb' tag, as mentioned above. We want to check if the DFA has properly learned the difference between subcategorization frames, given their respective environments. Thus the DFA should be able to recognize the single subcategorization frame that is in our test sentence as a probable frame in the context of the other tags in the sentence. In order to test this we construct copies of the test sentence in which the subcategorization frame is replaced by another frame. For this, all available subcategorization frames are used (resulting in 95 sentences, including the original). These sentences are used as input for the DFA, one at a time, and the probabilities assigned to each of the sentences in turn are compared. If the sentence containing the original frame receives the highest probability, the classification is correct, otherwise it is false (or undecided in the case of a draw). The classification accuracy is the percentage of correct classifications on 1000 test sentences. The results that

training size	vocabulary size	accuracy%
20,000	146.3	29.0
40,000	150.3	30.1
100,000	152	30.4
200,000	152.3	30.8

Table 7.5: Classification accuracies for bigram model

will be shown below are the mean accuracies of three runs of the above experiment, in which the training data was randomly selected each time the experiment was repeated.

Data

The data set consists of 324,574 sentences (with an average length of 15 words) from the *Volkskrant* newspaper corpus. Not all of these are used; for training we used 20,000 and 40,000 sentences (to see if the amount of training data has an effect on performance; in further experiments using less data gave poorer results while using more than 40,000 sentences did not improve upon the results). For testing we use 1000 sentences. Both training and testing data are randomly selected from the larger data set. The original data contained multi-word unit tags of the form 1-tag, 2-tag and so on; these were reduced to just the first tag in such a sequence, and with the leading number removed.

Baseline

An alternative to constructing a DFA based on a stochastic sample is to use an n-gram model of the same data. Both unigram and bigram models were applied to the experiment described above, both without smoothing. Table 7.5 gives the results for the bigram model. Using a unigram model leads to an average classification accuracy of 19.9%. A baseline accuracy could be assigned to an approach in which we always select the most frequent frame as being the correct one, and this would be equivalent to the use of a unigram model.

Results

Table 7.6 shows the results in terms of classification accuracy when using the MDI algorithm. Different amounts of training data were used, as well as different values for the Alpha parameter that controls the amount of generalization. The other values are all averages over three runs of the experiment. An exception to this is the number of transitions in the PPTA, which was determined by looking at the PPTAs of only one run.

The final column shows the classification accuracy, and it can be seen that using MDI results in lower accuracies than those attained by using the unigram model (except for one case, but the positive difference is insignificant). Using larger values for Alpha means allowing for more generalization, thus leading to the construction

training size	vocab. size	Alpha	PPTA transitions	DFA transitions	DFA states	acc.%
20,000	146.3	0.0002	222K	2298	152	15.5
20,000	146.3	0.0003	222K	776	45	20.0
20,000	146.3	0.0004	222K	305	14	19.4
20,000	146.3	0.0005	222K	265	8	19.4
40,000	150.3	0.0002	433K	353	17	19.4
40,000	150.3	0.0003	433K	207	6	19.9
40,000	150.3	0.0004	433K	196	5	19.9
40,000	150.3	0.0005	433K	156	4	19.9
Baseline: unigram model						19.9

Table 7.6: Classification accuracy for MDI-constructed DFA using different amounts of training data and different values for Alpha

of smaller automata. As the DFA gets smaller it resembles more and more the unigram model, leading to the same accuracy in classification (19.9%).

Using more information

Often part of the information in a subcategorization frame tag is about prepositions and particles being expected. One would think that keeping this kind of information available helps in classification, making the task easier when a certain subcategorization frame requires the presence of a certain preposition, and this preposition is (or is not) present in the sentence. However the resulting automaton is even smaller than the previously discussed result, leading to the same unigram-like behavior. Probably the positive effect of the availability of this extra information is overshadowed by the fact that the tag set has now increased in size from 152 to 1790 tags, of which 1529 are verb tags (compared to 95 when the extra information is hidden).

Conclusion

The MDI algorithm was used to create a DFA from a stochastic sample of the language. The ability of the DFA to recognize the correct subcategorization frame from amongst a number of alternatives, given the tags in the rest of the sentence, was tested. The baseline result would be the use of a unigram model of the same training data, corresponding to always selecting the most frequent frame as the best one, and this results in an accuracy close to 20%.

Using the MDI algorithm to construct a DFA using the same training data and using this DFA in the experiment also resulted in an accuracy close to 20%, meaning this approach did only just as well as the baseline.

Since the unigram model does not use any information besides the prior probability of the candidate frames in selecting the most appropriate frame in a given context, the above results seem to suggest that the DFA constructed by the MDI algorithm is equally unable to use the contextual information provided by the rest of the sentence. This could mean that the algorithm overgeneralizes the automaton, of which the small sizes of the resulting automata are also an indication.

7.5 Future work

As described in section 1, the method known as chunking has already been successfully used in finite-state approximation. Returning to the idea of using Hidden Markov Models to approximate a grammar, we could expand on this by having the model work with phrases instead of the smaller POS tags. The HMMs could be used in a chunker implemented as a tagger. (This could in turn be used as a filter; knowing which combinations of phrases are plausible can lead to decisions about which part-of-speech tags are likely or unlikely to be correct.)

Chapter 8

Finite State Transducers with Predicates and Identities

The research reported in this chapter is conducted in close cooperation with Dale Gerdemann (University of Tübingen).

8.1 Introduction

Finite automata are widely used in natural language processing. We present an extension to finite automata, in which atomic symbols are replaced by arbitrary predicates over symbols. Although the extension is fairly trivial for finite state acceptors, the introduction of predicates is more interesting for transducers. Below, we show how various operations on such extended acceptors and transducers can be defined and implemented. But first the extension is motivated as follows.

8.1.1 Predicates

In natural language processing, it is often more natural to think of symbols in terms of predicates or classes. The linguistic principle of *Community* dictates that similar segments behave similarly. Predicates are a means to express this similarity. In computational phonology it is thus more natural to talk about *vowels* and *consonants* rather than enumerate each of the phonemes in these classes. Phonological generalizations typically refer to predicates such as *fricative*, *nasal*, *voiced* and very seldomly to individual phonemes directly. Therefore, in finite state computational phonology, some have proposed finite state automata in which transitions are associated with sets of symbols (Walther, 1999; Bird and Ellison, 1994; Eisner, 1997; Walther, 2000).

As a further piece of motivation for the introduction of predicates, consider the *unknown symbol* regular expression operator, typically written `?`, as it is available in some regular expression compilers (Karttunen et al., 1996; van Noord and Gerdemann, 2000). An obvious implementation will expand the `?` operator into a set of transitions for each of the symbols in the alphabet Σ. In our proposal, the `?` operator will be expanded into a single transition with an associated predicate which is true for all symbols; this has the advantage that Σ need not be explicitly defined.

As a consequence, there is no need to assume that the alphabet is finite. Such considerations become important for applications with large alphabets, such as the Unicode alphabet. Even larger alphabets may surface in natural language processing applications in which the symbols are words. Typical electronic dictionaries have at least 200K words and even this large size alphabet is not enough to handle unrestricted texts. Realistically, robust syntactic parsing requires an infinite alphabet.[1]

Below, we define predicate augmented finite state automata more precisely; for now it suffices to assume that such automata are similar to classical finite state automata, except that we have predicates instead of symbols.

8.1.2 Notation

The predicates used in this paper are predicates on Σ. So, each predicate π is a total function such that for each $\sigma \in \Sigma$, $\pi(\Sigma)$ is either *true* or *false*. If π is the *characteristic function* of the set $S \subseteq \Sigma$, i.e., $S = \{\sigma \in \Sigma | \pi(\sigma)\}$, then in transition diagrams we often write S instead of π. As usual, if S is a set, then the complement of S is written $\overline{S}$. Moreover, if S is of the form $\{\mathtt{c}\}$, i.e., a singleton set, then we abbreviate this predicate simply as `c`. As a special case, Σ is written as `?`. In transducers, a transition is associated both with an input predicate π_d as well as with an output predicate π_r; such a pair of predicates is written as $\pi_d : \pi_r$.

Below, we will often refer to states in automata using p, q, and r. For examples of symbols we use characters from the beginning of the alphabet in typewriter font such as `a, b, c`; for sequences of symbols we use characters w, x, y, z. Typically, we use σ as a variable that takes a symbol as its value. Examples of predicates are written in small caps, using characters from the beginning of the alphabet, like A, B, C. A variable that takes a predicate as its value is written π. A sequence of predicates is often written using Greek symbols ϕ, ψ. Finally, note that the empty sequence is written ϵ, for either the empty sequence of symbols or the empty sequence of predicates.

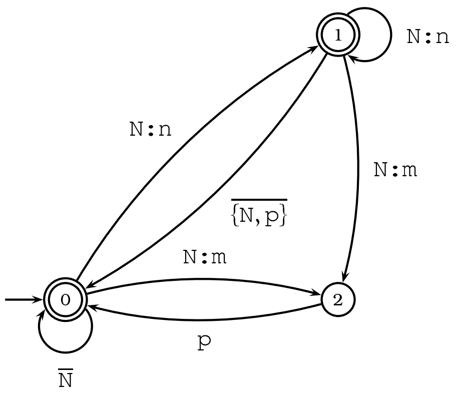
8.1.3 Identities

Consider the following phonological rule (from (Karttunen, 1991)) in which an underlying nasal segment `N` is mapped either to an `m` (if followed by a `p`) or an `n`:

$$\mathtt{N} \rightarrow \mathtt{m}/\_\ \mathtt{p}; \quad \text{elsewhere } \mathtt{n}$$

A transducer implementing this phonological rule can be illustrated as follows:

[1]If infinite alphabets are allowed, then certain non-regular languages such as $\{0, 1, \ldots\}^*$ can be recognized. A similar generalization of regular languages is used by (Perrin, 1990).



This transducer contains a single start state 0, and two final states, 0 and 1. First consider the cyclic transition on state 0, labeled by the predicate $\overline{\texttt{N}}$, i.e., the predicate which is true of all symbols except the symbol `N`. As long as the transducer does not read this special nasal segment `N`, it remains in state 0 and simply copies its input. Upon reading an `N`, the transducer non-deterministically moves to state 1 or state 2, writing out an `n` or `m` respectively. In the first case, the next input symbol cannot be a `p`; in the second case the next input symbol must be a `p`.

Note that the transition from state 2 to state 0 simply contains a `p`. The idea here is that if the input and output symbol must be identical, only a single predicate is written for that transition. The same abbreviation is used for the transition from 1 to 0, as well as over the looping transition from 0 to 0 with predicate $\overline{\texttt{N}}$. The intention here of course is that every incoming segment which is not equal to `N` should be mapped to itself in the output. However, note that this is quite different from the pair of predicates $\overline{\texttt{N}} : \overline{\texttt{N}}$. The latter would map an incoming symbol to an arbitrary output symbol, as long as both symbols are unequal to `N`.

The example illustrates an important point: if predicates are introduced in transducers, then for typical examples we must also be able to express the identity of input and output of a transition. In this example, if there were no way to express the identity between input and output, then we would be forced to have multiple transitions such as `a:a, b:b, c:c, d:d` for all of the relevant symbols; the introduction of predicates can be exploited in transducers only if identity between input and output can be expressed as well.

Expressing identity between input and output is crucial. This notion of identity can be seen as a consequence of the linguistic principle of *Faithfulness*: corresponding input and output segments tend to be identical. A similar argument is expressed in (Gildea and Jurafsky, 1996). Indeed, many interesting transducers are of the type ‘change all occurrences of α in some specific context into β, and pass on the rest of the input unaltered’. The various replacement and ‘local extension’ operators all produce transducers of this kind (Karttunen, 1995; Roche and Schabes, 1995; Karttunen, 1996; Kempe and Karttunen, 1996; Gerdemann and van Noord, 1999). Identities can be seen as a limited case of *backreferencing*. Backreferencing is an extension of regular expressions widely used in editors, scripting languages and other tools. A limited version of finite-state calculus backreferences is discussed in (Gerdemann and van Noord, 1999).

8.1.4 Smaller Automata

Another motivation for the introduction of predicates is the observation that the resulting automata are smaller. The size of automata is an important problem in practice (Daciuk, 1998; Kiraz, 1999). With predicates, potentially large sets of transitions are replaced by a single transition. For example, if an automaton has transitions from state p to state q over all ASCII symbols except for a symbol a, for which there is a transition from p to r, then there are 128 transitions leaving p. Using predicates, there are only two transitions leaving p (one labeled by a predicate $\{a\}$, and one labeled by $\overline{\{a\}}$). But note that similar space reductions can be achieved using *failure transitions* and related techniques (Kowaltowski, Lucchesi, and Stolfi, 1993; Kiraz, 1999; Daciuk, 2000a; Klarlund, 1998).

More interesting space reductions can be achieved in the case of transducers. The introduction of predicates with identity not only leads to transducers with fewer transitions, but also to transducers that have fewer states. This observation will be discussed in section 8.3.7. In section 8.4.2 we show that this space reduction is achieved for linguistically relevant examples too.

The implementation of various operations is faster for smaller automata. Although the implementation of some of the relevant operations becomes somewhat more complex, it is our experience that in almost all cases overall performance improves considerably.

8.1.5 Determinization of non-functional transducers

We show below that the introduction of predicates has the interesting effect that certain non-functional transducers can be treated by the transducer determinization algorithm (Oncina, García, and Vidal, 1993; Reutenauer, 1993; Mohri, 1996; Roche and Schabes, 1995; Roche and Schabes, 1997b). Therefore a larger class of transductions can be implemented efficiently.

8.1.6 Previous Work

A possible implementation of the question mark operator is the introduction of a special symbol **?** in finite state automata.[2] This special symbol is understood as 'any alphabet symbol not mentioned in the automaton', in order to translate examples such as `?-a`. This technique requires that each question mark operator is expanded into the set of symbols occurring in the regular expression as a whole. This solution (implemented in a previous version of the FSA Utilities (van Noord and Gerdemann, 2000) and in `xfst`, the Xerox regular expression compiler (Karttunen et al., 1996)) therefore leads to a proliferation of transitions. For example, the expression $(a..z \cdot ? - d)$ would result in an automaton with 52 transitions: 26 transitions from the initial state to an intermediate state for each of the letters of the alphabet and 26 transitions from this intermediate state to a final state for each

[2]Note that in such an implementation, the regular expression operator ? (any symbol) is not to be confused with the special symbol in automata **?** (any symbol not occurring in the automaton).

of the letters except 'd', as well as for **?**.[3]

The idea to allow predicates on transitions instead of symbols is also mentioned in (Watson, 1999a) and (Watson, 1999b). The details of this proposal, however, are not given. Apparently, in Watson's proposal predicates potentially inspect arbitrary parts of the input, and consume arbitrary prefixes of the input; the resulting formalism is therefore much more powerful, and hence various closure and efficiency properties are not applicable. In contrast, for the type of predicates proposed here, these attractive properties in fact are applicable, as is shown in the remainder of the article.

8.1.7 Overview

In the next section, predicate-augmented finite state recognizers are introduced, and it is shown how various operators and algorithms can be generalized. In section 8.3 predicate-augmented finite state transducers are introduced. We show that operations such as composition can be implemented straightforwardly; in addition we show how the transducer determinization algorithm can be generalized. The generalization leads to the definition of predicate-augmented finite state transducers with a bounded queue; the queue is required to be able to treat identities correctly. It is shown that this device allows a more compact representation of some finite-state transductions than the classical model. In section 8.5 we discuss some open problems and directions for future research.

8.2 Finite State Recognizers with Predicates

8.2.1 Definition

A predicate-augmented finite state recognizer (pfsr) M is specified by $(Q, \Sigma, \Pi, E, S, F)$ where Q is a finite set of states, Σ a set of symbols, Π a set of predicates over Σ, E a finite set of transitions $Q \times (\Pi \cup \{\epsilon\}) \times Q$. Furthermore, $S \subseteq Q$ is a set of start states and $F \subseteq Q$ is a set of final states.

The relation $\hat{E} \subseteq Q \times \Sigma^* \times Q$ is defined inductively:

1. for all $q \in Q$, $(q, \epsilon, q) \in \hat{E}$,
2. for all $(p, \epsilon, q) \in E$, $(p, \epsilon, q) \in \hat{E}$,
3. for all $(q_0, \pi, q) \in E$ and for all $\sigma \in \Sigma$, if $\pi(\sigma)$ then $(q_0, \sigma, q) \in \hat{E}$
4. if (q_0, x_1, q_1) and (q_1, x_2, q) are both in $\hat{E}$ then $(q_0, x_1x_2, q) \in \hat{E}$

The language $L(M)$ accepted by M is defined to be $\{w \in \Sigma^* | q_s \in S, q_f \in F, (q_s, w, q_f) \in \hat{E}\}$.

A pfsr is called ϵ-free if there are no $(p, \epsilon, q) \in E$. For any given pfsr there is an equivalent ϵ-free pfsr. It is straightforward to extend the corresponding algorithm for classical automata. Without loss of generality we assume below that pfsr are ϵ-free.

[3]Here we assume that we are not explicitly representing states which are not co-accessible, i.e. for which there is no path to a final state.

8.2.2 Properties

It is clear that in the case of recognizers, the addition of predicates is of limited theoretical interest. Let M^c be a classical finite automaton (Q, Σ, E, S, F) with Q a finite set of states, Σ a set of symbols, $S \subseteq Q$ the set of start states, $F \subseteq Q$ the set of final states and E a finite set of transitions $Q \times \Sigma \times Q$. Furthermore, let $s(p, q)$ be the set of symbols on transitions from p to q, i.e., $s(p, q) = \{\sigma | (p, \sigma, q) \in E\}$. If M^c is such a (minimal) finite automaton then clearly the equivalent (minimal) pfsr is given by $(Q, \Sigma, 2^{\Sigma}, E', S, F)$ where $E' = \{(p, s(p, q), q) | (p, s, q) \in E\}$. The construction in the other direction is similar.

The pfsr device typically is more compact in the number of transitions than an equivalent finite automaton. In the worst case, however, the number of transitions is the same (if it is the case for all states that its outgoing transitions have different target states for each symbol). In the best case, the number of transitions is reduced by a factor of $|\Sigma|$.

8.2.3 Operations on recognizers

Since predicate-augmented finite state recognizers are equivalent to ordinary finite-state automata, the class of languages defined by psfr is closed under the the usual regular operations such as *union*, *concatenation*, *Kleene-closure* and *reversal*. From a practical point of view, however, it is interesting to note that it is trivial to generalize the corresponding constructions for classical finite state automata (cf. for instance (Hopcroft and Ullman, 1979)). This means that the various constructions can be implemented directly, without the need to expand into ordinary finite automata first, which is impractical for large alphabets.

Intersection

An important and powerful operation is intersection. In the classical case, an automaton for the intersection of the languages defined by two given automata M_1 and M_2 is constructed by considering the cross product of states of M_1 and M_2. A transition $((p_1, p_2), \sigma, (q_1, q_2))$ exists iff the corresponding transition (p_1, σ, q_1) exists in M_1 and (p_2, σ, q_2) exists in M_2. In the case of pfsr a similar construction can be used, but instead of requiring that the symbol σ occurs in the corresponding transitions of M_1 and M_2, we require that the resulting predicate is the conjunction of the corresponding predicates in M_1 and M_2. The same technique is described in (Walther, 1999).

Given ϵ-free pfsr $M_1 = (Q_1, \Sigma, \Pi, E_1, S_1, F_1)$ and $M_2 = (Q_2, \Sigma, \Pi, E_2, S_2, F_2)$, the intersection $L(M_1) \cap L(M_2)$ is the language accepted by $M = (Q_1 \times Q_2, \Sigma, \Pi, E, S_1 \times S_2, F_1 \times F_2)$ and $E = \{((p_1, q_1), \pi_1 \wedge \pi_2, (p, q)) | (p_1, \pi_1, p) \in E_1, (q_1, \pi_2, q) \in E_2\}$.

Determinization

An ϵ-free pfsr is deterministic if there is a single start state, and if for all states $q \in Q$ and symbols $\sigma \in \Sigma$ there is at most one transition (q, π, q') such that $\pi(\sigma)$. If a pfsr M is deterministic then checking whether a given string w is accepted by M can be implemented efficiently: linear in w, and independent on the size of M.

A determinization algorithm (Aho, Sethi, and Ullman, 1986; Hopcroft and Ullman, 1979; Johnson and Wood, 1997) maintains subsets of states. Each subset is a state in the deterministic machine. To compute the transitions leaving a given subset D, a determinization algorithm computes for each symbol $\sigma \in \Sigma$ the set of states Q such that $p \in D, q \in Q$ and $(p, \sigma, q) \in E$.

In the case of predicates, however, transitions might overlap. For example, one transition may be applicable for high vowels, whereas another transition may be applicable for round vowels. In the determinized pfsr, such overlaps are not allowed. Therefore, we create a separate transition for high and round vowels, another transition for vowels which are high but not round, and a third transition for vowels which are round but not high.

In general, in order to compute the transitions leaving a given subset D we do as follows. Firstly we compute the function Trans^D: $\Pi \rightarrow 2^Q$, defined as: $\mathit{Trans}^D(\pi) = \{q \in Q | p \in D, (p, \pi, q) \in E\}$. For example, suppose $D = \{p\}$, and suppose we have transitions

$$\begin{aligned} E = \{ & (p, \pi_1, q_1), (p, \pi_1, q_2), (p, \pi_2, q_2), (p, \pi_2, q_3), \\ & (p, \pi_2, q_4), (p, \pi_3, q_3), (p, \pi_3, q_5)\} \end{aligned}$$

In that case:

$$\mathit{Trans}^D(\pi_1) = \{q_1, q_2\}, \mathit{Trans}^D(\pi_2) = \{q_2, q_3, q_4\}, \mathit{Trans}^D(\pi_3) = \{q_3, q_5\}$$

Let Π' be the predicates in the domain of Trans^D. For each split of Π' into two subsets $\pi_1 \ldots \pi_i$ and $\pi_{i+1} \ldots \pi_n$ we have a transition:

$$(D, \pi_1 \wedge \ldots \wedge \pi_i \wedge \neg\pi_{i+1} \wedge \ldots \wedge \neg\pi_n, \mathit{Trans}^D(\pi_1) \cup \ldots \cup \mathit{Trans}^D(\pi_i))$$

for the example we obtain the transitions:[4]

$$\begin{array}{lccr} (& D, & \pi_1 \wedge \pi_2 \wedge \pi_3, & \{q_1, q_2, q_3, q_4, q_5\} &) \\ (& D, & \pi_1 \wedge \pi_2 \wedge \neg\pi_3, & \{q_1, q_2, q_3, q_4\} &) \\ (& D, & \pi_1 \wedge \neg\pi_2 \wedge \pi_3, & \{q_1, q_2, q_3, q_5\} &) \\ (& D, & \pi_1 \wedge \neg\pi_2 \wedge \neg\pi_3, & \{q_1, q_2\} &) \\ (& D, & \neg\pi_1 \wedge \pi_2 \wedge \pi_3, & \{q_2, q_3, q_4, q_5\} &) \\ (& D, & \neg\pi_1 \wedge \pi_2 \wedge \neg\pi_3, & \{q_2, q_3, q_4\} &) \\ (& D, & \neg\pi_1 \wedge \neg\pi_2 \wedge \pi_3, & \{q_3, q_5\} &) \\ (& D, & \neg\pi_1 \wedge \neg\pi_2 \wedge \neg\pi_3, & \emptyset &) \end{array}$$

Complementation

If the determinizer also maintains the empty subset of states (cf. the last line in the previous example), then the resulting determinized automaton is *complete*: for each state a transition is applicable for each symbol of the alphabet. This property is important in order to define complementation. If an automaton M_1 with final states $F \subseteq Q$ is deterministic and complete, then an automaton accepting the language $\overline{L(M_1)}$ is obtained from M_1 simply by replacing F with $Q - F$.

As usual, the difference operation is defined straightforwardly in terms of complementation and intersection: if A and B are regular languages, then $A - B$ is defined as $A \wedge \overline{B}$.

[4]An implementation might choose to ignore transitions for which the corresponding predicate is not satisfiable.

Minimization

In Hopcroft's minimization algorithm (Hopcroft, 1971; Aho, Hopcroft, and Ullman, 1974b) a situation arises very similar to the determinization case. In this minimization algorithm, a partition of states is repeatedly refined by considering a pair of state and symbol which might reveal that an existing subset must be split. Rather than considering a pair of state and symbol, we consider in the generalization a pair of state and 'exclusive' predicate. As in the determinization algorithm we therefore need to consider all boolean combinations over the predicates present on a given state. In the actual implementation, we re-use the additional code required for the determinization algorithm in the implementation of the minimization algorithm.

The generalized minimization algorithm produces a pfsr that is minimal in the number of states. However, the pfsr is not necessarily unique, and could also be non-minimal in the number of transitions. This is caused by the fact that the predicates used in the pfsr might not be sufficiently general. For example, the language $\{a, b, c\}$ can be presented with a 2-state automaton with a single transition labeled $\in \{a, b, c\}$, but e.g. also with a 2-state automaton with two transitions labeled respectively by $\in \{a, b\}$ and $\in \{c\}$. Therefore, the minimization of a pfsr includes a final *cleanup* step in which for each pair of states p and q all transitions from p to q with labels $\pi_1 \ldots \pi_i$ are combined into a single transition from p to q with associated label $\pi_1 \vee \ldots \vee \pi_i$. It turns out that in the case of transducers, the corresponding cleanup operator is more difficult, as we discuss in section 8.5.1.

8.3 Transducers with Predicates and Identities

8.3.1 Definitions

A predicate-augmented finite state transducer (pfst) M is a tuple $(Q, \Sigma, \Pi, E, S, F)$ with Q a finite set of states, Σ a set of symbols, Π a set of predicates over Σ. As before, S and F are sets of start states and final states respectively. E is a finite set $Q \times (\Pi \cup \{\epsilon\}) \times (\Pi \cup \{\epsilon\}) \times Q \times \{0, 1\}$. The final component of a transition is used to indicate identities. For all transitions $(p, d, r, q, 1)$ it must be the case that $d = r \neq \epsilon$.[5]

We define the function *str* from $\Pi \cup \{\epsilon\}$ to 2^{Σ^*}.

$$\begin{aligned} \text{str}(\epsilon) &= \{\epsilon\} \\ \text{str}(\pi) &= \{\sigma \in \Sigma | \pi(\sigma)\} \end{aligned}$$

If $\pi \in \Pi$ and $\text{str}(\pi)$ is a singleton set, then the transitions (p, π, π, q, i) where $i \in \{0, 1\}$ are equivalent.

The relation $\hat{E} \subseteq Q \times \Sigma^* \times \Sigma^* \times Q$ is defined inductively.

1. for all p: $(p, \epsilon, \epsilon, p) \in \hat{E}$.
2. for all $(p, \phi, \psi, q, 0) \in E, x \in \text{str}(\phi), y \in \text{str}(\psi)$: $(p, x, y, q) \in \hat{E}$.
3. for all $(p, \pi, \pi, q, 1) \in E$, $x \in \text{str}(\pi)$: $(p, x, x, q) \in \hat{E}$.

[5]Note that without loss of generality we assume that there is no separate input and output alphabet, nor separate sets of predicates for input and output.

4. if (q_0, x_1, y_1, q_1) and (q_1, x_2, y_2, q) are both in $\widehat{E}$ then $(q_0, x_1x_2, y_1y_2, q) \in \widehat{E}$

The relation $R(M)$ accepted by a pfst M is defined to be $\{(w_d, w_r) | q_s \in S, q_f \in F, (q_s, w_d, w_r, q_f) \in \widehat{E}\}$.

8.3.2 Operations on transducers

It is immediately clear that if Σ is finite, a pfst defines a regular relation. Therefore, the relations defined by a pfst are closed under various operations such as *union*, *concatenation*, *Kleene closure* and *composition*. From a practical point of view, it is important to note that it is possible to adapt the constructions for classical transducers for pfst.

The introduction of predicates over symbols is straightforward for operations such as *union*, *concatenation*, *Kleene closure* and *cross-product*. The *identity* and *composition* operations are described now as follows.

Identity

The identity relation for a given language L is $\mathrm{id}(L) = \{(w, w) | w \in L\}$. For a given pfsr $M = (Q, \Sigma, \Pi, E, S, F)$, the identity relation is given by the pfst $M' = (Q, \Sigma, \Pi, E', S, F)$. Note that it would be wrong to define $E' = \{(p, \pi, \pi, q, 0) | (p, \pi, q) \in E\}$. Suppose π is true only of σ_1, σ_2. The pair $\pi : \pi$ then would be true of the pairs of symbols $\{(\sigma_1, \sigma_1), (\sigma_1, \sigma_2), (\sigma_2, \sigma_1), (\sigma_2, \sigma_2)\}$, whereas identity requires that we only allow the pairs $\{(\sigma_1, \sigma_1), (\sigma_2, \sigma_2)\}$. Another example to stress the point: the expression `identity(?)` ('copy') is quite different from `?:?` ('garbage-in garbage-out'). It is therefore necessary to introduce an identity marker for each of the transitions. The identity of a pfsr $M = (Q, \Sigma, \Pi, E, S, F)$ is given by $\mathrm{id}(M) = (Q, \Sigma, \Pi, E', S, F)$ where $E' = \{(p, \pi, \pi, q, 1) | (p, \pi, q) \in E\}$.

The operations *domain*, *range* and *inverse* are straightforward. For a given pfst $M = (Q, \Sigma, \Pi, E, S, F)$, we have:

- domain($R(M)$) is given by the pfsr $M' = (Q, \Sigma, \Pi, E', S, F)$ where $E' = \{(p, \phi, q) | (p, \phi, \psi, q, i) \in E\}$.
- range($R(M)$) is given by the pfsr $M' = (Q, \Sigma, \Pi, E', S, F)$ where $E' = \{(p, \psi, q) | (p, \phi, \psi, q, i) \in E\}$.
- inverse($R(M)$) is given by the pfst $M' = (Q, \Sigma, \Pi, E', S, F)$ where $E' = \{(p, \psi, \phi, q, i) | (p, \phi, \psi, q, i) \in E\}$.

Composition

The composition of two binary relations is $R_1 \circ R_2 = \{(x_1, x_3) | (x_1, x_2) \in R_1, (x_2, x_3) \in R_2\}$. The composition operation is perhaps the most important operation on transducers. Its implementation is similar to the intersection operation for recognizers. In the classical case, a transducer for the composition of two given transducers M_1 and M_2 is constructed by considering the cross product of states of M_1 and M_2. A transition $((p_1, p_2), \sigma_d, \sigma_r, (q_1, q_2))$ exists iff there is some σ such that the corresponding transition $(p_1, \sigma_d, \sigma, q_1)$ exists in M_1 and $(p_2, \sigma, \sigma_r, q_2)$ exists in M_2. In the case of pfst a similar construction can be used, but instead of requiring that the output part of a transition in M_1 is identical to the input part of a transition in M_2, we now merely require that the conjunction of both predicates is satisfiable. In the case of identities, some further complications arise. The effect of combining two transitions is defined by means of the function *ct* that takes two transitions and returns a new transition:

$$\begin{array}{ll}
\mathrm{ct}((p_1,\pi_1,\pi_1,q_1,1),(p_2,\pi_2,\pi_2,q_2,1)) & =((p_1,p_2),\pi_1\wedge\pi_2,\pi_1\wedge\pi_2,(q_1,q_2),1)\\
\mathrm{ct}((p_1,\phi,\pi_1,q_1,0),(p_2,\pi_2,\pi_2,q_2,1)) & =((p_1,p_2),\phi,\pi_1\wedge\pi_2,(q_1,q_2),0)\\
\mathrm{ct}((p_1,\pi_1,\pi_1,q_1,1),(p_2,\pi_2,\psi,q_2,0)) & =((p_1,p_2),\pi_1\wedge\pi_2,\psi,(q_1,q_2),0)\\
\mathrm{ct}((p_1,\phi,\pi_1,q_1,0),(p_2,\pi_2,\psi,q_2,0)) & =((p_1,p_2),\phi,\psi,(q_1,q_2),0)\\
 & \quad \text{if satisfiable}(\pi_1\wedge\pi_2)
\end{array}$$

Note that this function is not defined in case either the input part of the second transition or the output part of the first transition is ϵ. These cases are treated separately in the definition below. Given two pfst $M_1=(Q_1,\Sigma,\Pi,E_1,S_1,F_1)$ and $M_2=(Q_2,\Sigma,\Pi,E_2,S_2,F_2)$, the relation $R(M_1)\circ R(M_2)$ is defined by $M=(Q_1\times Q_2,\Sigma,\Pi,E,S_1\times S_2,F_1\times F_2)$ where

$$\begin{array}{rl}
E= & \{\mathrm{ct}(e_1,e_2)|e_1\in E_1,e_2\in E_2\}\\
\cup & \{((p_1,p_2),\epsilon,\psi,(p_1,q_2),0)|p_1\in Q_1,(p_2,\epsilon,\psi,q_2,0)\in E_2\}\\
\cup & \{((p_1,p_2),\phi,\epsilon,(q_1,p_2),0)|p_2\in Q_2,(p_1,\phi,\epsilon,q_1,0)\in E_1\}
\end{array}$$

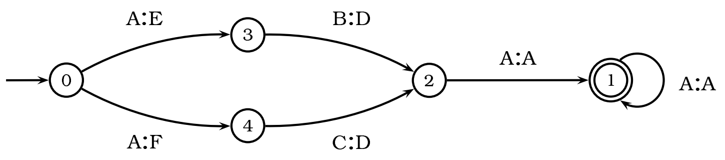
8.3.3 Determinization of Transducers

We will call a pfst M *deterministic* if M has a single start state, if there are no states $p,q\in Q$ such that $(p,\epsilon,\psi,q,i)\in E$, and if for all states p and symbols σ there is at most one transition (p,π_d,ψ,q,i) such that $\pi_d(\sigma)$. The transduction of an input string by means of a deterministic pfst is simple: in going through the input from left to right, you know exactly in which state you are (so there is no backtracking; alternatively if a parallel implementation is considered, there is no need to maintain a number of states linear in the size of the transducer). If a pfst M is deterministic then computing the transductions of a given string w as defined by M can be implemented efficiently. This computation is linear in w, and independent on the size of M. Since w can have several transductions (unless M is functional), we assume that this computation constructs a pfsr accepting $\{w'|(w,w')\in R(M)\}$.[6]

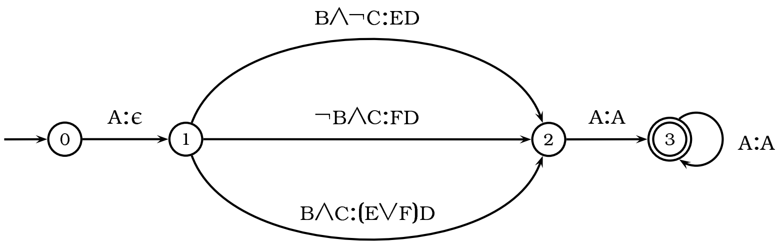
In order to extend the determinization algorithm for transducers (Oncina, García, and Vidal, 1993; Reutenauer, 1993; Mohri, 1996; Roche and Schabes, 1995; Roche and Schabes, 1997b), we must extend pfst in such a way that the output part of a transition is a sequence of predicates. This extension is described later, but first we illustrate some of the complications that arise. For the moment we will simply assume that the output part of a transition contains a sequence of predicates. We first create an equivalent pfst which has no ϵ on the domain part of transitions, using the same technique as described in (Roche and Schabes,

[6]As is well-known, not all finite-state transductions can be encoded by a deterministic transducer. As an example, a transduction which maps every a to a b if the input is of even length, and which maps every a to itself otherwise is a finite-state transduction, but cannot be encoded deterministically.

1997b, page 29).[7] In the determinization algorithm, local ambiguities such as those encountered in state 0 in (here, $\text{A}\ldots\text{F}$ are arbitrary predicates $\in \Pi$):



are solved by delaying the outputs as far as needed, until these symbols can be written out deterministically:[8]



The determinization algorithm for transducers maintains sets of pairs $Q \times \Pi^*$. Such a set corresponds to a state in the determinized transducer. In order to compute the transitions leaving such a set of pairs P, we compute for each π, $\mathit{Trans}^P(\pi) = \{(q, \phi\psi) | (p, \phi) \in P, (p, \pi, \psi, q) \in E\}$. In the example, we can be in states 3 and 4 after reading a symbol compatible with A, with pending outputs E and F. We thus have $P = \{(3, \text{E}), (4, \text{F})\}$. Therefore, we have:

$$\mathit{Trans}^P(\text{B}) = \{(2, \text{ED})\}, \mathit{Trans}^P(\text{C}) = \{(2, \text{FD})\}$$

Let Π' be the predicates in the domain of Trans^P. For each split of Π' into $\pi_1 \ldots \pi_i$ and $\pi_{i+1} \ldots \pi_n$ we have a proto-transition:

$$(P, \pi_1 \wedge \ldots \wedge \pi_i \wedge \neg\pi_{i+1} \wedge \ldots \wedge \neg\pi_n, \mathit{Trans}^P(\pi_1) \cup \ldots \cup \mathit{Trans}^P(\pi_i))$$

In the example, we have the following proto-transitions (we need not represent the $\emptyset$ state):

$$\begin{array}{ccccc} (& P, & \text{B} \wedge \neg\text{C}, & \{(2, \text{ED})\} &) \\ (& P, & \neg\text{B} \wedge \text{C}, & \{(2, \text{FD})\} &) \\ (& P, & \text{B} \wedge \text{C}, & \{(2, \text{ED}), (2, \text{FD})\} &) \end{array}$$

A transition is created from a proto-transition by removing the longest common prefix of predicates in the target pairs; this prefix is the sequence of output predicates of the resulting transition. However, before we remove this longest common prefix, we first consider possible simplifications in the sequences of output

[7]We represent emissions associated with final states, as they surface in the determinization algorithm below, using an extra transition with ϵ as the domain part. We thus allow transitions (p, ϵ, ψ, q) only in case q is a final state and there are no transitions leaving q.

[8]By 'writing out deterministically' we mean writing out with a deterministic state transition. Such 'deterministic' outputs may still in the end be rejected if for some input, the machine ends in a non-final state.

predicates, by packing multiple sequences associated with the same target state into a smaller number of sequences (using disjunction). In particular, two pairs of target state and predicate sequences (p_1, ψ_1) and (p_2, ψ_2) can be combined into a single pair (p, ψ) iff $p_1 = p_2 = p$ and $\psi_1 = \pi_1 \ldots \pi_i \ldots \pi_n$, $\psi_2 = \pi_1 \ldots \pi_i' \ldots \pi_n$ and $\psi = \pi_1 \ldots \pi_i \vee \pi_i' \ldots \pi_n$. In a proto-transition this simplification is applied repeatedly until no further simplifications are possible.

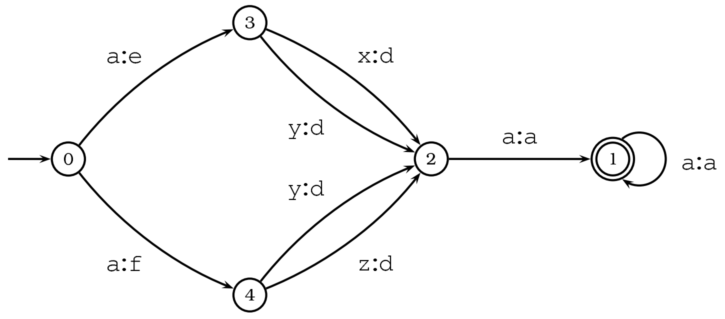
Here, the third proto-transition is simplified into:

$$(\quad P, \quad \text{B} \wedge \text{C}, \quad \{(2, (\text{E} \vee \text{F})\text{D})\} \quad)$$

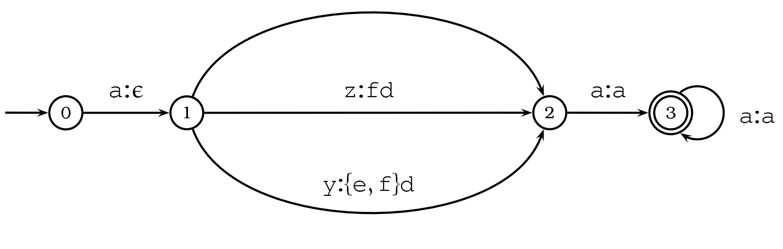
Moving the longest common prefix into the output part of the label yields:

$$\begin{array}{llll}
(& P, & \text{B} \wedge \neg \text{C} : \text{ED}, & \{(2, \epsilon)\} \;) \\
(& P, & \neg \text{B} \wedge \text{C} : \text{FD}, & \{(2, \epsilon)\} \;) \\
(& P, & \text{B} \wedge \text{C} : (\text{E} \vee \text{F})\text{D}, & \{(2, \epsilon)\} \;)
\end{array}$$

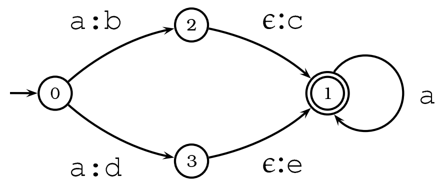
The introduction of predicates thus has the interesting effect that certain non-functional transducers can be treated by the transducer determinization algorithm. Assume that B is the predicate $\{x, y\}$, C is the predicate $\{y, z\}$ and the predicates A, D, E and F are true only of the symbols a, d, e and f respectively. The equivalent normal transducer is:



This transducer cannot be treated by the transducer determinization algorithm (that algorithm does allow a limited form of ambiguity, but only if this ambiguity can be delayed to a final state; here this is not possible). However, the same transduction can be determinized if expressed by a pfst:

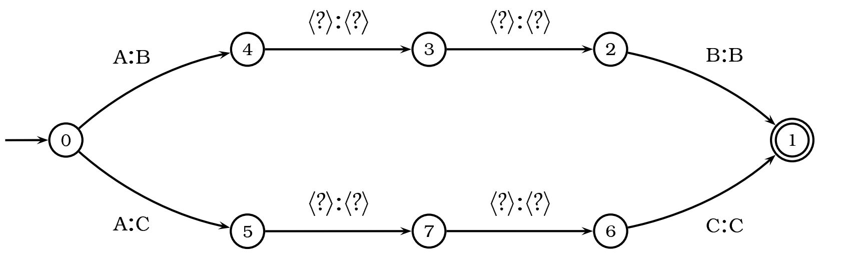


If predicates are used, then a larger class of transductions can be implemented efficiently. A precise classification of this class is beyond the scope of this paper, but note that the type of ambiguities that can be implemented in this way is limited to ambiguities that extend only over a single symbol.[9] For instance, a simple example such as the following cannot be determinized:

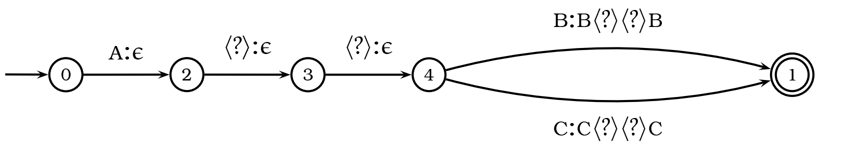


8.3.4 Determinization and identities

To treat identities, we must assume in the definition of proto-transition that if one of the positively occurring predicates in the boolean combination is associated with an identity, then the resulting predicate is associated with an identity as well. As an example consider the following transducer. For simplicity we assume B and C are mutually exclusive predicates; as before ? is a predicate which is true of all symbols. Also, we write ⟨A⟩:⟨A⟩ for a transition A:A with an associated identity constraint.



Determinization produces:



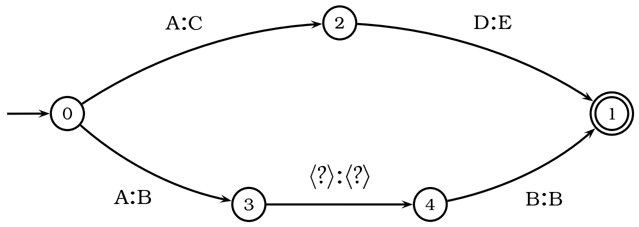
Outputs associated with an identity are delayed like ordinary outputs. Generalizing an idea due to Tamás Gaál and Lauri Karttunen[10] transducers with such disconnected identities are interpreted as follows. During the transduction of a

[9]Of related interest is the approach of (Kempe, 2000). He shows that ambiguous transductions can be computed efficiently by factorizing an ambiguous transducer T into a functional transducer T_1 and an ambiguous transducer T_2 such that T is equivalent to $T_1 \circ T_2$, and such that T_2 contains no 'failing paths'. In typical cases, T_1 contains meta-symbols which are expanded in T_2. This approach is more general in the sense that these meta-symbols range over sequences of symbols, rather than single symbols. It is more limited in the sense that identities cannot be expressed.

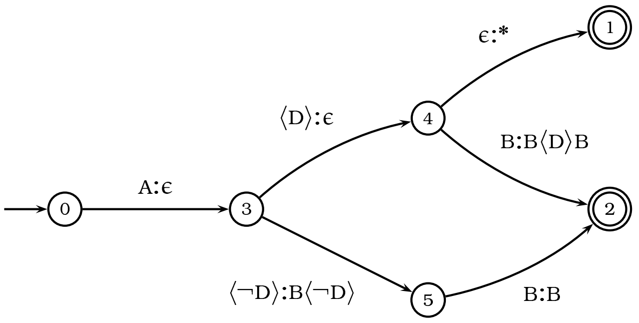
[10]personal communication

string, a queue is maintained. Each time an input symbol is matched by a predicate with an associated identity, this symbol is enqueued. If a symbol matched by the corresponding predicate on the output side has to be written, then that symbol is obtained by a dequeue operation. With this use of a queue, our method for interpreting a transducer is no longer finite state. The transducer itself, however, still encodes a regular transduction.

A complication arises in cases like:



Determinization yields:



What sequence of output predicates should be put on the position of the *? According to the definitions, we get CE. However, this is not right because then there is a path $0 \to 3 \to 4 \to 1$ which has an identity on the input side without a corresponding identity on the output. Embedding such examples would lead to transducers in which identities are 'out of sync'. The determinization algorithm is therefore extended by marking in the output part that the scope of an identity ends; procedurally such a mark is interpreted as a dequeue operation which ignores the dequeued value. We write such a mark as $\langle\rangle$. In the example the sequence of outputs X becomes CE$\langle\rangle$. In the definition of proto-transition, if at least one of the positively occurring π_k has an associated identity then we append a $\langle\rangle$ mark to each of the outputs $\mathit{Trans}^P(\pi_l)$ for which π_l was not associated with an identity.

8.3.5 Finite State Transducers with a bounded Queue

We are now ready to define predicate-augmented finite state transducers with a bounded queue. A predicate-augmented finite state transducer with queue (qpfst) M is a tuple $(Q, \Sigma, \Pi, E, S, F)$ with Q a finite set of states, Σ a set of symbols, Π a set of predicates over Σ. As before, S and F are sets of start states and final states respectively. E is a finite set $Q \times ((\Pi \cup \{\epsilon\}) \times \{0,1\}) \times ((\Pi \cup \{\lambda\}) \times \{0,1\})^* \times Q$.

In a transition, each predicate is associated with a queue marker, which is one of $\{0,1\}$. On the input side, 1 will imply an enqueue operation of the symbol matching the predicate; on the output side 1 will imply a dequeue operation of the symbol matching the predicate. In the input part of the transition, ϵ can be used as well, in which case the queue marker must be 0 (input epsilons will be employed to represent outputs associated with initial and final states). In the output part of a transition we can have λ instead of a predicate, in order to represent the explicit dequeue operations motivated earlier. We require that every λ must have a corresponding queue marker which is 1.

The relation $O : ((\Pi \cup \{\lambda\}) \times \{0,1\})^* \times \Sigma^* \times \Sigma^* \times \Sigma^*$ determines the effect of the output part of a transition. Its arguments represent respectively the output sequence of a transition, the (incoming and outgoing) queues, and the resulting output string. Note that queues are written from left to right in such a way that an element is enqueued to the left and dequeued from the right.

1. for all $x \in \Sigma^*$ we have $(\epsilon, x, x, \epsilon) \in O$
2. if $(\phi, x_0, x, z) \in O$ then for all $\sigma \in \Sigma$ we have $((\lambda, 1)\phi, x_0\sigma, x, z) \in O$
3. if $(\phi, x_0, x, z) \in O$ then for all $\sigma \in \Sigma$ and $\pi \in \Pi$ such that $\pi(\sigma)$ we have $((\pi, 1)\phi, x_0\sigma, x, \sigma z) \in O$
4. if $(\phi, x_0, x, z) \in O$ then for all $\sigma \in \Sigma$ and $\pi \in \Pi$ such that $\pi(\sigma)$ we have $((\pi, 0)\phi, x_0, x, \sigma z) \in O$

The relation $\widehat{E} \subseteq Q \times \Sigma^* \times \Sigma^* \times Q \times \Sigma^* \times \Sigma^*$ is a relation between source states, sequences of input symbols, sequences of output symbols, target states, and source- and target queues. It is defined inductively.

1. for all $p \in Q$, $(p, \epsilon, \epsilon, p, \epsilon, \epsilon) \in \widehat{E}$.
2. for each transition $(p, (\epsilon, 0), \phi, q) \in E$ such that $(\phi, x_0, x, w) \in O$, $(p, \epsilon, w, q, x_0, x) \in \widehat{E}$
3. for each transition $(p, (\pi, 0), \phi, q) \in E$ such that $\pi(\sigma)$ and $(\phi, x_0, x, w) \in O$, $(p, \sigma, w, q, x_0, x) \in \widehat{E}$.
4. for each transition $(p, (\pi, 1), \phi, q) \in E$ such that $\pi(\sigma)$ and $(\phi, \sigma x_0, x, w) \in O$, $(p, \sigma, w, q, x_0, x) \in \widehat{E}$.
5. if $(q_0, x_1, y_1, q_1, x_0, x_1)$ and $(q_1, x_2, y_2, q, x_1, x)$ are both in $\widehat{E}$ then $(q_0, x_1x_2, y_1y_2, q, x_0, x) \in \widehat{E}$

The relation $R(M)$ accepted by a qpfst M is defined to be $\{(w_d, w_r) | q_s \in S, q_f \in F, (q_s, w_d, w_r, q_f, \epsilon, \epsilon) \in \widehat{E}\}$.

Such qpfst are generally very powerful. However, the qpfst which result from the generalized transducer determinization algorithm are all limited. Not only are these transducers deterministic by construction, but they are also limited in the way the queue is actually used: in each case the maximum size of the queue is some constant. And of course, since the input was a finite-state transducer, the resulting equivalent qpfst describes a finite-state transduction too. Another way to characterize this limited use of qpfst is to observe that in such cases every cyclic path through such a transducer will have identical input and output queue: the queue is only used in a strictly local sense.

The ordinary transducer determinization algorithm is guaranteed to terminate only if the input transducer can be determined, i.e., the transducer must be *subsequential.* A separate algorithm exists to check a given transducer for subsequentiality (section 8.5.2). The same termination property holds for the generalized transducer determinization algorithm. If the generalized transducer determinization algorithm terminates for a given pfst, then the result is an equivalent deterministic qpfst. The application of a determinized (potentially non-functional) qpfst T to a given string w is linear in the size of w, and independent of the size of T.

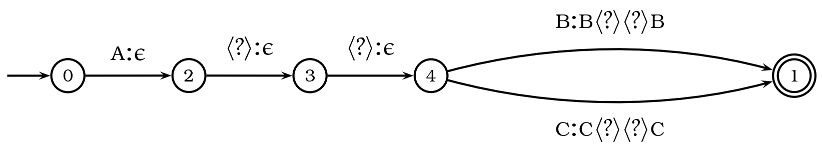
8.3.6 Synchronization

Operations such as composition are defined for pfst. Therefore, we have implemented an operator which transforms a given bounded qpfst back into pfst by synchronizing the identities. Of course, the resulting pfst will generally not be deterministic anymore.

The synchronization is implemented by an algorithm which maintains an agenda of 'synchronous states' (initialized by the set of start states). For each state on the agenda minimal synchronous paths are generated. The target states of these paths are added to the agenda, and these paths themselves are broken into pieces such that each piece is synchronous (by introducing transitions with ϵ on the input or output side).

8.3.7 Succinctness

Predicate-augmented finite state transducers typically require fewer transitions than classical finite state transducers, by an argument similar to that for pfsr. In the case of predicate-augmented pfst with bounded queue, however, the number of states can often be much smaller than the number of states in an equivalent, classical, sub-sequential transducer. Consider again the first example in section 8.3.4. Application of our variant of Mohri's determinization algorithm yields a transducer of 5 states and 5 transitions, repeated here for convenience:



Suppose we were to expand this example into a classical subsequential transducer, then depending on the size of the alphabet, the resulting transducers would have many more states. The example for the alphabet {x, y, z} with 15 states and 31 transitions is given in figure 8.1; for an alphabet of 26 symbols, the result al-

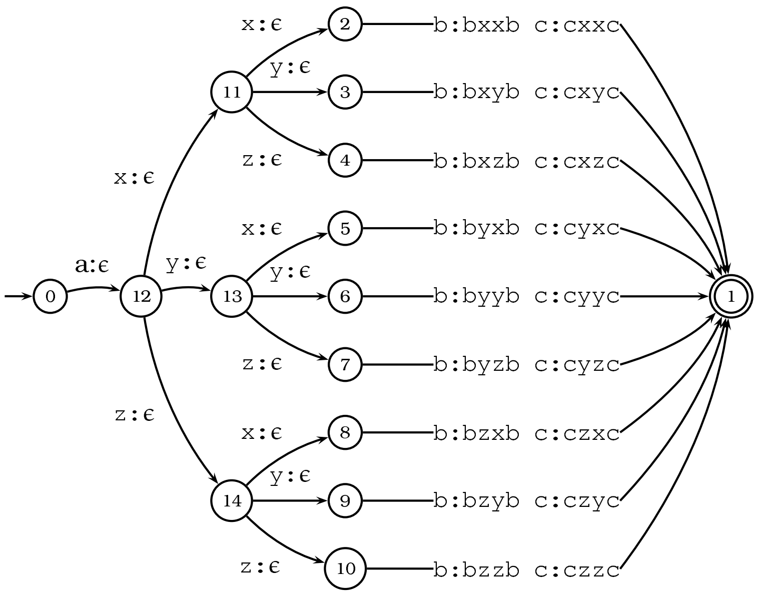
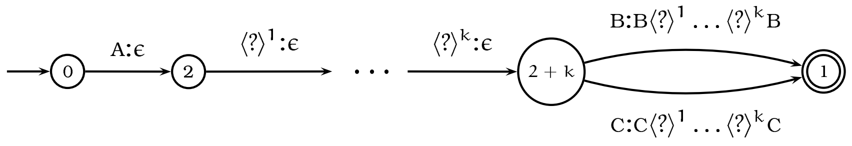


Figure 8.1: A minimal subsequential transducer without predicates. The equivalent minimal transducer employing predicates only has 5 states and 5 transitions.

ready has 705 states and 2055 transitions. For an alphabet of 254 symbols, the result has 64773 states and 193803 transitions. Instead of having two question marks on the input side in a row, consider similar examples where we have k such question marks in a row:



In these cases, the minimal qpfst will have $3 + k$ states. The equivalent minimal subsequential transducer will require $3+|\Sigma|+|\Sigma|^2+\ldots+|\Sigma|^k$ states. An analysis of the difference in succintness in terms of descriptional complexity (e.g. (Dassow, Paun, and Salomaa, 1997)) is beyond the scope of this article; but this class of examples suggests that there are arbitrarily many relations for which the qpfst device requires exponentially fewer states than subsequential transducers.

8.4 Practical Considerations

Predicate-augmented finite state automata are fully integrated in version 6 of the Fsa Utilities toolbox. The toolbox is freely available from `http://www.let.rug.nl/~vannoord/Fsa/`. In addition, some of the algorithms have been implemented in C++.

8.4.1 Membership and Non-membership Predicates

In practice, we have mostly assumed that all predicates are of the form $\in S$ and $\notin S$ for arbitrary finite sets of symbols S. The non-membership predicates are very useful to specify in a compact form large (potentially infinite) sets of symbols. A boolean combination of membership and non-membership predicates can always be written in this form, as the following table shows:

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$
$\in S_1$	$\in S_2$	$\notin S_1$	$\in S_1 \cap S_2$	$\in S_1 \cup S_2$
$\in S_1$	$\notin S_2$		$\in S_1 - S_2$	$\notin S_2 - S_1$
$\notin S_1$	$\in S_2$	$\in S_1$	$\in S_2 - S_1$	$\notin S_1 - S_2$
$\notin S_1$	$\notin S_2$		$\notin S_2 \cup S_1$	$\notin S_1 \cap S_2$

In the implementation, any boolean combination of predicates that occurs is immediately rewritten into this atomic form. Determining whether a symbol satisfies a predicate is trivial. Determining satisfiability of an atomic formula is trivial too: the only atomic formula that is not satisfiable is $\in \emptyset$. The actual computation thus involves standard operations on sets: membership, union, intersection and difference. The implementation provides three alternative implementations, by representing sets as ordered lists, bit vectors or balanced binary trees.

The system also supports the addition of various application-specific predicate sets. There are various possibilities here. For instance, predicates could be expressed in terms of type hierarchies as in (Carpenter, 1992). Another possibility is a predicate module in which predicates are membership tests of regular languages. A syntax component could be implemented by a pfsr in which predicates describe words. These predicates themselves might be implemented by finite automata over character strings. If predicates get complicated, the efficiency of checking such predicates may become important.

8.4.2 Smaller Transducers

The operations on predicate-augmented finite state recognizers and transducers discussed here have been fully implemented and integrated in a finite state toolkit. Although the implementation of these operations is more involved than for normal automata it turned out that the introduction of predicates has improved performance considerably, because automata are smaller.

For example, consider the soundex algorithm expressed as a regular expression, presented at the Xerox web-site.[11] The soundex algorithm maps proper names to four-letter codes, where 'similar' names are assigned the same code. This algorithm can be used to match names that are misspelled, for instance due to poor handwriting or voice transmission; similar problems occur in historical archives. A description of the algorithm and some historical remarks are given in (Knuth, 1998). The compilation of the soundex regular expression yields a transducer with

[11] http://www.rxrc.xerox.com/research/mltt/fst/fsexamples.html

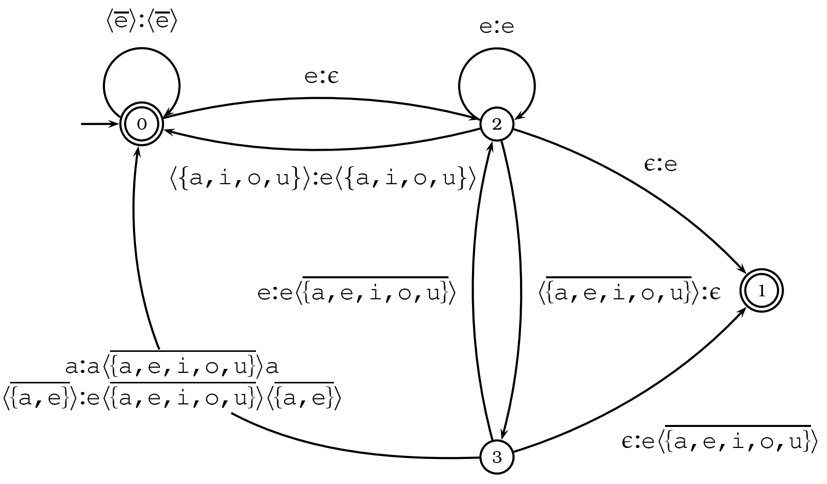

$\langle\overline{e}\rangle$:$\langle\overline{e}\rangle$
e:e
e:ϵ
0
2
1
3
⟨{a,i,o,u}⟩:e⟨{a,i,o,u}⟩
ϵ:e
e:e$\langle\overline{\{a,e,i,o,u\}}\rangle$
$\langle\overline{\{a,e,i,o,u\}}\rangle$:ϵ
a:a$\langle\overline{\{a,e,i,o,u\}}\rangle$a
$\langle\overline{\{a,e\}}\rangle$:e$\langle\overline{\{a,e,i,o,u\}}\rangle\langle\overline{\{a,e\}}\rangle$
ϵ:e$\langle\overline{\{a,e,i,o,u\}}\rangle$

Figure 8.2: A minimal transducer with predicates implementing the phonological rule e → a/_C a. The equivalent minimal transducer without predicates has 24 states and 620 transitions.

1217 transitions. By design, the soundex algorithm treats various classes of characters identically. Using predicates for each of these classes yields a transducer with 198 transitions. The construction is four times faster as well. Depending on how predicates are implemented, running the resulting transducer might be slower. In our experiments these effects were not noticeable.

The observation that the use of predicates generally leads to transducers with fewer states can be observed in practically relevant examples as well. Consider the following hypothetical phonological rule:

$$\texttt{e} \rightarrow \texttt{a}/\_\texttt{C a}$$

This rule indicates that an e should be mapped to an a if it is followed by a consonant and an a. Assuming an alphabet consisting of 5 vowels and 21 consonants, the corresponding minimal transducer for this example consists of 24 states and 620 transitions. If predicates are used, the resulting automaton only has 4 states and 10 transitions (cf. figure 8.2).

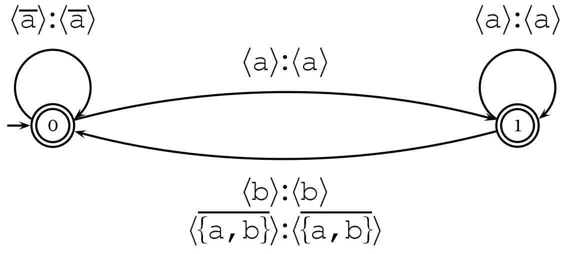
8.5 Future Work

8.5.1 Minimization

The minimization algorithm for transducers (Mohri, 1994; Mohri, 2000) can be applied to a bounded qpfst without modifications. The transducer minimization algorithm consists of two steps. In the first step, all output symbols are moved into preceding transitions as much as is possible. This is done by computing for each state the longest common prefix of the outputs associated with all paths from that state to a final state. The second step of the transducer minimization algorithm consists of the application of ordinary recognizer minimization to the resulting transducer, temporarily treating the labels as atomic symbols.

The application of the transducer minimization algorithm to a bounded qfst might result in a qpfst with identities in which the output has to be produced before the corresponding input symbol has been observed. The queue-mechanism can be generalized to treat such cases as well. We use an implementation of queues described in (Sterling and Shapiro, 1994, page 299) in which an element can be dequeued before it is enqueued. The output is a variable temporarily; obviously this requires output to be buffered. We implemented this both in C++ as well as in Prolog.

However, applying the transducer minimization algorithm in this way does not neccessarily produce a minimal qpfst. One problem is that in the transducer minimization algorithm, the final step consists of an application of the recognizer minimization algorithm in such a way that the labels of the transducer are temporarily treated as unanalyzable atoms. This works in the case of ordinary transducers, but is not good enough for our purposes. The following example illustrates this particular problem.



The transduction implemented by this transducer is simply the identity relation over Σ^*. However, the application of the transducer minimization algorithm will produce an identical transducer, rather than the minimal one.

In the implementation in the Fsa Utilities we have constructed a variety of heuristics, which includes a generalization of the transducer minimization algorithm, in order to reduce the size of deterministic transducers. In most practical cases, the heuristics produce a minimal transducer.

8.5.2 Subsequentiality and Bi-machines

Recall that the transducer determinization algorithm is guaranteed to terminate only in case the input transducer can be determinized, i.e., the transducer describes a subsequential transduction. Therefore, it is important to implement an algorithm which checks for this property. We are working on an algorithm to check subsequentiality of a given pfst, based on the algorithm presented in (Roche and Schabes, 1997b). We have adapted the algorithm proposed in (Roche and Schabes, 1997b) since it fails to treat certain types of transducer correctly; we intend to provide details somewhere else.

A further natural extension is the generalization of bi-machines and the related algorithms to the case of predicates.

Chapter 9

Approximation and Exactness in Finite State Optimality Theory

The research reported in this chapter is conducted in close cooperation with Dale Gerdemann (University of Tübingen).

9.1 Introduction

Finite state methods have proven quite successful for encoding rule-based generative phonology (Johnson, 1972; Kaplan and Kay, 1994). Recently, however, Optimality Theory (Prince and Smolensky, 1993) has emphasized phonological accounts with default constraints on surface forms. While Optimality Theory (OT) has been successful in explaining certain phonological phenomena such as *conspiracies* (Kisseberth, 1970), it has been less successful for computation. The negative result of (Frank and Satta, 1998) has shown that in the general case the method of counting constraint violations takes OT beyond the power of regular relations. To handle such constraints, (Karttunen, 1998) has proposed a finite-state approximation that counts constraint violations up to a predetermined bound. Unlike previous approaches (Ellison, 1994; Walther, 1999), Karttunen's approach is encoded entirely in the finite state calculus, with no extra-logical procedures for counting constraint violations.

In this paper, we will present a new approach that seeks to minimize constraint violations without counting. Rather than counting, our approach employs a filter based on matching constraint violations against violations in alternatively derivable strings. As in Karttunen's counting approach, our approach uses purely finite state methods without extra-logical procedures. We show that our *matching* approach is superior to the *counting* approach for both size of resulting automata and closeness of approximation. The matching approach can in fact exactly model many OT analyses where the counting approach yields only an approximation; yet, the size of the resulting automaton is typically much smaller.

In this paper we will illustrate the matching approach and compare it with the counting approach on the basis of the Prince & Smolensky syllable structure example (Prince and Smolensky, 1993; Ellison, 1994; Tesar, 1995), for each of the different constraint orderings identified in Prince & Smolensky.

`[]`	empty string
`[E1,E2,...,En]`	concatenation of `E1...En`
`{}`	empty language
`{E1,E2,...,En}`	union of `E1...En`
`(E)`	grouping for op. precedence
`E*`	Kleene closure
`E+`	Kleene plus
`E^`	optionality
`E1 - E2`	difference
`~E`	complement
`$ E`	containment
`E1 & E2`	intersection
`?`	any symbol
`E1 x E2`	cross-product
`A o B`	composition
`domain(E)`	domain of a transduction
`range(E)`	range of a transduction
`identity(E)`	identity transduction[2]
`inverse(E)`	inverse transduction

Table 9.1: Regular expression operators.

9.2 Finite State Phonology

9.2.1 Finite State Calculus

Finite state approaches have proven to be very successful for efficient encoding of phonological rules. In particular, the work of (Kaplan and Kay, 1994) has provided a compiler from classical generative phonology rewriting rules to finite state transducers. This work has clearly shown how apparently procedural rules can be recast in a declarative, reversible framework.

In the process of developing their rule compiler, Kaplan & Kay also developed a high-level finite state calculus. They argue convincingly that this calculus provides an appropriate high-level approach for expressing regular languages and relations. The alternative conception in term of states and transitions can become unwieldy for all but the simplest cases.[1]

Kaplan & Kay's finite state calculus now exists in multiple implementations, the most well-known of which is that of (Karttunen et al., 1996). In this paper, however, we will use the alternative implementation provided by the FSA Utilities (van Noord, 1997b; van Noord, 1999; van Noord and Gerdemann, 2000). The FSA Utilities allows the programmer to introduce new regular expression operators of arbitrary complexity. This higher-level interface allows us to express our algorithm more easily. The syntax of the FSA Utilities calculus is summarized in Table 9.1.

The finite state calculus has proven to be a very useful tool for the development of higher-level finite state operators (Karttunen, 1995; Kempe and Karttunen, 1996;

[1]Although in some cases such a direct implementation can be much more efficient (Mohri and Sproat, 1996; van Noord and Gerdemann, 2000).

[2]If an expression for a recognizer occurs in a context where a transducer is required, the identity operation will be used implicitly for coercion.

Karttunen, 1996; Gerdemann and van Noord, 1999). An interesting feature of most such operators is that they are implemented using a generate-and-test paradigm. (Karttunen, 1996), for example, introduces an algorithm for a leftmost-longest replacement operator. Somewhat simplified, we may view this algorithm as having two steps. First, the generator freely marks up possible replacement sites. Then the tester, which is an identity transducer, filters out those cases not conforming to the leftmost-longest strategy. Since the generator and tester are both implemented as transducers, they can be composed into a single transducer, which eliminates the inefficiency normally associated with generate-and-test algorithms.

9.2.2 Finite State Optimality Theory

The generate-and-test paradigm initially appears to be appropriate for optimality theory. If, as claimed in (Ellison, 1994), *Gen* is a regular relation and if each constraint can be implemented as an identity transducer, then optimality theory analyses could be implemented as in 9.1.

```
     Gen
      o
 Constraint1
      o
     ...
      o
 ConstraintN
```

Figure 9.1: Optimality Theory as Generate and Test

The problem with this simple approach is that in OT, a constraint is allowed to be violated if none of the candidates satisfy that constraint. (Karttunen, 1998) treats this problem by providing a new operator for *lenient* composition, which is defined in terms of the auxiliary operation of priority union. In the FSA Utilities calculus, these operations can be defined as:[3]

```
macro(priority_union(Q,R),
      {Q, ~domain(Q) o R}).
macro(lenient_composition(S,C),
      priority_union(S o C, S)).
```

The effect here is that the lenient composition of `S` and `C` is the composition of `S` and `C`, except for those elements in the domain of `S` that are not mapped to anything by `S o C`. For these elements not in the domain of `S o C`, the effect is the same as the effect of `S` alone. We use the notation `S lc C` as a succinct notation for the lenient composition of S and C. Using lenient composition an OT analysis can be written as in 9.2.

[3]The notation `macro(Expr1,Expr2)` is used to indicate that the regular expression `Expr1` is an abbreviation for the expression `Expr2`. Because Prolog variables are allowed in both expressions this turns out to be an intuitive and powerful notation (van Noord and Gerdemann, 2000).

```
Gen
 lc
Constraint1
 lc
 ...
 lc
ConstraintN
```

Figure 9.2: Optimality Theory as Generate and Test with Lenient Composition

The use of lenient composition, however, is not sufficient for implementing optimality theory. In general, a candidate string can violate a constraint multiple times and candidates that violate the constraint the least number of times need to be preferred. Lenient composition is sufficient to prefer a candidate that violates the constraint 0 times over a candidate that violates the constraint at least once. However, lenient composition cannot distinguish two candidates if the first contains one violation, and the second contains at least two violations.

The problem of implementing optimality theory becomes considerably harder when constraint violations need to be counted. As (Frank and Satta, 1998) have shown, an OT describes a regular relation under the assumptions that `Gen` is a regular relation, and each of the constraints is a regular relation which maps a candidate string to a natural number (indicating the number of constraint violations in that candidate), where the range of each constraint is finite. If constraints are defined in such a way that there is no bound to the number of constraint violations that can occur in a given string, then the resulting OT may describe a relation that is not regular. A simple example of such an OT (attributed to Markus Hiller) is the OT in which the inputs of interest are of the form `[a*,b*]`, *Gen* is defined as a transducer which maps all `a`'s to `b`'s and all `b`'s to `a`'s, or alternatively, it performs the identity map on each `a` and `b`:

```
{[(a x b)*,(b x a)*],
 [(a x a)*,(b x b)*]}
```

This OT contains only a single constraint, *A: a string should not contain `a`. As can easily be verified, this OT defines the relation $\{(a^n b^m, a^n b^m) | n \leq m\} \cup \{(a^n b^m, b^n a^m) | m \leq n\}$, which can easily be shown to be non-regular.

Although the OT described above is highly unrealistic for natural language, one might nevertheless expect that a constraint on syllable structure in the analysis of Prince & Smolensky would require an unbounded amount of counting (since words are of unbounded length), and that therefore such analyses would not be describable as regular relations. An important conclusion of this paper is that, contrary to this potential expectation, such cases in fact *can* be shown to be regular.

9.2.3 Syllabification in Finite State OT

In order to illustrate our approach, we will start with a finite state implementation of the syllabification analysis as presented in chapter 6 of (Prince and Smolensky, 1993). This section is heavily based on (Karttunen, 1998), which the reader should consult for more explanation and examples.

The inputs to the syllabification OT are sequences of consonants and vowels. The input will be marked up with *onset*, *nucleus*, *coda* and *unparsed* brackets; where a syllable is a sequence of an optional onset, followed by a nucleus, followed by an optional coda. The input will be marked up as a sequence of such syllables, where at arbitrary places *unparsed* material can intervene. The assumption is that an unparsed vowel or consonant is not spelled out phonetically. Onsets, nuclei and codas are also allowed to be empty; the phonetic interpretation of such constituents is *epenthesis*.

First we give a number of simple abbreviations:

```
macro(cons,
      {b,c,d,f,g,h,j,k,l,m,n,
       p,q,r,s,t,v,w,x,y,z}  ).
macro(vowel,   {a,e,o,u,i}).

macro(o_br,    'O['). % onset
macro(n_br,    'N['). % nucleus
macro(d_br,    'D['). % coda
macro(x_br,    'X['). % unparsed
macro(r_br,    ']').
macro(bracket,
      {o_br,n_br,d_br,x_br,r_br}).

macro(onset,   [o_br,cons^  ,r_br]).
macro(nucleus, [n_br,vowel^ ,r_br]).
macro(coda,    [d_br,cons^  ,r_br]).
macro(unparsed,[x_br,letter ,r_br]).
```

Following Karttunen, *Gen* is formalized as in 9.3. Here, `parse` introduces *onset*, *coda* or *unparsed* brackets around each consonant, and *nucleus* or *unparsed* brackets around each vowel. The `replace(T,Left,Right)` transducer applies transducer `T` obligatory within the contexts specified by `Left` and `Right` (Gerdemann and van Noord, 1999). The `replace(T)` transducer is an abbreviation for `replace(T,[],[])`, i.e. `T` is applied everywhere. The *overparse* transducer introduces optional 'empty' constituents in the input, using the *intro_each_pos* operator.[4]

In the definitions for the constraints, we will deviate somewhat from Karttunen. In his formalization, a constraint simply describes the set of strings which do not violate that constraint. It turns out to be easier for our extension of Karttunen's formalization below, as well as for our alternative approach, if we return to the concept of a constraint as introduced by Prince and Smolensky where a constraint

[4]An alternative would be to define *overparse* with a Kleene star in place of the option operator. This would introduce unbounded sequences of empty segments. Even though it can be shown that, with the constraints assumed here, no optimal candidate ever contains two empty segments in a row (proposition 4 of (Prince and Smolensky, 1993)) it is perhaps interesting to note that defining *Gen* in this alternative way causes cases of infinite ambiguity for the counting approach but is unproblematic for the matching approach.

```
macro(gen,        {cons,vowel}*
                        o
                    overparse
                        o
                      parse
                        o
                syllable_structure ).

macro(parse, replace([[] x {o_br,d_br,x_br},cons, [] x r_br])
                                  o
             replace([[] x {n_br,x_br},     vowel,[] x r_br])).

macro(overparse,intro_each_pos([{o_br,d_br,n_br},r_br]^)).

macro(intro_each_pos(E), [[ [] x E, ?]*,[] x E]).

macro(syllable_structure,ignore([onset^,nucleus,coda^],unparsed)*).
```

Figure 9.3: The definition of *Gen*

adds *marks* in the candidate string at the position where the string violates the constraint. Here we use the symbol @ to indicate a constraint violation. After checking each constraint the markers will be removed, so that markers for one constraint will not be confused with markers for the next.

```
macro(mark_violation(parse),
     replace(([] x @),x_br,[]).

macro(mark_violation(no_coda),
     replace(([] x @),d_br,[]).

macro(mark_violation(fill_nuc),
     replace(([] x @),[n_br,r_br],[])).

macro(mark_violation(fill_ons),
     replace(([] x @),[o_br,r_br],[])).

macro(mark_violation(have_ons),
     replace(([] x @),[],n_br)
                 o
     replace((@ x []),onset,[])).
```

The *parse* constraint simply states that a candidate must not contain an unparsed constituent. Thus, we add a mark after each unparsed bracket. The *no_coda* constraint is similar: each coda bracket will be marked. The *fill_nuc* constraint is only slightly more complicated: each sequence of a nucleus bracket immediately followed by a closing bracket is marked. The *fill_ons* constraint treats empty onsets in the same way. Finally, the *have_ons* constraint is somewhat more complex. The constraint requires that each nucleus is preceded by an onset. This is achieved by marking all nuclei first, and then removing those marks where in fact an onset is present.

This completes the building blocks we need for an implementation of Prince and Smolensky's analysis of syllabification. In the following sections, we present two alternative implementations which employ these building blocks. First, we discuss the approach of (Karttunen, 1998), based on the lenient composition operator. This approach uses a *counting* approach for multiple constraint violations. We will then present an alternative approach in which constraints eliminate candidates using *matching*.

9.3 The Counting Approach

In the approach of (Karttunen, 1998), a candidate set is leniently composed with the set of strings which satisfy a given constraint. Since we have defined a constraint as a transducer which marks candidate strings, we need to alter the definitions somewhat, but the resulting transducers are equivalent to the transducers produced by (Karttunen, 1998). We use the (left-associative) *optimality operator* `oo` for applying an OT constraint to a given set of candidates:[5]

```
macro(Cands oo Constraint,
          Cands
            o
  mark_violation(Constraint)
            lc
       ~ ($ @)
            o
     { @ x [], ? - @}*   ).
```

Here, the set of candidates is first composed with the transducer which marks constraint violations. We then leniently compose the resulting transducer with `~($ @)`[6], which encodes the requirement that no such marks should be contained in the string. Finally, the remaining marks (if any) are removed from the set of surviving candidates. Using the optimality operator, we can then combine *Gen* and the various constraints as in the following example (equivalent to figure 14 of (Karttunen, 1998)):

```
macro(syllabify, gen
                   oo
                have_ons
                   oo
                no_coda
```

[5]The operators 'o' and 'lc' are assumed to be left associative and have equal precedence.

[6]As explained in footnote 2, this will be coerced into an identity transducer.

```
                            oo
                         fill_nuc
                            oo
                         parse
                            oo
                         fill_ons     ).
```

As mentioned above, a candidate string can violate a constraint multiple times and candidates that violate the constraint the least number of times need to be preferred. Lenient composition cannot distinguish two candidates if the first contains one violation, and the second contains at least two violations. For example, the above `syllabify` transducer will assign three outputs to the input `bebop`:

```
O[b]N[e]X[b]X[o]X[p]
O[b]N[e]O[b]N[o]X[p]
X[b]X[e]O[b]N[o]X[p]
```

In this case, the second output should have been preferred over the other two, because the second output violates 'Parse' only once, whereas the other outputs violate 'Parse' three times. Karttunen recognizes this problem and proposes to have a sequence of constraints Parse0, Parse1, Parse2 . . . ParseN, where each ParseX constraint requires that candidates not contain more than X unparsed constituents.[7] In this case, the resulting transducer only *approximates* the OT analysis, because it turns out that for any X there are candidate strings that this transducer fails to handle correctly (assuming that there is no bound on the length of candidate strings).

Our notation is somewhat different, but equivalent to the notation used by Karttunen. Instead of a sequence of constraints Cons0 . . . ConsX, we will write `Cands oo Prec :: Cons`, which is read as: apply constraint `Cons` to the candidate set `Cands` with *precision* `Prec`, where "precision" means the predetermined bound on counting. For example, a variant of the `syllabify` constraint can be defined as:

```
macro(syllabify, gen
                            oo
                         have_ons
                            oo
                         no_coda
                            oo
                         1 :: fill_nuc
                            oo
                         8 :: parse
                            oo
                         fill_ons     ).
```

Using techniques described in 9.5, this variant can be shown to be *exact* for all strings of length ≤ 10. Note that if no precision is specified, then a precision of 0 is assumed.

[7]This construction is similar to the construction in (Frank and Satta, 1998), who used a suggestion in (Ellison, 1994).

This construct can be defined as follows (in the actual implementation the regular expression is computed dynamically based on the value of `Prec`):

```
macro(Cands oo 3 :: Constraint,
          Cands
            o
  mark_violation(Constraint)
            lc
  ~ ([($ @),($ @),($ @),($ @)])
            lc
    ~ ([($ @),($ @),($ @)])
            lc
      ~ ([($ @),($ @)])
            lc
        ~ ($ @)
            o
     { @ : [], ? - @}*  ).
```

9.4 The Matching Approach

9.4.1 Introduction

In order to illustrate the alternative approach, based on matching we return to the `bebop` example given earlier, repeated here:

```
c1:   O[ b ] N[ e ] X[ b ] X[ o ] X[ p ]
c2:   O[ b ] N[ e ] O[ b ] N[ o ] X[ p ]
c3:   X[ b ] X[ e ] O[ b ] N[ o ] X[ p ]
```

Here an instance of 'X[' is a constraint violation, so c2 is the best candidate. By counting, one can see that c2 has one violation, while c1 and c3 each have 3. By matching, one can see that all candidates have a violation in position 13, but c1 and c3 also have violations in positions not corresponding to violations in c2. As long the positions of violations line up in this manner, it is possible to construct a finite state filter to rule out candidates with a non-minimal number of violations. The filter will take the set of candidates, and subtract from that set all strings that are similar, except that they contain additional constraint violations.

Given the approach of marking up constraint violations introduced earlier, it is possible to construct such a matching filter. Consider again the 'bebop' example. If the violations are marked, the candidates of interest are:

```
O[   b ] N[   e ] X[ @ b ] X[ @ o ] X[ @ p ]
O[   b ] N[   e ] O[   b ] N[   o ] X[ @ p ]
X[ @ b ] X[ @ e ] O[   b ] N[   o ] X[ @ p ]
```

For the filter, we want to compare alternative mark-ups for the same input string. Any other differences between the candidates can be ignored. So the first step in constructing the filter is to eliminate everything except the markers and the

```
macro(add_violation,
      {(bracket x []), ? - bracket}*    % delete brackets
                     o
           [[? *,([] x @)]+, ? *]       % add at least one @
                     o
      {([] x bracket), ? - bracket}*    % reinsert brackets
     ).
```

Figure 9.4: Macro to introduce additional constraint violation marks.

original input. For the syllable structure example, finding the original input is easy since it never gets changed. For the “bebop” example, the filter first constructs:

```
   b   e @ b @ o   @   p
   b   e   b   o   @   p
 @ b @ e   b   o   @   p
```

Since we want to rule out candidates with at least one more constraint violation than necessary, we apply a transducer to this set which inserts at least one more marker. This will yield an infinite set of bad candidates each of which has at least two markers and with one of the markers coming directly before the final ‘p’.

In order to use this set of bad candidates as a filter, brackets have to be reinserted. But since the filter does not care about positions of brackets, these can be inserted randomly. The result is the set of all strings with at least two markers, one of the markers coming directly before the final ‘p’, and arbitrary brackets anywhere. This set includes the two candidates c1 and c3 above. Therefore, after applying this filter only the optimal candidate survives. The three steps of deleting brackets, adding extra markers and randomly reinserting brackets are encoded in the add_violation macro given in 9.4.

The application of an OT constraint can now be defined as follows, using an alternative definition of the optimality operator:

```
macro(Cands oo Constraint,
         Cands
           o
   mark_violation(Constraint)
           o
     ~ range(Cands
              o
         mark_violation(Constraint)
              o
          add_violation)
           o
      {(@ x []),(? -  @)}* ).
```

Note that this simple approach only works in cases where constraint violations line up neatly. It turns out that for the syllabification example discussed earlier that this is the case. Using the `syllabify` macro given above with this matching implementation of the optimality operator produces a transducer of only 22 states, and can be shown to be exact for all inputs!

9.4.2 Permutation

In the general case, however, constraint violations need not line up. For example, if the order of constraints is somewhat rearranged as in:

```
parse oo fill_ons oo have_ons
    oo fill_nuc oo no_coda
```

the matching approach is not exact: it will produce wrong results for an input such as ‘arts’:

```
N[a]D[r]O[t]N[]D[s]      %cf: art@s
N[a]O[r]N[]D[t]O[s]N[]   %cf: ar@ts@
```

Here, the second output should not be produced because it contains one more violation of the `fill_nuc` constraint. In such cases, a limited amount of permutation can be used in the filter to make the marker symbols line up. The `add_violation` filter of 9.4 can be extended with the following transducer which permutes marker symbols:

```
macro(permute_marker,
 [{[? *,(@ x []),? *,([] x @)],
   [? *,([] x @),? *,(@ x [])]}*,? *]).
```

Greater degrees of permutation can be achieved by composing `permute_marker` several times. For example:[8]

```
macro(add_violation(3),
 {(bracket x []), (? - bracket)}*
                o
      [[? *,([] x @)]+, ? *]
                o
        permute_marker
                o
        permute_marker
                o
        permute_marker
                o
 {([] x bracket), (? - bracket)}*  ).
```

So we can incorporate a notion of ‘precision’ in the definition of the optimality operator for the matching approach as well, by defining:

[8]An alternative approach would be to compose the `permute_marker` transducers before inserting extra markers. Our tests, however, show this alternative to be somewhat less efficient.

```
macro(Cands oo Prec :: Constraint),
        Cands
          o
 mark_violation(Constraint)
          o
   ~  range(Cands
             o
       mark_violation(Constraint)
             o
        add_violation(Prec))
          o
   { (@ x []),(? - @)}*  ).
```

The use of permutation is most effective when constraint violations in alternative candidates tend to occur in corresponding positions. In the worst case, none of the violations may line up. Suppose that for some constraint, the input "bebop" is marked up as:

```
c1:   @ b @ e    b    o    p
c2:     b   e  @ b  @ o  @ p
```

In this case, the precision needs to be two in order for the markers in c1 to line up with markers in c2. Similarly, the counting approach also needs a precision of two in order to count the two markers in c1 and prefer this over the greater than two markers in c2. The general pattern is that any constraint that can be treated exactly with counting precision N, can also be handled by matching with precision less than or equal to N. In the other direction, however, there are constraints, such as those in the Prince and Smolensky syllabification problem, that can only be exactly implemented by the matching approach.

For each of the constraint orderings discussed by Prince and Smolensky, it turns out that at most a single step of permutation (i.e. a precision of 1) is required for an exact implementation. We conclude that this OT analysis of syllabification is regular. This improves upon the result of (Karttunen, 1998). Moreover, the resulting transducers are typically much smaller too. In 9.5 we present a number of experiments which provide evidence for this observation.

9.4.3 Discussion

Containment. It might be objected that the Prince and Smolensky syllable structure example is a particularly simple *containment theory* analysis and that other varieties of OT such as *correspondence theory* (McCarthy and Prince, 1995) are beyond the scope of matching.[9] Indeed we have relied on the fact that *Gen* only adds brackets and does not add or delete anything from the set of input symbols. The filter that we construct needs to compare candidates with alternative candidates generated *from the same input*.

If *Gen* is allowed to change the input then a way must be found to remember the original input. Correspondence theory is beyond the scope of this paper, however a

[9](Kager, 1999) compares containment theory and correspondence theory for the syllable structure example.

simple example of an OT where *Gen* modifies the input is provided by the problem described in 9.2.2 (from (Frank and Satta, 1998)). Suppose we modify *Gen* here so that its output includes a representation of the original input. One way to do this would be to adopt the convention that input symbols are marked with a following 0 and output symbols are marked with a following 1. With this convention *Gen* becomes:

```
macro(gen,
 {[(a x [a,0,b,1])*,(b x [b,0,a,1])*],
  [(a x [a,0,a,1])*,(b x [b,0,b,1])*]})
```

Then the constraint against the symbol `a` needs to be recast as a constraint against `[a,1]`.[10] And, whereas above *add_violation* was previously written to ignore brackets, for this case it will need to ignore output symbols (marked with a 1). This approach is easily implementable and with sufficient use of permutation, an approximation can be achieved for any predetermined bound on input length.

Locality. In discussing the impact of their result, (Frank and Satta, 1998) suggest that the OT formal system is too rich in generative capacity. They suggest a *shift in the type of optimization carried out in OT, from global optimization over arbitrarily large representations to local optimization over structural domains of bounded complexity.* The approach of matching constraint violations proposed here is based on the assumption that constraint violations can indeed be compared *locally.*

However, if *locality* is crucial then one might wonder why we extended the local matching approach with global permutation steps. Our motivation for the use of global permutation is the observation that it ensures the matching approach is strictly more powerful than the counting approach. A weaker, and perhaps more interesting, treatment is obtained if locality is enforced in these permutation steps as well. For example, such a weaker variant is obtained if the following definition of `permute_marker` is used:

```
macro(permute_marker,   % local variant
 {? ,[([] x @),?,(@ x [])],
     [(@ x []),?,([] x @)]}* ).
```

This is a weaker notion of permutation than the definition given earlier. Interestingly, using this definition resulted in equivalent transducers for all of the syllabification examples given in this paper. In the general case, however, matching with local permutation is less powerful.

Consider the following artificial example. In this example, inputs of interest are strings over the alphabet {`b`,`c`}. *Gen* introduces an `a` *before* a sequence of `b`'s, or two `a`'s *after* a sequence of `b`'s. *Gen* is given as an automaton in 9.5. There is

[10]OT makes a fundamental distinction between *markedness* constraints (referring only to the surface) and *faithfulness* constraints (referring to both surface and underlying form). With this mark-up convention, faithfulness constraints might be allowed to refer to both symbols marked with 0 and symbols marked with 1. But note that the `Fill` and `Parse` constraints in syllabification are also considered to be faithfulness constraints since they correspond to epenthesis and deletion respectively.

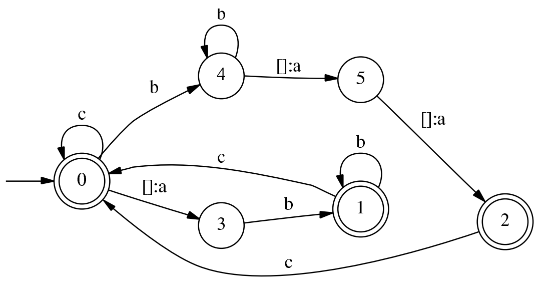


Figure 9.5: *Gen* for an example for which local permutation is not sufficient.

only a single constraint, which forbids a. It can easily be verified that a matching approach with global permutation using a precision of 1 exactly implements this OT. In contrast, both the counting approach as well as a matching approach based on local permutation can only approximate this OT.[11]

9.5 Comparison

In this section we compare the two alternative approaches with respect to accuracy and the number of states of the resulting transducers. We distinguish between *exact* and *approximating* implementations. An implementation is exact if it produces the right result for all possible inputs.

Assume we have a transducer T which correctly implements an OT analysis, except that it perhaps fails to distinguish between different numbers of constraint violations for one or more relevant constraints. We can decide whether this T is exact as follows. T is exact if and only if T is exact with respect to each of the relevant constraints, i.e., for each constraint, T distinguishes between different numbers of constraint violations. In order to check whether T is exact in this sense for constraint C we create the transducer is_exact(T,C):

```
macro(is_exact(T,C),
          T
          o
     mark_violation(C)
          o
   {(? - @) x [], @}*).
```

If there are inputs for which this transducer produces multiple outputs, then we know that T is not exact for C; otherwise T *is* exact for C. This reduces to the question of whether is_exact(T,C) is ambiguous. The question of whether a given transducer is ambiguous is shown to be decidable in (Blattner and Head, 1977); and an efficient algorithm is proposed in (Roche and Schabes, 1997b).[12] Therefore,

[11]Matching with local permutation is not strictly more powerful than counting. For an example, change *Gen* in this example to: {[([] x a),{b,c}*],[{b,c}*,([] x [a,a])]}. This can be exactly implemented by counting with a precision of one. Matching with local permutation, however, cannot exactly implement this case, since markers would need to be permuted across unbounded sequences.

[12]We have adapted the algorithm proposed in (Roche and Schabes, 1997b) since it fails to treat certain types of transducer correctly; we intend to provide details somewhere else.

in order to check a given transducer `T` for exactness, it must be the case that for each of the constraints `C`, `is_exact(T,C)` is nonambiguous.

If a transducer `T` is not exact, we characterize the quality of the approximation by considering the maximum length of input strings for which `T` is exact. For example, even though `T` fails the exactness check, it might be the case that

```
    [? ^,? ^,? ^,? ^,? ^]
           o
           T
```

in fact is exact, indicating that `T` produces the correct result for all inputs of length ≤ 5.

Suppose we are given the sequence of constraints:

```
 have_ons >> fill_ons >> parse
    >> fill_nuc >> no_coda
```

and suppose furthermore that we require that the implementation, using the counting approach, must be exact for all strings of length ≤ 10. How can we determine the level of precision for each of the constraints? A simple algorithm (which does not necessarily produce the smallest transducer) proceeds as follows. Firstly, we determine the precision of the first, most important, constraint by checking exactness for the transducer

```
gen oo P :: have_ons
```

for increasing values for `P`. As soon as we find the minimal `P` for which the exactness check succeeds (in this case for P=0), we continue by determining the precision required for the next constraint by finding the minimal value of P in:

```
gen oo 0 :: have_ons oo P :: fill_ons
```

We continue in this way until we have determined precision values for each of the constraints. In this case we obtain a transducer with 8269 states implementing:

```
gen oo 0 :: have_ons
    oo 1 :: fill_ons
    oo 8 :: parse
    oo 5 :: fill_nuc
    oo 4 :: no_coda
```

In contrast, using matching an exact implementation is obtained using a precision of 1 for the `fill_nuc` constraint; all other constraints have a precision of 0. This transducer contains only 28 states.

The assumption in OT is that each of the constraints is universal, whereas the constraint *order* differs from language to language. Prince and Smolensky identify nine interestingly different constraint orderings. These nine "languages" are presented in table 9.2.

In table 9.3 we compare the size of the resulting automata for the matching approach, as well as for the counting approach, for three different variants which are created in order to guarantee exactness for strings of length ≤ 5, ≤ 10 and ≤ 15 respectively.

Finally, the construction of the transducer using the matching approach is typically much faster as well. In table 9.4 some comparisons are summarized.

id	constraint order
1	`have_ons` $\gg$ `fill_ons` $\gg$ `no_coda` $\gg$ `fill_nuc` $\gg$ `parse`
2	`have_ons` $\gg$ `no_coda` $\gg$ `fill_nuc` $\gg$ `parse` $\gg$ `fill_ons`
3	`no_coda` $\gg$ `fill_nuc` $\gg$ `parse` $\gg$ `fill_ons` $\gg$ `have_ons`
4	`have_ons` $\gg$ `fill_ons` $\gg$ `no_coda` $\gg$ `parse` $\gg$ `fill_nuc`
5	`have_ons` $\gg$ `no_coda` $\gg$ `parse` $\gg$ `fill_nuc` $\gg$ `fill_ons`
6	`no_coda` $\gg$ `parse` $\gg$ `fill_nuc` $\gg$ `fill_ons` $\gg$ `have_ons`
7	`have_ons` $\gg$ `fill_ons` $\gg$ `parse` $\gg$ `fill_nuc` $\gg$ `no_coda`
8	`have_ons` $\gg$ `parse` $\gg$ `fill_ons` $\gg$ `fill_nuc` $\gg$ `no_coda`
9	`parse` $\gg$ `fill_ons` $\gg$ `have_ons` $\gg$ `fill_nuc` $\gg$ `no_coda`

Table 9.2: Nine different constraint orderings for syllabification, as given in Prince and Smolensky, chapter 6.

Method	Exactness	Constraint order								
		1	2	3	4	5	6	7	8	9
matching	exact	29	22	20	17	10	8	28	23	20
counting	≤ 5	95	220	422	167	10	240	1169	2900	4567
counting	≤ 10	280	470	1667	342	10	420	8269	13247	16777
counting	≤ 15	465	720	3812	517	10	600	22634	43820	50502

Table 9.3: Comparison of the matching approach and the counting approach for various levels of exactness. The numbers indicate the number of states of the resulting transducer.

Method	Exactness	Constraint order								
		1	2	3	4	5	6	7	8	9
matching	exact	1.0	0.9	0.9	0.9	0.8	0.7	1.5	1.3	1.1
counting	≤ 5	0.9	1.7	4.8	1.6	0.5	1.9	10.6	18.0	30.8
counting	≤ 10	2.8	4.7	28.6	4.0	0.5	4.2	83.2	112.7	160.7
counting	≤ 15	6.8	10.1	99.9	8.6	0.5	8.2	336.1	569.1	757.2

Table 9.4: Comparison of the matching approach and the counting approach for various levels of exactness. The numbers indicate the CPU-time in seconds required to construct the transducer.

9.6 Conclusion

We have presented a new approach for implementing OT which is based on matching rather than the counting approach of (Karttunen, 1998). The matching approach shares the advantages of the counting approach in that it uses the finite state calculus and avoids off-line sorting and counting of constraint violations. We have shown that the matching approach is superior in that analyses that can only be approximated by counting can be exactly implemented by matching. Moreover, the size of the resulting transducers is significantly smaller.

We have shown that the matching approach along with global permutation provides a powerful technique technique for minimizing constraint violations. Although we have only applied this approach to permutations of the Prince & Smolensky syllabification analysis, we speculate that the approach (even with local permutation) will also yield exact implementations for most other OT phonological analyses. Further investigation is needed here, particularly with recent versions of OT such as correspondence theory. Another line of further research will be the proper integration of finite state OT with non-OT phonological rules as discussed, for example, in papers collected in (Hermans and van Oostendorp, 1999) .

Finally, we intend also to investigate the application of our approach to syntax. (Karttunen, 1998) suggests that the Constraint Grammar approach of (Karlsson et al., 1995) could be implemented using lenient composition. If this is the case, it could most probably be implemented more precisely using the matching approach. Recently, (Oflazer, 1999) has presented an implementation of Dependency syntax which also uses lenient composition with the counting approach. The alternative of using a matching approach here should be investigated.

Chapter 10

Finite State Methods for Hyphenation

This chapter has benefitted from comments from reviewers and participants of the Finite State Methods for Natural Language Processing 2001 workshop during ESSLLI XII in Helsinki. Following suggestions by Lauri Karttunen en Theo Jansen, the method for compiling TEX patterns into a FST *was implemented partially during the workshop.*

10.1 Introduction

Hyphenation is the task of identifying potential hyphenation points in words. A typesetting program uses this information to produce justified paragraphs. In this chapter, three finite-state hyphenation methods for Dutch are presented and compared in terms of accuracy and size of the resulting automata.

Hyphenation can be done using a word list, using patterns, or using rules. The first approach requires a word list with hyphenation points, usually derived from an electronic dictionary. The major drawback of this method is that it will only cover words explicitly listed in the dictionary.

Pattern-based methods specify hyphenation patterns, where a pattern typically consists of a sequence of two or more characters, for which valid (or invalid) hyphenation positions are specified. The pattern can be applied to all words containing the pattern as a substring. A well-known pattern-based hyphenation-method, described in more detail below, is that of Liang (1983), which is used in the typesetting package TEX. Patterns are usually derived from a dictionary, but the crucial difference with a method using word lists only is that patterns represent substrings of words, and therefore also apply to words not in the word list.

Rule-based methods rely on syllable and morpheme structure to determine hyphenation points. The biggest challenge for rule-based methods is providing an accurate description of the linguistic concepts involved. The approach described in the next section uses the syllable structure to implement a hyphenation procedure. No attempt was made, however, to deal accurately with morpheme structure in derivations and compound words, and therefore the system achieves only modest accuracy. Rule-based methods use dictionary information only indirectly and are therefore not restricted to a fixed set of words.

It has been observed that rule-based and pattern-based hyphenation can be carried out by a finite-state transducer (Kaplan and Karttunen, 1998). Such a transducer would take a character string as input and return a string with hyphenation points. A transducer of this form can be constructed by translating hyphenation rules or patterns into a finite state transducer or sequence of transducers. As far as we are aware, no attempts have been made to actually develop an accurate finite-state hyphenator for a given language.

Below, we consider three finite-state methods for constructing a hyphenator for Dutch.

The core rule of Dutch hyphenation is that hyphenation points fall between syllables, where words can be divided into syllables using a maximal onset rule. The first method implements this rule as a deterministic finite state transducer. When evaluated on word list derived from the Celex lexical database (Baayen, Piepenbrock, and van Rijn, 1993), this simple system achieves a hyphen accuracy of 94.5%.

The second system builds on the results of the first system, but uses transformation-based learning (TBL) (Brill, 1995) to derive rules which correct the errors of the first system. The system is trained on the Celex word list. The induced rules typically help to correct mistakes which are due to the fact that the first system ignores morpheme structure. The hyphen accuracy after applying TBL is 99.35%, which is a significant improvement over earlier results on the same task. The rules induced by TBL can be interpreted as finite-state transducers (Roche and Schabes, 1997a), and the application of a sequence of such rules corresponds to the composition of the corresponding FST's. Constructing a single FST for the large number of rules needed for accurate hyphenation (well over 1,000) remains a challenge.

Finally, we present some results for the pattern-based hyphenation method of Liang (1983), which is used in TEX. The acquisition of patterns from a word list, which is the crucial part of the method, appears to be a special case of TBL. The hyphenation patterns for Dutch (more than 8,000) are derived from the same Celex data as used in the experiments below. As all of the Celex data was used in creating the patterns, we compared the performance of TEX and the TBL-system on running text. Hyphen accuracy of TEX on running text is estimated to be 99.8%, whereas the TBL-system achieves approximately 99.1% accuracy. Although the method of Liang (1983) uses finite-state technology to store and apply patterns efficiently, the hyphenation procedure itself is not a implemented as a FST. We present a method for compiling hyphenation-patterns into a single FST which takes a word as input and returns the word with the predicted hyphenation points.

10.2 Hyphenation rules for Dutch

Brandt Corstius (1978) gives the following procedure for hyphenating Dutch words:

Compound: First, if a word is a compound or derivation, insert hyphens between word boundaries and derivational morphemes.

Syllable: Next, insert hyphens between well-formed syllable strings,

Maximal Onset: While maximizing onsets.

The procedure can be illustrated using with the following example. The word *drugspanden* (*drug-houses*) is a plural compound noun composed of *drugs* and *panden*. Thus, the compound rule requires that a hyphenation point is inserted between *s* and *p*. Next, *panden* can be segmented into syllables as *pan-den* or *pand-en*. The ambiguity is resolved by the maximal onset rule, which states that the segmentation *pan-den* is the correct one. Thus, the correct hyphenation is *drugs-pan-den*. Note that the compound rule must identify compounds and derivational affixes, but not all morpheme structure. The word *panden* is a plural noun consisting of the noun *pand* and the plural suffix *-en*, yet it is segmented as *pan-den*.

There are a few cases which are not covered by the procedure above. In some cases the spelling of a hyphenated word differs from that of its non-hyphenated counterpart. The words *extraatje* (*extra* + dimunitive suffix *tje*) is hyphenated as *extra-tjes*. This phenomenon is known in TEX as *discretionary hyphenation* (Sojka, 1995). Another complication arises in cases where a word has an ambiguous morphological structure, and hyphenation depends on which analysis is chosen. Daelemans and van den Bosch (1992) mention *kwartslagen* as an example, which can be hyphenated as *kwart-slagen* (*quarter turns*) or *kwarts-lagen* (*quartz layers*).

Automated hyphenation of Dutch words is a well-studied problem. Brandt Corstius (1978) reports high accuracies on manually constructed word lists for his rule-based procedure. Using various machine learning techniques, Daelemans and van den Bosch (1992) and Vosse (1994) were able to achieve hyphenation accuracies of 97.8% - 98.3%. The hyphenation method of Liang (1983), which was originally applied to English, has been adapted to Dutch (Tutelaers, 1999) with very accurate results. A more detailed presentation of this method is given in section 10.6.

10.3 Syllable-based hyphenation

In this section a rule-based finite state method for hyphenating Dutch words is presented. We concentrate on the definition of a transducer which inserts hyphens between well-formed syllables, while satisfying the maximal onset rule. The *replace*-operator of Karttunen (1995) supports a straightforward definition of such a transducer. The accuracy of the system is limited, as it ignores the fact that compounds and derived words may require hyphenation patterns conflicting with the maximal onset rule. In the next section, we present a method for correcting such errors.

Finite state automata will be defined using regular expressions. Regular expressions are compiled into automata by the sc fsa utilities toolkit (van Noord, 1997b; van Noord and Gerdemann, 2001)[1] The regular expression syntax used in this chapter is defined in figure 10.1.

Consider the problem of segmenting a word into a sequence of syllables. Given a suitable definition of `syllable`, one might consider the following as a first attempt:

[1]The sc fsa utilities toolkit is available from www.let.rug.nl/ vannoord/fsa.

`[]`	the empty string
`[R1,...,Rn]`	concatenation
`{R1,...,Rn}`	disjunction
`R^`	optionality
`R*`	zero or more occurrences of R
`A:B`	the transducer which maps strings of A onto strings of B
`id(A)`	identity: the transducer which maps each element in A onto itself.
`T o U`	composition of the transducers T and U.
`macro(Term,R)`	use Term as an abbreviation for R.

Figure 10.1: FSA regular expression syntax used in this chapter. A and B are regular expressions denoting recognizers, T and U transducers, and R can be either.

(1) `[ [ syllable, []:- ]*, syllable ]`

This regular expression defines a transducer that accepts non-empty sequences of syllables as input and outputs the same sequence, with a hyphen inserted after every syllable except for the last. Given an input such as *alfabet* (*alphabet*), it will produce *al-fa-bet*, *alf-a-bet*, *alf-ab-et*, or *al-fab-et*, whereas in fact only the first is correct. This transducer is non-deterministic and does not respect the maximal onset rule. A second problem with this definition is that, in general, it will fail to give the right results for syllables containing a nucleus represented by more than a single character. I.e. for a word such as *waait* (*blows*), ((1)) would produce, among others, an output with three hyphenation points (*wa-a-it*), whereas in fact *waait* is a monosyllabic word. The problem is caused by the fact that the characters *a,a,* and *i* represent a diphtong but can also each occur independently as a nucleus, and form a syllable. Thus, it seems that apart from a maximal onset rule, there is also something like a *maximal nucleus* rule.

A more accurate definition of hyphenation needs to define a deterministic automaton, which, given multiple ways of segmenting a string into valid syllables, returns only the output in which both nuclei and onsets are maximal. Note that this requires selection between alternatives, which seems hard to implement in finite-state terms. However, Karttunen (1998) and Gerdemann and van Noord (2000) propose finite state implementations of Optimality Theory (Prince and Smolensky, 1995) which address this issue explicitly. Karttunen's finite-state analysis of the OT theory of syllabification uses a cascade of finite state transducers composed by means of an operation called *lenient composition*. The *lenient composition* of a transducer A with a recognizer B is a transducer which is similar to A, except that its only admitted outputs must be accepted by B as well. However, if no outputs exist which satisfy B, all outputs are returned (and thus no filtering occurs). This captures the effect of *violable* constraints as used in OT. Karttunen shows that his formalism correctly accounts for the semantics of constraints such as *fill onset*, a constraint which prefers onsets to be filled. *Fill onset* is similar to the notion of *maximal onset* required for Dutch hyphenation. Maximizing onsets (and nuclei) is more complicated, however, as it requires a notion of longest match as well. While it is conceivable that the effect of a preference for maximal onsets and nuclei could be defined using OT-style constraints, we opted for using an alternative finite-state technique, in which *longest match* replacements are accounted for directly.

The *replace*-operator of Kaplan and Kay (1994) and Karttunen (1995) was introduced to facilitate the implementation of (SPE-style) phonological rewrite rules as finite state transducers. In the FSA-notation of Gerdemann and van Noord (1999) a regular expression `replace(Target, LeftContext, RightContext)`, where `Target` is a transducer and `LeftContext` and `RightContext` are recognizers, defines a transducer which replaces all occurrences of the domain of `Target` between `LeftContext` and `RightContext` by strings in the range of `Target`. Furthermore, `replace` performs left-most, longest match, replacement, i.e. it operates as if moving through the string from left to right, at each point identifying the longest possible replacement target.

The regular expression for hyphenation given above can be rephrased as a replace statement, which inserts hyphens between syllables:

```
(2) replace([]:-, syllable, syllable)
```

Again, given an adequate definition of `syllable`, this transducer will insert hyphens between syllable strings. Note, however, that the left-to-right mode of application of `replace` ensures that hyphens are inserted as early as possible, thus implicitly ensuring that onsets are maximal. The word *alfabet* is hyphenated only as *al-fa-bet* by this expression, and not as any of the alternatives mentioned above.

Implementing the *maximal nucleus* rule required for the correct hyphenation of words containing multiple character nuclei, can be achieved using the longest match property of `replace` explicitly:

```
(3) replace([ []:@, identity(nucleus), []:@ ], [], [] )
```

This transducer inserts the marker '@' before and after nuclei. As the target for a replacement is determined using longest match, multiple character strings corresponding to a possible nucleus are always marked as a single nucleus. The word *waait* is marked as *w@aai@t* only and not as *w@aa@@i@t* or *w@a@@a@@i@t*. Left-to-right, longest match, identification of nuclei appears to be very accurate. One rare case where it fails is the word *dieet* (*diet*), which contains the nucleus *i* followed *ee*, but longest match recognizes *ie* and *e*.

A syllable-based hyphenation algorithm can now be given as the composition of the transducers defined in (3) and (2), given a definition of syllable which incorporates the '@'-marker:

```
(4)  replace([[]:@, identity(nucleus), []:@],[],[])
                                   o
           replace( []:-, syllable, syllable )
```

The actual implementation imposes slightly weaker conditions on the insertion of hyphens:

(5)
```
replace([[]:@, identity(nucleus), []:@],[],[])
                        o
replace( []:-, [@, consonant *], [onset^, @] )
```

Instead of specifying a full syllable as left and right context, this definition only requires a left context consisting of a marker followed by an arbitrary number of consonants (corresponding to the coda of the preceding syllable), and a right context consisting of an (optional) onset followed by a marker. It imposes no constraints on codas, other than that they must form a sequence of consonants. While this does not lead to overgeneration, it does account for the hyphenation of loan-words containing codas which do not follow Dutch spelling (i.e. the words *checklist*, *arctisch*, and *pizza* are hyphenated as *check-list*, *arc-tisch* (*arctic*), and *piz-za* in spite of the fact that *ck*, *rc* and *z* are normally not used as coda).

The accuracy of the hyphenation method defined above can be further improved by making a number of adjustments. First, in syllables such as `qua`, the *u* is part of the onset (i.e. the grapheme *qu* is pronounced as *kw*). As *ua* is not a nucleus, two nucleus markers would be inserted in this case. To prevent such mistakes, *qu* is transduced into *Q* by a preprocessing step, and transduced into *qu* again by postprocessing. Second, the character *x* (pronounced as the two consonants *ks*) is not followed (or preceded) by a hyphen in words such as *examen* in spite of the fact that *x* separates the two nuclei *e* and *a*. This can be corrected for by explicitly removing hyphens following *x* and preceding a vowel. Third, systematic exceptions to the maximal onset rule exist. The long vowels *a*, *o*, *u* are written as *aa*, *oo*, *uu* in syllables with a non-empty coda (*aap*, (*monkey*), *oor* (*ear*), uur (*hour*)), but as single characters in syllables with an empty coda (*a-pen*, *o-ren*, *u-ren*). This implies that the correct hyphenation of *haasten* (*to hurry*) is *haas-ten*, and not *haa-sten* (as the latter would result in a misspelled syllable *haa*). Note that the latter is predicted by the maximal onset rule. The solution for this problem is to include a following consonant in the definition of nucleus for the strings *aa*, *oo* and *uu*. Thus, *haasten* is first transduced into *h@aas@t@e@n* and then into *h@aas@-t@e@n*, which is the correct hyphenation after removal of the @-markers. Finally, as noted above, in some cases (such as *dieet*) the longest match strategy for marking the nucleus fails. Some of these mistakes can be corrected easily.

The actual implementation is given in figure 10.2.

Evaluation and Error-analysis

The deterministic FST for the regular expression for `hyphenation` as given in figure 10.2 has 89 states and 826 transitions.[2]

The Celex (Baayen, Piepenbrock, and van Rijn, 1993) DOW-list (*dutch orthograhpy words*) provides hyphenation patterns for over 330,000 words. We removed all items containing capitals and diacritics, and ensured that each word form occurs only once. This leaves approximately 290,000 items. The average word length is 10.85 characters and there are approximately 2.5 hyphens per word. On

[2]Transitions include predicates as described in (van Noord and Gerdemann, 2001). A predicate ranges a set of symbols, and thus, the number of transitions is in general (much) smaller than would normally have been the case.

```
macro(hyphenate,
    replace([q,u]:Q, [], [])                         % qu -> Q
  o replace([ []:@, id(nucleus),[]:@ ],[],[]) % mark nucleus
  o replace([e,@,@]:[@,@,e],i,{e,u})          % d@ie@@e@t -> d@i@@ee@t
  o replace( []:-, [@, cons *], [onset^ , @]) % insert hyphens
  o replace(-:[], x, [@,{a,e,u,i,o,y}])       % remove - after x
  o replace(Q:[q,u], [], [])                  % Q -> qu
  o replace(@:[], [], [])                     % remove markers
     ).

macro(nucleus,{ a, [a,i], [a,a,i], ..., u, [u,i], y,
                [{[a,a],[o,o],[u,u]}, cons ]
              }).

macro(onset, { b, [b,l], [b,r], ..., z, [z,w] } ).
```

Figure 10.2: Finite state syllable-based hyphenation of Dutch

this list, `hyphenate` achieves a word accurary of 86.1% and a hyphen accuracy of 94.5%.

A 10% subset of the data, containing 4,319 errors in total, was inspected for error analysis, where errors were counted and classified using the alignment method described in the next section. As the system implements the maximal onset rule, but has no way of recognizing compounds or derivational affixes, errors are expected to occur typically in situations where a word or affix boundary conflicts with the maximal onset rule. This is confirmed by the fact that 87.5% of the errors are cases where a hyphen is displaced one position to the left (i.e. *drug-span-den* should be *drugs-pan-den*) and 1.1% of the errors are cases where a hyphen is displaced two positions to the left (*ang-staan-ja-gend* (*scary*, lit. *scare-on-making*) should be *angst-aan-ja-gend*). A hyphen was displaced one position to the right in 4.9% of the errors. These are all caused by the fact that our list of onsets is not exhaustive. For instance, the system produces *ar-tis-jok* (*artichoke* which should be hyphenated as *ar-ti-sjok*) because it does not contain the onset *sj*. This type of error can be excluded if all onsets occurring in the word list are included in the system. This will have a negative effect on the overall accuracy, however, as expanding the set of onsets will in general lead to an increase in cases where a hyphen is displaced to the left as well. Including *sj*, for instance, implies that *dok-ters-jas* (*doctors-coat*) will be hyphenated as *dok-ter-sjas*. As *s* is very frequent both in the final position of a coda and in the first position of an onset, these errors will be frequent as well and will outnumber the positive effect of being able to treat (rare) *sj*-onsets correctly. In 5.5% of the errors the system has produced a spurious hyphen. These typically occur in loan words containing a nucleus not included in the syllable-based system (*coach* is hyphenated as *co-ach*) . The system missed a hyphen in 1.0% of the errors. Such errors can for instance be caused by mistakes in the identification of a nucleus. The word *be-ij-ver* (*work towards*) is hyphenated as *beij-ver* because *ei* is a valid nucleus. This is a case, therefore, where the longest match strategy for recognizing the nucleus fails and which was left unaccounted for in the definition of `hyphenate`.

10.4 Improving accuracy using TBL

Transformation-based learning (TBL) (Brill, 1995) can be used to improve the accuracy of the system outlined above. Given a word list hyphenated by the base system, aligned with the correct hyphenation patterns, TBL will attempt to induce a rule which corrects the maximal number of errors while introducing a minimum of new errors. This rule is applied to the training data. This process iterates until no new rules with a score (i.e. the number of corrections minus the number of errors introduced by a rule) above a certain threshold can be found. The fact that most rules are not 100% accurate (i.e. introduce new errors besides correcting existing errors) is not necessarily a problem, as more specific rules can be learned later which correct the newly introduced errors. The result of TBL can be tested on a data-set by applying the induced rules in the order in which they have been induced.

10.4.1 Alignment

TBL requires aligned data for training and testing. Hyphenation can be seen as a classification task, which decides, for instance, for each character whether it is preceded by a hyphen or not. Thus, the result of hyphenating a word list using the base system can be encoded as a character string, where each character is aligned with a 1 if it is preceded by a hyphen, and with a 0 otherwise. Furthermore, as some rules should apply only to the beginning or end of a word, boundary markers are added. The correct hyphenation can be aligned with this string as a similar list of 0's and 1's:

(6)

word	*potato*	+	a	a	r	d	a	p	p	e	l	+
system	aar-dap-pel	0	0	0	0	1	0	0	1	0	0	0
correct	aard-ap-pel	0	0	0	0	0	1	0	1	0	0	0

For TBL, the encoding in (6) has the disadvantage that correcting a single error (i.e. *aar-dap-pel* $\rightarrow$ *aard-ap-pel*) requires learning two error-correction rules, one changing a 1 into a 0 when aligned with a *d* in a suitable context, and one changing a 0 into a 1 when aligned with an *a* in a similar context. Obviously, these two rules are closely related. Therefore, more effective error correction can take place if a single rule could be learned to correct the error.

We therefore adopted a slightly more involved alignment procedure, where the correct output can be marked with 0 and 1 as before, but also with 2 (the character is followed by a hyphen), 3 (the next character is followed by a hyphen) or 9 (the preceding character was preceded by a hyphen). Examples of the new alignment method are given in figure 10.3.

Note that the system output is still given as a sequence of 0's and 1's, and thus, that the aligned system output does not contain information which points to the correct alignment. Alignment of the correct answer with the system data requires a procedure which codes the correct answer relative to the system output. As the

word	(*potato*)	+ a a r d a p p e l +
system	aar-dap-pel	0 0 0 0 1 0 0 1 0 0 0
correct	aard-ap-pel	0 0 0 0 2 0 0 1 0 0 0
word	(*coach*)	+ c o a c h +
system	co-ach	0 0 0 1 0 0 0
correct	coach	0 0 0 0 0 0 0
word	*artichoke*)	+ a r t i s j o k +
system	ar-tis-jok	0 0 0 1 0 0 1 0 0 0
correct	ar-ti-sjok	0 0 0 1 0 0 9 0 0 0
word	(*scary*)	+ a n g s t a a n j a g e n d +
system	ang-staan-ja-gend	0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0
correct	angst-aan-ja-gend	0 0 0 0 3 0 0 0 0 1 0 1 0 0 0 0
wrd	(*with make up on*)	+ g e s c h m i n k t +
sys	gesch-minkt	0 0 0 0 0 0 1 0 0 0 0 0
cor	ge-schminkt	0 0 0 1 0 0 0 0 0 0 0 0

Figure 10.3: Improved alignment

number of hyphens in the system output and the correct output tends to be equal (but see below), a procedure which determines for each correct hyphen where it is located relative to the corresponding system hyphen gives good results. If no corresponding system hyphen can be found in the two proceeding positions or in the next position, a 0/1 alignment is introduced. Similarly, if a system hyphen cannot be aligned with any hyphen in the correct answer, a 1/0 alignment is introduced.

10.4.2 Experiments

Training and test data consisted of the Celex word list described in the previous section. The list was divided into 10 sections (selecting every 10th word, with an offset of 0-9), hyphenated by the base system, and the result was aligned with the correct hyphenation as provided by Celex.

For training and testing we used the fnTBL toolkit[3] (Ngai and Florian, 2001) which implements an efficient version of Brill's original algorithm. Training took between a few minutes and 3 hours, depending on the amount of data used and the complexity of the rule templates.

Rule templates for TBL were provided which change the value of one cell in the system output (i.e. change 1 into 2), using a surrounding window of maximally 5 characters as context to constrain the rule. For instance, to correct *aar-dap-pel* into *aard-ap-pel*, the system might learn the following rule:

(7)
$$\begin{array}{ccccc} \hline a & a & r & d & a \\ & & & 1 & \\ & & & \downarrow & \\ & & & 2 & \\ \hline \end{array}$$

[3]http://nlp.cs.jhu.edu/ rflorian/fntbl/index.html

Rank	Good	Bad	Score	Rule
1	2918	314	2604	i-st → is-t
2	1890	103	1787	-?achti → ?-achti
3	1666	101	1565	a-st → as-t
4	1411	135	1276	ing-s → ings-
5	714	37	677	u-st → us-t
6	693	31	662	+ve-r → +ver-
7	992	336	656	-th → t-h
8	1333	703	630	e-ste → es-te
9	635	36	599	+a-f → +af-
10	596	27	569	+o-n → +on-
16	320	26	294	bes-t → be-st
27	191	3	188	ges- → ge-s
1408	3	0	3	-??eid → ?-?eid

Figure 10.4: Rules learned by TBL, trained on 90% of the data. '?' represents an arbitrary character, '+' is the word boundary symbol.

	initial	10%	20%	30%	60%	90%
Number of Induced Rules		264	507	737	1,139	1,409
Hyphen Accuracy	94.16	98.15	98.60	98.82	99.04	99.27
Word Accuracy	85.30	95.34	96.49	97.02	97.59	98.17

Figure 10.5: Results of learning hyphenation rules using 10%-90% of the data (29.000 - 260.000 words).

An overview of some of the rules learned using 90% of the data for training is given in figure 10.4. Half of the rules in the top-10 (1, 3, 4, 5, 8) illustrate that the character '*s*' is problematic for hyphenation, as it can be the start of many different onsets, but also can be the final character in many codas. The second rule correctly hyphenates words with the suffix-morpheme *-achtig*, which counts as introducing a boundary for the compound rule. The majority of induced rules are of the 1→2 type. Rule 16 is the highest-ranked rule which makes a different correction. It is of the 2→1 type and corrects the effect of an earlier rule (rule 8 in particular). Rule 27 is the second rule which shifts a hyphen leftward. It recognizes the prefix *ge-*. Note that according to the maximal onset rule, the hyphen would have been placed in front of the *s* to begin with. Thus, this rule also corrects some of the errors introduced by rule 8 (and possibly by other preceding rules).

We performed experiments on various portions of the Celex data. The results are given in figure 10.5. When trained on only 10% of the available data, TBL learns a relatively small number of highly effective rules. The error rate can be further reduced by using 90% of the data, although the number of induced rules also increases substantially in that case.

Discussion

The results improve considerably on previous work. The best result of Daelemans and van den Bosch (1992) (98.3% hyphen accuracy) used an exemplar-based learning method, trained on a word list of 19,000 words and using a fixed context window of 7 (i.e. the window only refers to the target character and the three preceding and following characters). Vosse (1994) trains a Hidden-Markov Model and a pattern-based hyphenator similar to the system of Liang (1983) on a word list of 190,000 words and achieves 97.8% and 98.2% hyphen accuracy for the Markov Model and pattern-based system respectively.

The results given in figure 10.5 can be improved upon only slightly by using more context. Using 90% of the data for training, a context of 7 gives a hyphen accuracy of 99.35% (using 1,484 rules) and including information about the absence or presence of other hyphens in the context gives an accuracy of 99.32% (using 1,404 rules).

Belz (2000) observes that the distribution of syllables is not equal for all positions within a word. That is, initial syllables will relatively often consist of a derivational prefix, while final syllables will relatively often be inflection endings or derivational suffixes. This suggests that learning can profit from a setup which considers separately the first and second syllable string (and thus only the first hyphen), the penultimate and last syllable (and thus only the last hyphen), and the intermediate syllables and hyphens. We performed experiments where we tried to learn hyphenation rules specifically for these positions. The experiments showed that there is considerable variation in the diffuclty of placing the first, final, and intermediate hyphens. Accuracies of 99.6% (using 185 rules) were obtained for the final hyphen, 98.6% (748 rules) for the initial hyphen and 99.3% (446 rules) for the intermediate hyphens. The high accuracy for the final hyphen is as expected, given the large number of suffixes in this position. The low accuracy of the initial hyphen (lower than that of the intermediate hyphens) is unexpected, however. At the moment, we have no explanation for this fact. The weighted average of the accuracies for initial, intermediate, and final hyphens is 99.1%. This suggests that training on specific subproblems does not by itself lead to higher overall accuracy.

10.5 Compilation of TBL rules to a FST

TBL rules can be interpreted as finite state transducers (Roche and Schabes, 1997a). A single rule corresponds to a transducer which interchanges characters and hyphens or, alternatively, changes digits. Rules learned by TBL are applied to the data in the order in which they are learned. A sequence of TBL rules therefore corresponds to the composition of the individual rule transducers.

The syllable-based hyphenation system transduces an input string into a hyphenated string. TBL rules can be interpreted as rules which correct errors of the base-system by shifting inserting or deleting hyphens in specific contexts. Recall that, given the alignment-method used for TBL, changing a '1' into a '2' means that a hyphen has to follow rather than precede the corresponding character. Thus, the first rule induced by TBL corresponds to the following regular expression:

(8) `replace([-,s]:[s,-], [i], [])`

The base system hyphenates *communisme* (*comunism*) as *com-mu-ni-sme*. The regular expression above corrects this to *com-mu-nis-me*. Similar regular expressions can be given for rules which shift a hyphen two positions rightwards, which shift a hyphen leftward, or which insert or delete a hyphen.

A second method for interpreting the TBL-rules makes more direct use of the alignment-method used for TBL. The output of the syllable-based system can be transduced easily into a string where each character is preceded by a '0' or a '1', indicating the absence or presence of a hyphen in that position. Thus, *com-mu-ni-sme* would be represented as (9-a). TBL-rules can now be interpreted as regular expressions for replacing a single digit. Thus, the first rule learned by TBL would now correspond to the regular expression in (9-b).

(9) a. 0c0o0m1m0u1n0i1s0m0e
b. `replace(1:2, [i], [s])`

The output of a cascade of such rules is a character string interspersed with digits. The corresponding hyphenated string is obtained by a transducer which deletes `0`, translates a `1` into a hyphen, `[2,C]` into `[C,-]` (for any character `C`), etc.

A finite state transducer implementing the base system and the result of TBL can now be conceptualized as follows:

(10)
```
      hyphenate
          o
  introduce_digits
          o
 apply_rule_cascade
          o
  interpret_digits
```

The advantage of the second method is that TBL-rules correspond to one character substitutions, whereas the first method introduces more complicated replacement-targets. In practice, we observed that the second method is also less computationally demanding (both in terms of memory required during compilation and in terms of the size of the resulting automaton).

A disadvantage of the second method is that it requires that contexts must refer both to characters and digits. That is, the fourth rule learned by TBL does not correspond to (11-b) but to (11-c), where `digit` is the disjunction defined in (11-a).

(11) a. `macro(digit, {0, 1 ,2, 3, 9})`
b. `replace(1:2, [i,n,g], [s])`
c. `replace(1:2, [i,digit,n,digit,g], [s])`

Rule contexts therefore become very large (i.e. a 5 character context gives rise to a regular expression with a context of length 10). In practice, compilation of such rules is difficult. We therefore also experimented with a rule templates which allowed digits to be included in the rules. The effect of this is that instead of having to insert `digit` in contexts, we now obtain rules where the value of each digit in the

Rules	no digits		digits		combined	
	S	T	S	T	S	T
25	137	1155	112	704	48	832
50	355	4159	261	2184	115	3163
75	589	6883	460	4229	180	5891
100	783	9196	611	5628	219	7673
150	1,152	13,867	888	8,583	340	12,706
200	1,846	24,019	1,206	11,748	486	20,388
250	2,563	33,187	1,524	15,543	642	27,860

Figure 10.6: Number of states S and transitions T for the FST consisting of the composition of N TBL rules using various methods.

context is known (usually, 0). For instance, instead of learning the rule in (11-c), the system now learns:

(12) `replace(1:2, [i,0,n,0,g], [s])`

A second alternative we expermimented with is treating pairs of digits and characters as a single symbol. This has the advantage that contexts are reduced even further, but has the disadvantage that the alphabet increases to approximately 5 x 26 (i.e. all combinations of a digit used in the alignment and a character).

Some results for composing N FST's for individual TBL-rules into a single transducer for the initial system, the system with digits in contexts, and the system with combined characters, are given in figure 10.6. Although the size of the transducer grows approximately linear with the number of incorporated rules, the overall size of the transducer is nevertheless too large for incorporation of all 1,400 rules induced by TBL. Using the Prolog-based FSA implementation on a 64-bit machine with 1 Gb of memory, we managed to compile maximally 400 rules into a single transducer. To apply the full set of induced rules to new data, the best one can do therefore is compose FST's for up to 400 rules, and use a pipeline architecture to pass the output of one transducer as input to the next.

These results are less encouraging than those of Roche and Schabes (1997a). Note, however, that the rule-set they experimented with were the result of applying TBL for part-of-speech tagging (as in Brill (1995)). Their rule set consisted of 280 rules with a context of at most three symbols. If the rule set is much larger, and contains lengthy contexts, compilation may not be feasible in practice.

10.6 A comparison with TeX

The pattern-based hyphenation method of Liang (1983), incorporated in TeX and related programs such as LaTeX, uses a word list to derive patterns which indicate legal and illegal hyphenation points. The extraction of patterns is very similar to TBL in that it tries to find the patterns which introduce most legal (or illegal) hyphenation points, while introducing a minimal number of errors. In this section, we present two results which shed light on the relationship between the hyphenation

Level	Patterns	String
(mark boundaries)		.pijnappel.
1	jn$_1$a $_1$na	.pij$_1$n$_1$appel.
2	i$_2$j j$_2$na $_2$nap	.pi$_2$j$_2$n$_1$appel.
3	$_3$pijn p$_3$p	.$_3$pi$_2$j$_2$n$_1$ap$_3$pel.
4	.p$_4$ jna$_4$ $_4$pp pe$_4$l $_4$l.	.$_3$p$_4$i$_2$j$_2$n$_1$a$_4$p$_3$pe$_4$l.
5	p$_5$p	.$_3$p$_4$i$_2$j$_2$n$_1$a$_4$p$_5$pe$_4$l.
(interpret result)		pijn-ap-pel

Figure 10.7: Pattern-based Hyphenation in TeX

methods presented above and the pattern-based method. We evaluate the accuracy of TeX and the TBL system for Dutch on running text. Next, we present a method for compiling hyphenation patterns into a single FST. We start with an overview of the pattern-based method.

10.6.1 Hyphenation in TeX

Hyphenation in TeX uses five levels of patterns to determine legal and illegal hyphenation points. Level 1 contains patterns which insert the digit 1, level 2 patterns introduce 2, etc. In the resulting string, odd numbers stand for potential hyphenation points, while even numbers indicate illegal hyphenation points. Tutelaers (1999) illustrates the method with the example in figure 10.7. First, word boundary markers (.) are added to the word to be hyphenated. Next, level 1 patterns are applied. A pattern matches if it contains a character-string which matches some part of the word. Application of the pattern means that the corresponding marker is added to the word string. In this case, two patterns apply, introducing two markers. Next, level 2 patterns are applied. Higher level rules override the effects of lower rules, and thus the j$_2$na pattern overwrites the effect of the $_1$na pattern. In the final step, odd numbers are realized as hyphens, while even numbers can simply be discarded. LaTeX adopts the typographic convention of never hyphenating words after the first or before the penultimate and last character.

The acquisition of patterns follows a procedure which is strikingly similar to TBL. In a first round, patterns are collected which identify a high number of potential hyphenation points, while overgeneralizing minimally. In a second round, patterns are learned which block the erroneuos hyphenation points which are the result of applying the level-1 rules. This process iterates three more times, giving rise to five levels of rules in total.

There are also a few differences between this method and TBL. First of all, all patterns on a given level can be applied simultaneously to an input string. Ordering of patterns is only relevant between levels. TBL rules, on the other hand, should be applied in the order in which the are acquired. Second, the score of each rule must be above a certain threshold T, where the score of a pattern is computed as follows:

(13) $\mathit{Score} = \mathit{Good} \times G - \mathit{Bad} \times B$

Good and *Bad* are the counts for the number of correct and wrong applications of the pattern to the data, G and B are weights which determine the accuracy of the rule (i.e. by making B much higher than G, accurate rules will score higher than less accurate rules which might correct a larger number of errors). The value of G, B and the threshold T may vary per level. Furthermore, the maximal length of the patterns considered may vary per level. A typical set-up appears to be one where the context is short and the threshold is high initially, while for higher levels, the context is longer, the threshold is lower, and the value of B is much higher than that of G. For TBL, there is no fixed number of levels, and parameters can only be set globally. Finally, Liang's method learns both legal and illegal hyphenation points. The TBL method combines the effect of identifying legal and illegal hyphenation points by learning rules which shift a hyphen. Shifting a hyphen implicitly classifies the original position as an illegal hyphenation point and the target position as a legal hyphenation point.

Hyphenation patterns for Dutch were created on the basis of the same Celex word list used for training and testing in the previous sections. As a consequence of a spelling reform in 1996[4] the construction of patterns has been redone recently (Tutelaers, 1999). The new Dutch pattern file contains a total of 8.870 patterns, where patterns are maximally 8 characters long.

10.6.2 Hyphen accuracy on running text

Training and testing on word lists for which the correct hyphenation is known, is convenient but also artificial, as the characteristics of running text differ sharply from that of a word list. The average word length for the Celex list is 10.9 while it is approximately 4.5 for Dutch running text. The average number of hyphenation points per word is 2.5 for Celex, but only 0.6 for the fragment of running text described below. Evaluation on running text may therefore give results which differ from the results for a word list. Note also that, as the TeX patterns for Dutch were derived using 100% of the Celex list, evaluation on outheld data from the word list is impossible.

A test set was created consisting of 1,000 sentences of newspaper text (selected from the CD-ROM version of *de Volkskrant*, 1997). This set contained a total of 11,641 words. Approximately 10% of the word types in the test set were not included in the Celex. 4,219 of the words were hyphenated by LaTeXwhen forced to produce all hyphenation points, giving rise to a total of 7,024 hyphens. Instead of checking all results, we only inspected those cases where LaTeXand the TBL system disagreed. This was the case for 83 words, 9 of which were typo's which were discarded. The results of the comparison are given in figure 10.8.

Both systems make very few mistakes, but TeX does considerably better than the TBL system. The difference seems to be due mostly to the fact that TeXuses well over 8,000 patterns, where the TBL system uses only 1,400 patterns. On the other hand, the TBL system only corrects the output of the syllable-based system, and thus is expected to require less rules. Both systems used the same data for acquisition of rules or patterns, but the TeX system uses rules with more context (up to 8

[4]See www.minocw.nl/spelling.

	LaTeX	TBL
Mistake	3	36
Missing hyphen	12	25
Total	15	61
Hyphen Accuracy (estimate)	99.8	99.1
Word Accuracy (estimate)	99.8	99.5
Mistakes	by-pas-s-o-pe-ra-tie	al-snog
	(*bypass-operation*	(*eventually*)
	chi-que	aut-hen-tieke
	(*chic*)	(*authentic*)
	fa-mi-lief-ront	be-de-vaart-soord
	familyfront	(*holy place*)
		be-leid-sme-de-wer-ker
(*burocrat*)		bus-i-ness
		(*business*)
Missing	ana-ly-semo-del	afle-ve-ring
	analysis model	(*sequel*)
	autofa-bri-kant	erach-ter
	(*car manufacturer*	(*behind it*)
	be-de-vaartsoord	li-chaampje
(*holy place*)	*small body*)	
	be-leidsme-de-wer-ker	onaf-han-ke-lijke
	(*burocrat*)	(*independent*)
	drugspan-den	opeens
	(*drug houses*)	(*suddenly*)

Figure 10.8: Comparing Latex and TBL on 11,641 words (containing 7,024 potential hyphenation points) of running text.

characters). Furthermore, it seems the careful tuning of parameters guiding the acquisition of patterns at each level allows the pattern-based system to acquire more effective patterns than the generic TBL method.

It is tempting to think of the results in figure 10.8 as indications of accuracy. Note, however, that, apart from the fact that the test set was small, the systems have only been compared on words were there was disagreement. It is possible therefore that the test results contain errors which have gone unnoticed because both systems made the same mistake.

10.6.3 Compilation of patterns

Although Liang (1983) uses a finite-state method to store and apply patterns efficiently, patterns are not actually compiled into a transducer which takes a word as input and produces the hyphenation according to the patterns as output. The construction of such a transducer is proposed in Kaplan and Karttunen (1998).

A hyphenating FST for hyphenation patterns can be constructed as the composition of the following sequence of transducers. First, 0's are introduced between all characters in the input string. Next, level 1 rules are applied. Applying a level 1 rule means that a 0 is replaced by a 1 in the relevant context. For instance, the pattern in (14-a) corresponds to the regular expression in (14-b). Note that contexts must be interspersed with digits up to the corresponding level of rule application. Application of all level 1 rules can be achieved by composing (in arbitrary order) all regular expressions for the individual patterns into a single transducer. Higher level rules must be able to overwrite the effect of earlier, lower level rules. This can be achieved by interpreting a level 2 pattern such as (14-c) as the regular expression in (14-d). This pattern substitutes both 0's and 1's by 2's in the relevant context. The level 1 transducer is composed with the transducer for all level 2 rules, etc. Finally, odd numbers are realized as hyphens and even number are discarded. Schematically, we have the FST defined by the regular expression in (14-e).

(14) a. jn$_1$a
b. `replace(0:1,[j,{0,1},n],[a])`
c. j$_2$na
d. `replace({0,1}:2,[j],[n,{0,1,2},a])`
e.
```
insert_digits(0)
          o
   patterns(1)
          o
   patterns(2)
          o
   patterns(3)
          o
   patterns(4)
          o
   patterns(5)
          o
 digits2hyphens
```

The construction of an FST for hyphenation patterns is not unlike the construction of an FST for TBL-rules. Yet, compilation of more than 8,000 patterns into a single transducer turned out to be feasible. The result has over 600,000 transitions, and gives rise to a binary of 15 Mb.

It is not clear why compilation of hyphenation patterns is 'easier' than compilation of TBL patterns for the same task. One possible explanation could be the fact that hyphenation patterns are ordered in blocks, where the application of rules within a block is not ordered, whereas TBL-rules have to be applied in order of acquisition. The latter suggests more interaction between rules than the former, which might have an effect on the complexity of the corresponding automaton. Strict ordering of TBL-rules is necessary because some rule may provide input for a later rule (*feeding*), or because some rule blocks the application of a later rule (*bleeding*). If no feeding or bleeding relationship exists between rules, they can be applied in any order. We computed these relationships for the hyphenation rules learned by TBL, and concluded that 19 blocks of rules (where the order of application within a block is irrelevant) are needed to account for all feeding and bleeding relationships. Thus, it seems that the interaction between rules is indeed more complex in the TBL system than in the pattern-based system.

10.7 Conclusions

In this chapter we have presented two finite-state methods for hyphenation as well as a method for compiling an existing method into a finite-state transducer. The use of the *replace*-operator was crucial in all methods. In particular, the syllable-based method capitalizes both on the fact that *replace* performs a longest-match replacement and on the fact that it preforms replacements from left to right. The method for compiling TBL-rules into a single transducer proposed by Roche and Schabes (1997a) turned out to be impractical for the large number of rules required for accurate hyphenation. Hyphenation patterns as used by TEX on the other hand, proved to be compilable into a single FST, albeit a large one. It is unclear why TBL-rules are 'harder' in this respect than patterns.

Accuracy of hyphenation after applying TBL turned out to be higher than that of previous systems for Dutch, that were trained and evaluated on word lists. The numerous hyphenation patterns for Dutch used by TEX are even more accurate. The acquisition of TBL-rules and hyphenation patterns is strikingly similar. By using larger context windows for patterns, and by tuning the acquisition of patterns carefully to the hyphenation problem, TEX is able to induce a much larger set of patterns, which turns out to have a positive effect on accuracy.

Part V

Ambiguity Resolution

Chapter 11

Parse Selection with Log-linear Models

11.1 Background

Computational linguistics adopts from theoretical linguistics the notion of a *grammar* as a set of constraints on the relationship between the form of an utterance and its meaning. The properties of grammars have important consequences for the ways in which computational linguists process language. One fact which underlies the search algorithms used for parsing is that the form-meaning relation is in general many to many. For example, the sentence *Chris saw tourists with binoculars* might mean either that Chris saw tourists who carried binoculars, or that Chris saw tourists with the aid of binoculars (see Figure 11.1). This kind of ambiguity is pervasive in language.

Computational linguistics has contributed to the study of language the insight that the grammar allows a huge number of structures per form. This insight is does not follow from common sense, as language users effortlessly recognize the meaning intended in a particular context. But it is the repeated experience of all computational grammar projects, especially those which attempt to encode grammar precisely, that the number of structures which a sentence may have grows exponentially as a function of the length of the sentence in words (Church and Patil, 1982). An illustration of this effect was given in figure 5.2 in chapter 5.

This means that a fundamental problem facing developers of natural language processing systems is that the grammatical constraints created by linguists admit structures in language which no human would recognize. For example, a sentence like *The tourist saw museums* sounds simple enough, but most NLP systems will recognize not only the intended meaning, but also the meaning in which *saw* is a noun, and the entire string is parsed as a determiner *the* followed by a compound noun *tourist saw museums.* This reading is nonsensical, but cannot be ruled out on purely structural grounds without also ruling out the parallel structure in *the circular saw blades.*

The central importance of disambiguation, or of finding the intended reading among the many readings produced by a parser, has been recognized at least since 1963, when a Harvard University research team headed by Susumo Kuno tested

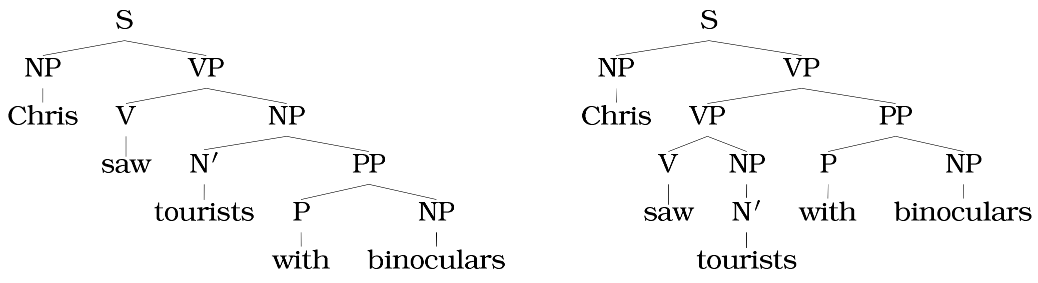


Figure 11.1: Possible structures corresponding to the two readings of *Chris saw tourists with binoculars.*

their parser with the sentence *Time flies like an arrow* and were rewarded with four separate readings. A typical architecture for disambiguation uses a probabilistic context free rule system, where estimates of rule probabilities are de rived from the frequency with which rules have been encountered in collections of parses which have been disambiguated by hand. With a sufficient quantity of annotated training data and careful selection of stochastic features, such systems perform adequately enough on structural disambiguation tasks to support simple applications.

More sophisticated applications such as open-domain question answering or dialog systems, however, require more sophisticated grammar formalisms like Head-Driven Phrase Structure Grammar (Pollard and Sag, 1994). Furthermore, as grammars become more comprehensive, parsers will find an ever larger number of potential readings for a sentence and effective disambiguation becomes even more important. Since these formalisms involve more a complex flow of information than simple context-free grammars, more complex statistical methods are required to capture the subtle dependencies among grammatical structures.

As Abney (1997) shows, the simple rule frequency methods applied to disambiguating context free parses cannot be used for disambiguating constraint grammar parses, since these methods rely crucially on the statistical independence of context-free rule applications. Since constraint-based grammars involve more a complex flow of information than context-free grammars, more complex statistical methods are required to capture the subtle dependencies among grammatical structures.

One solution to the independence problem is provided by maximum entropy models (Jaynes, 1957; Berger, Della Pietra, and Della Pietra, 1996; Della Pietra, Della Pietra, and Lafferty, 1997), a class of exponential models which have proven to be very successful in general for integrating information from disparate and possibly overlapping sources. Maximum entropy (ME) models, variously known as log-linear, Gibbs, exponential, and multinomial logit models, provide a general purpose machine learning technique for classification and prediction which has been successfully applied to fields as diverse as computer vision and econometrics. In particular, maximum entropy models have been fruitfully applied to the problem of disambiguation in constraint based grammar formalisms (Johnson et al., 1999; Riezler et al., 2000; Osborne, 2000; Malouf and Osborne, 2000; Bouma, van Noord, and Malouf, 2001). Maximum entropy models require no unwarranted independence assumptions, and thus are well suited for capturing the kinds of interactions found in constraint-based grammars.

A further benefit of maximum entropy models is that they allow stochastic rule systems to be augmented with additional syntactic, semantic, and pragmatic features, so that a broader range of resources can be brought to bear on the problem of disambiguation. However, the richness of the representations maximum entropy models is not without cost: even modest maximum entropy models can require considerable computational resources and very large quantities of annotate training data in order to accurately estimate the model's parameters.

11.2 Stochastic unification-based grammars

Suppose we have a probability distribution p over a set of events X which are characterized by a d dimensional feature vector function $f : X \to \mathbb{R}^d$. In the context of a stochastic context-free grammar (SCFG), for example, X might be the set of possible trees, and the feature vectors might represent the number of times each rule applied in the derivation of each tree. Our goal is to construct a model distribution q which satisfies the constraints imposed by the empirical distribution p, in the sense that:

$$E_p[f] = E_q[f] \tag{11.1}$$

where $E_p[f]$ is the expected value of the feature vector under the distribution p:

$$E_p[f] = \sum_{x \in X} p(x) f(x)$$

In general, this problem is ill posed: a wide range of models will fit the constraints in (11.1). As a guide to selecting one that is most appropriate, we can call on the *Principle of Maximum Entropy* of Jaynes (1957): "In the absence of additional information, we should assume that all events have equal probability." In other words, we should assign the highest prior probability to distributions which maximize the entropy.

$$H(q) = -\sum_{x \in X} q(x) \log q(x) \tag{11.2}$$

This is effectively a problem in constrained optimization: we want to find a distribution q which maximizes (11.2) while satisfying the constraints imposed by (11.1). It can be straightforwardly shown (Jaynes, 1957; Good, 1963; Campbell, 1970) that the solution to this problem has the parametric form:

$$q_\theta(x) = \frac{\exp\left(\theta^T f(x)\right)}{\sum_{y \in X} \exp\left(\theta^T f(y)\right)} \tag{11.3}$$

where θ is a d-dimensional parameter vector and $\theta^T f(x)$ is the inner product of the parameter vector and a feature vector.

One complication which makes models of this form difficult to apply to problems in natural language processing is that the events space X is often very large or even infinite, making the denominator in (11.3) impossible to compute. One modification we can make to avoids this problem is to consider conditional probability distributions instead (Berger, Della Pietra, and Della Pietra, 1996; Chi, 1998; Johnson et al., 1999). Suppose now that in addition to the event space X and the feature function f, we have also a set of contexts W and a function Y which partitions the members of X. In our SCFG example, W might be the set of possible strings of words, and $Y(w)$ the set of trees whose yield is $w \in W$. Computing the conditional probability $q_\theta(x|w)$ of an event x in context w as

$$q_\theta(x|w) = \frac{\exp\left(\theta^T f(x)\right)}{\sum_{y \in Y(w)} \exp\left(\theta^T f(y)\right)} \tag{11.4}$$

now involves evaluating a more much tractable sum in the denominator.

Of course, in the case of a SCFG, the rule probabilities are independent and we could use a much simpler probability model. The strength of ME models such as (11.4) is that the independence assumption underlying standard SCFG parameter estimation is replaced by the Maximum Entropy Principe, and ME models can be used even when the feature probabilities are not independent. This means that conditional ME models can be used to accurately model rule probabilities for the Alpino grammar. Furthermore, we are not limited to looking only at rule probabilities. Indeed, as we will see in the next sections, we can construct a disambiguation model which is sensitive to a range of distinct but partially overlapping sources of information.

11.2.1 Parse selection

For parse selection, we consider a context w to be a sentence and the events $x \in Y(w)$ within this context are the possible parses of the sentence. Each parse is characterized by a vector of feature values $f(x)$, and may be compared on the basis of those features with other possible parses. Disambiguated parsing proceeds in two steps. In the first step a parse forest is constructed. The second step consists of the selection of the best parse from the parse forest. Following Johnson et al. (1999), the best-first search proceeds on the basis of the unnormalized conditional probabilities derived from the numerator of equation (11.4) for each possible subtree.

The motivation for constructing a parse forest is efficiency: the number of parse trees for a given sentence can be enormous. In addition to this, in most applications the objective will not be to obtain *all* parse trees, but rather the *best* parse tree. Thus, the final component of the parser consists of a procedure to select these best parse trees from the parse forest.

Note that best-first parsing methods proposed for SCFGs (e.g. Caraballo and Charniak (1998)) cannot be used in this context. In attribute-value grammars, it might easily happen that the locally most promising sub-trees cannot be extended to global parses because of conflicting feature constraints.

A variety of parse evaluation functions were considered. A naive algorithm constructs all possible parse trees, assigns each one a score, and then selects the best one. Since it is too expensive to construct all parse trees, we have implemented an algorithm which computes parse trees from the parse forest as an approximate best-first search. This requires that the parse evaluation function is extended to partial parse trees. We implemented a variant of a best-first search algorithm in such a way that for each state in the search space, we maintain the b best candidates, where b is a small integer (the *beam*). If the beam is decreased, then we run a larger risk of missing the best parse (but the result will typically still be a relatively 'good' parse); if the beam is increased, then the amount of computation increases as well.

11.2.2 Experiments and results

To evaluate the effectiveness of ME models for disambiguation, we constructed a model for the Alpino grammar based on the dependency structures in the Alpino treebank. The first step in building the model is to construct the reference distribution $p(x)$ which defines the constraints (11.1). For each sentence in the tree bank, we used the Alpino parser to construct a number of parses. Ideally, we would use all parses licensed by the grammar as part of the training data. However, as a few sentences have an astronomical number of parses, this is not practical. In addition, as Osborne (2000) observes, constructing a model based on a representative sample of parses is not only more efficient, it often gives as good or better results.

Next, each parse was assigned a probability proportional to the number of errors in its dependency structure, determined by comparison to the dependency structure stored in the treebank. An 'error' is any relation which appears in the dependency structure constructed by the parser but not in the treebank, or conversely which appears in the treebank but not in the constructed parse.

Then, since the model for the probability of parses is based on the probabilities of features, each parse x is represented by a feature vector $f(x)$ which characterizes the properties of parses which the probability model will be sensitive to. These features should not be confused with the features in an HPSG feature structure. The features in this stochastic parsing model are chosen by the grammarian and in principle they can be any characteristic of the parse that can be counted. The features that we use at present are grammar rules, dependency relations, subcategorization frames, and unknown word heuristics. For dependency relations and subcategorization frames, we include both a fully lexicalized version of the feature and a version backed off to fairly coarse-grained part of speech labels.

As an example of the features used in the model, consider one parse of the following sentence in the treebank:

(1)	Hij	is	na	zijn	reis	naar	de	telefooncel	niet	meer	teruggekeerd.
	he	is	after	his	trip	to	the	phone.booth	not	more	returned

'He never returned after his trip to the phone booth.'

The rule features assigned are shown in Figure 11.2. The rules range from extremely general–every parse includes exactly one invocation of the rule *robust*–to very specific, and are equivalent to the features that would be used in a probability model for a SCFG. Besides the rule features, which are derived from the constituent structure of the parse, we also add features based on its dependency structure. First, for every head-dependent relation, there is a dependency feature (see Figure 11.3). Each dependency feature is a triple of pairs: the part of speech and

robust	top_start_xp(1)	mod1
max_xp(root)	a_adv_a	non_wh_topicalization(np_light,2)
imp	subj_topic	vproj_vc
v2_vp_vproj	vp_mod_v	vgap
mod2	pp_p_arg(np)	adv_a
np_det_n	n_n_pps	a_adv_a(not_er)
pp_p_arg(np)	n(naar_a)	a_adv_a(not_er)
np_det_n	vp_mod_v	v_v_v
vgap	vc_vb	

Figure 11.2: Rule features and unknown word heuristics

lexeme of the head, a relation type and direction, and the part of speech and lexeme of the dependent, including both lexicalized and backed off versions. Finally, subcategorization features (Figure 11.4) list all of the types of all the relations which occur in the parse with a particular word as the head. These features too appear in both lexicalized and backed off versions.

Finally, when we have constructed for each sentence w a set of parses $x \in Y(w)$ with their reference probabilities $p(x|w)$ and the corresponding feature vectors f, we need to find a parameter vector θ such that the conditional probabilities (11.4) maximize the likelihood of the training data (or, equivalently, minimize the divergence between the predicted distribution $q(x|w)$ and the reference probabilities $p(x|w)$).

The values of the parameters can be interpreted as weights reflecting preferences for particular features: a large positive weight denotes a preference for the model to use a certain feature, whereas a negative weight denotes a dispreference. Various algorithms exist that guarantee to find the global optimal settings for these weights so that the probability distribution in the training set is best represented, which we will discuss in more detail in section 12. For the results reported in this section, we used a limited memory variable metric algorithm.

For these experiments, we used the first 250 parses of 6,182 sentences from the treebank. For 76% of the sentences, the parsers found fewer than 250 parses, and for those sentences all parses were used. Overall, there were and average of 86.6 parses per sentence in the training data. Two models were constructed, a simple model using only 315 rule features, and a fully lexicalized model using 830,899 rule, dependency, and subcategorization features. The performance of these models, evaluated using 10-fold cross-validation, are given in Table 11.1. The concept accuracy metrics are defined in chapter 5.

Also shown are the results of using no disambiguation model (simply using the first parse produced by the parser), and the best possible result (which is determined by the limitations in coverage of the Alpino grammar).

As these results show, adding lexical and lexicalized features to the model does substantially increase the performance of the model, though at the cost of substantial complexity. In addition, there is considerable room for improvement, both in grammatical coverage and in the performance of the disambiguation model. Current work is focused on addressing both the complexity and the performance issues

verb:ben:su:left:pronoun:hij	verb:_:su:left:pronoun:hij
verb:ben:su:left:pronoun:_	verb:_:su:left:pronoun:_
verb:ben:vc:right:verb:keer	verb:_:vc:right:verb:keer
verb:ben:vc:right:verb:_	verb:_:vc:right:verb:_
verb:keer:su:left:pronoun:hij	verb:_:su:left:pronoun:hij
verb:keer:su:left:pronoun:_	verb:_:su:left:pronoun:_
verb:keer:mod:left:preposition:na	verb:_:mod:left:preposition:na
verb:keer:mod:left:preposition:_	verb:_:mod:left:preposition:_
preposition:na:obj1:right:noun:reis	preposition:na:obj1:right:noun:_
preposition:_:obj1:right:noun:reis	preposition:_:obj1:right:noun:_
noun:reis:mod:right:preposition:naar	noun:_:mod:right:preposition:naar
noun:reis:mod:right:preposition:_	noun:_:mod:right:preposition:_
preposition:naar:obj1:right:noun:telefooncel	preposition:_:obj1:right:noun:telefooncel
preposition:naar:obj1:right:noun:_	preposition:_:obj1:right:noun:_
noun:telefooncel:det:left:determiner:de	noun:_:det:left:determiner:de
noun:telefooncel:det:left:determiner:_	noun:_:det:left:determiner:_
noun:reis:det:left:determiner:zijn	noun:_:det:left:determiner:zijn
noun:reis:det:left:determiner:_	noun:_:det:left:determiner:_
verb:keer:mod:left:adjective:meer	verb:_:mod:left:adjective:meer
verb:keer:mod:left:adjective:_	verb:_:mod:left:adjective:_
adjective:meer:mod:left:adverb:niet	adjective:_:mod:left:adverb:niet
adjective:meer:mod:left:adverb:_	adjective:_:mod:left:adverb:_

Figure 11.3: Dependency features

noun:telefooncel:[det]	noun:_:[det]
preposition:naar:[obj1]	preposition:_:[obj1]
noun:reis:[det,mod]	noun:_:[det,mod]
preposition:na:[obj1]	preposition:_:[obj1]
adjective:meer:[mod]	adjective:_:[mod]
verb:keer:[su,mod,mod]	verb:_:[su,mod,mod]
verb:ben:[su,vc]	verb:_:[su,vc]

Figure 11.4: Subcategorization frames

	CA	CA_κ
No model	68.76	0.00
Rules	78.56	59.65
Lex+Backoff	80.12	69.14
Best	85.19	100.00

Table 11.1: Performance evaluation of disambiguation models

by more elaborate feature and model selection strategies.

11.3 Feature merging

One of the goals of model selection is to maximally exploit the small but informative training data set we have by taking a more sophisticated view of the feature set itself. To this end we experimented with the method of *feature merging*, a means of constructing equivalence classes of statistical features based upon common elements within them. This technique allows the model to retain information which would otherwise be discarded in a simple frequency-based feature cutoff by producing new, generalized features which serve as a variant of backed-off features.

11.3.1 The Features and Feature Merging

The model depends on the distribution of the features and their informativeness, thus it is important that the features used be germane to the task. In parsing, features should reflect the sort of information pertinent to making structural decisions.

In the present experiments, we employ several types of features corresponding to grammatical rules, valency frames, lexicalized dependency triples, and lexical features constituting surface forms, base forms, and lexical frames (a subset of the sentences and features used for the experiments described in the section 11.2.2). Instances of each feature type were collected from the training data in advance to yield a feature set consisting of 82,371 distinct features.

Examples of these features may be seen below, where example 1 is a rule for creating a VP, 2 contains a valency frame for the noun *mens*, 3 describes a dependency triple between the noun *mens* and the adjective *modern*, and the direction of the modification, and finally example 4 contains lexical information about the word *modern* as it occurs in context.

1 `vp_arg_v(np)`
2 `noun(de):mens:[mod]`
3 `noun:mens:mod:left:adjective:modern`
4 `moderne:modern:adjective(e,adv)`

11.3.2 Noise reduction and feature merging

The feature set used here exploits the maxent technique in that it relies on features which are overlapping and mutually dependent. The features represent varying degrees of linguistic generality and hence some occur much more frequently than others. Furthermore, the features may also represent information which is redundant in the sense that it is represented in multiple different features, in which case we say that the features “overlap”. Features which share information in this way are necessarily dependent in their distributions.

The overlapping features allow for a variety of “backing off” in which features which share a structure but contain less specific information than others are used in the same model as features with more specific information.

It is desirable that the features be as informative as possible. The model should contain specific features to the extent that the features' distributions are accurately represented in the training data. There is a point, however, regardless of the size of the corpus, at which the specificity of features translates to sparseness in the data, causing noise and leading to deterioration in the model's performance.

A number of approaches have been taken to smoothing exponential models, including imposing a Gaussian prior over the distribution (Chen and Rosenfeld, 1999) and building the feature set up by a process of induction to ensure that only maximally representative features are admitted into the model (Della Pietra, Della Pietra, and Lafferty, 1997). The most commonly employed and computationally inexpensive approach to reducing noise is to use a frequency-based feature cutoff (Ratnaparkhi, 1998), in which features which occur fewer times in the training data than some predetermined cutoff are eliminated. This has shown to be an effective way to improve results. Because of its simplicity and effectiveness, it is the approach we have focused on improving on in the present research. Although it is an effective way to reduce noise in a model, there is a risk with a cutoff that information encoded in the discarded features may be useful. A feature may be rare due to some rare element within it, but otherwise useful. To prevent discarding such useful features, we experiment with a method of *feature merging* similar to that introduced by Mullen and Osborne (2000) and explored in detail by Mullen (2002). This approach considers features as being composed of informative elements. Before any feature cutoff is applied, features which are identical except for particular rare elements are generalized by merging; that is, the features are unioned and considered as a single feature. The elements upon which these merges are done are determined with a pre-set threshold, and merges are done on elements which occur fewer times than this. The merging process eliminates the distinction between two features, thus eliminating the information provided by the element which distinguishes them, while the rest of the information provided by the merged features remains intact in the model.

Individual unique features may be considered as sets of instantiations in the data. A feature which is the result of merging is thus the union of the features which were merged. The count of occurrences of the new feature is the sum of the counts of the merged features. If a cutoff is incorporated subsequently, the newly merged feature is more likely to survive in the model, as its frequency is greater than each of the features before merging. Thus information in features which otherwise might have been lost in a cutoff is retained in the form of a more general feature.

11.3.3 Building merged models

The first step is to determine how the features are composed and what the elements are which make them up. Factors which contribute most to sparseness, such as lexical items and certain grammatical attributes, are good candidates. In the present work, lexical items, both sentence forms and word stems, are considered as elements. Frequency counts are taken for all elements. A threshold is determined using a held-out test set. Using this threshold, a new model is created as follows: in the representation of the original model's features, all instances of elements which occur fewer times than the threshold are replaced by a dummy element. Features which were identical aside from these infrequent elements are thus rendered completely identical. For example, let

$$\begin{aligned} \text{feature 1} &= \mathtt{A} : \mathtt{B} \text{ with count 2} \\ \text{feature 2} &= \mathtt{A} : \mathtt{C} \text{ with count 4} \end{aligned}$$

where A, B, and C are elements. We may count the occurrences of each element in the training data and find that the count of A is 20, of B is 5, and of C is 7. We determine by use of held-out test data that an optimal cutoff is, e.g., 10. Since both B and C have counts lower than this, all instances of B and C are replaced by a dummy element X. Thus features 1 and 2 above are both in effect replaced by feature 3, below, whose count is now the sum of those of the features which have been merged.

$$\text{feature 3} = \mathtt{A} : \mathtt{X} \text{ with count 6}$$

Iterative scaling was performed on the new feature set to obtain the appropriate maximum entropy weights (see section 12).

11.3.4 Composition of features

A quality of compositionality is necessary in features in order to perform the merging operation. That is, it is necessary that features be composed of discrete elements for which frequency counts can be attained from the data. The features described in subsection 11.3.1 may be viewed as being composed of words, base forms, POS tags, grammar attributes, and other discrete elements which occur together in a particular way. Merging proceeds by first establishing a merging threshold via experiments on held-out data. Frequencies of all elements are gathered from the training data. Finally, features containing elements whose counts are fewer than the threshold are merged. This is done by replacing all instances of sub-threshold elements with a dummy element in features. For example, if it were found that the element `modern` had a count less below the threshold, all features containing that would be altered. A feature such as

```
noun:mens:mod:left:adjective:modern
```

would be changed to

```
noun:mens:mod:left:adjective:xxxxx
```

and likewise if the element `aardig` occurred with a count below the threshold, the same would be done with the feature

```
noun:mens:mod:left:adjective:aardig
```

so that both features merged as the single feature

```
noun:mens:mod:left:adjective:xxxxx
```

with a count equal to the sum of the two merged features.

Model	CA	CA_κ
baseline	63.01	0.00
maxent	77.45	56.99
maxent+cutoff	79.36	64.52
maxent+cutoff+merge	80.01	67.08
best possible	88.35	100.00

Table 11.2: Preliminary results on held-out data

11.3.5 Why feature merging?

It is well known that many models benefit from a frequency-based feature cutoff. Using feature merging, we seek to take a more sophisticated view of the features themselves, allowing the same degree of noise reduction as a feature cutoff, while simultaneously generalizing the features to obtain the benefits of backing off. By operating on sub-feature information sources, we hope to discard noisy information from the model with a greater degree of control, maintaining useful information contained by features which would otherwise be lost.

11.3.6 Experiments

Experiments to evaluate the effectiveness of feature merging were performed using the features described above with a training set of 1,220 sentences, whose parses totaled 626,699 training events, initially with a held-out set of 131 sentences and subsequently on a test set of unseen sentences totaling 566 sentences. A merging threshold of 500 and a feature cutoff of 500 were determined by use of the held-out test set. The number of active (non-zero weighted) features in the original model was 75,500, the number of active features in the model with cutoff alone was 11,639, and the number of active features in the model which had been merged prior to the cutoff was 11,890.

As in the previous experiments, for each sentence in the test set, the dependency structure of the highest scoring parse was extracted and compared to the gold standard dependency structure in the treebank, and the concept accuracy was calculated. The results given below for each model are in terms of average per-sentence raw and adjusted concept accuracy.

Preliminary results on held-out data to establish thresholds, shown in table 11.2, were promising, suggesting that incorporation of merging with a merge threshold of 500 elements performed somewhat better than the best possible feature cutoff of 500 elements alone.

Unfortunately, these preliminary results were not supported by experiments on unseen data. As can be seen in table 11.3, the averaged results of four folds of ten-fold cross validation, representing a total of 566 sentences, show that the use of merging at the threshold determined by the held-out data does not appear to benefit the model.

Thus, results so far are inconclusive. The effectiveness of the merging technique appears to depend greatly on qualities of the feature set itself. It is hoped that

Model	CA	CA_{κ}
baseline	58.92	0.00
maxent	71.53	45.77
maxent+cutoff	73.67	53.54
maxent+cutoff+merge	73.26	52.05
best possible	86.47	100.00

Table 11.3: Results on unseen data

further experiments with different types of features will shed light on circumstances in which feature merging may be most effectively employed as a tool to optimize model performance.

11.4 Conclusions and future work

In this section we have shown how to construct maximum entropy models for parse selection. These models are capable to some extent to disambiguate natural language sentences. We showed that an accuracy of about 80% can be obtained using such models. Given the current state of the grammar, this means that the maximum entropy model solves about 70% of the disambiguation problem.

We have also attempted to improve the models by means of feature selection. The results we obtained are inconclusive.

In the near future we hope to improve the models in two ways. Firstly, we will attempt to apply other feature selection methods in order to construct accurate and compact models. Secondly, we will attempt to improve our results by incorporating *unsupervised* methods. Such methods do not rely on the availability of large sets of annotated sentences. If models are to be sensitive to lexical information, then it appears unlikely that the amount of annotated date that would be required will ever become available. Therefore, methods that are capable of extracting lexical information from unannotated data or automatically annotated data appear to be a necessity.

Chapter 12

Parameter estimation

A leading advantage of maximum entropy models is their flexibility: they allow stochastic rule systems to be augmented with additional syntactic, semantic, and pragmatic features. However, the richness of the representations is not without cost. Even modest ME models can require considerable computational resources and very large quantities of annotated training data in order to accurately estimate the model's parameters. While parameter estimation for ME models is conceptually straightforward, in practice ME models for disambiguation, like many other natural language tasks, are usually quite large, and frequently contain hundreds of thousands of free parameters. Estimation of such large models is not only expensive, but also, due to sparsely distributed features, sensitive to round-off errors. Thus, highly efficient, accurate, scalable methods are required for estimating the parameters of practical models.

In this chapter, we discuss experiments comparing a number of algorithms for estimating the parameters of ME models, including *Generalized Iterative Scaling* and *Improved Iterative Scaling*, as well as general purpose optimization techniques such as *gradient ascent*, *conjugate gradient*, and *variable metric* methods. Surprisingly, the widely used iterative scaling algorithms perform quite poorly, and for all of the test problems, a limited memory variable metric algorithm outperformed the other choices.

12.1 Maximum likelihood estimation

Given the parametric form of an ME model in (11.4), fitting an ME model to a collection of training data entails finding values for the parameter vector θ which minimize the Kullback-Leibler divergence between the model q_θ and the empirical distribution p:

$$D(p||q_\theta) = \sum_{w,x} p(x,w) \log \frac{p(x|w)}{q_\theta(x|w)}$$

or, equivalently, which maximize the log likelihood:

$$L(\theta) = \sum_{w,x} p(w,x) \log q_\theta(x|w) \tag{12.1}$$

```
ESTIMATE(p)
1  $\theta^0 \leftarrow 0$
2  $k \leftarrow 0$
3  repeat
4          compute $q^{(k)}$ from $\theta^{(k)}$
5          compute update $\delta^{(k)}$
6          $\theta^{(k+1)} \leftarrow \theta^{(k)} + \delta^{(k)}$
7          $k \leftarrow k+1$
8    until converged
9  return $\theta^{(k)}$
```

Figure 12.1: General parameter estimation algorithm

The gradient of the log likelihood function, or the vector of its first derivatives with respect to the parameter θ is:

$$G(\theta) = \sum_{x,y} p(x,y)f(y) - \sum_{x,y} p(x)q_\theta(y|x)f(y)$$

or, simply:

$$G(\theta) = E_p[f] - E_{q_\theta}[f] \tag{12.2}$$

Since the likelihood function (12.1) is concave over the parameter space, it has a global maximum where the gradient is zero. Unfortunately, simply setting $G(\theta) = 0$ and solving for θ does not yield a closed form solution, so we proceed iteratively. Figure 12.1. At each step, we adjust an estimate of the parameters $\theta^{(k)}$ to a new estimate $\theta^{(k+1)}$ based on the divergence between the estimated probability distribution $q^{(k)}$ and the empirical distribution p. We continue until successive improvements fail to yield a sufficiently large decrease in the divergence.

While all parameter estimation algorithms we will consider take the same general form, the method for computing the updates $\delta^{(k)}$ at each search step differs substantially. As we shall see, this difference can have a dramatic impact on the number of updates required to reach convergence.

12.2 Iterative Scaling

One popular method for iteratively refining the model parameters is *Generalized Iterative Scaling* (GIS), due to Darroch and Ratcliff (1972). An extension of Iterative Proportional Fitting (Deming and Stephan, 1940), GIS scales the probability distribution $q^{(k)}$ by a factor proportional to the ratio of $E_p[f]$ to $E_{q^{(k)}}[f]$, with the restriction that $\sum_j f_j(x) = C$ for each event x in the training data (a condition which can be easily satisfied by the addition of a correction feature). We can adapt GIS to estimate the model parameters θ rather than the model probabilities q, yielding the update rule:

$$\delta^{(k)} = \log\left(\frac{E_p[f]}{E_{q^{(k)}}[f]}\right)^{\frac{1}{C}}$$

The step size, and thus the rate of convergence, depends on the constant C: the larger the value of C, the smaller the step size. In case not all rows of the training data sum to a constant, the addition of a correction feature effectively slows convergence to match the most difficult case. To avoid this slowed convergence and the need for a correction feature, Della Pietra, Della Pietra, and Lafferty (1997) propose an *Improved Iterative Scaling* (IIS) algorithm, whose update rule is the solution to the equation:

$$E_p[f] = \sum_{w,x} p(w)q^{(k)}(x|w)f(x)\exp(M(x)\delta^{(k)})$$

where $M(x)$ is the sum of the feature values for an event x in the training data. This is a polynomial in $\exp\left(\delta^{(k)}\right)$, and the solution can be found straightforwardly using, for example, the Newton-Raphson method.

12.3 First order methods

Iterative scaling algorithms have a long tradition in statistics and are still widely used for analysis of contingency tables. Their primary strength is that on each iteration they only require computation of the expected values $E_{q^{(k)}}$. They do not depend on evaluation of the gradient of the log-likelihood function, which, depending on the distribution, could be prohibitively expensive. In the case of ME models, however, the vector of expected values required by iterative scaling essentially *is* the gradient G. Thus, it makes sense to consider methods which use the gradient directly.

The most obvious way of making explicit use of the gradient is by *Cauchy's method*, or the method of *steepest ascent* (Zhu, We, and Mumford, 1997). The gradient of a function is a vector which points in the direction in which the function's value increases most rapidly. Since our goal is to maximize the log-likelihood function, a natural strategy is to shift our current estimate of the parameters in the direction of the gradient via the update rule:

$$\delta^{(k)} = \alpha^{(k)} G(\theta^{(k)})$$

where the step size $\alpha^{(k)}$ is chosen to maximize $L(\theta^{(k)} + \delta^{(k)})$. Finding the optimal step size is itself an optimization problem, though only in one dimension and, in practice, only an approximate solution is required to guarantee global convergence.

Since the log-likelihood function is concave, the method of steepest ascent is guaranteed to find the global maximum. However, while the steps taken on each iteration are in a very narrow sense locally optimal, the global convergence rate of steepest ascent is very poor. As shown in Figure 12.2, each new search direction is orthogonal (or, if an approximate line search is used, nearly so) to the previous direction. This leads to a characteristic "zig-zag" ascent, with convergence slowing as the maximum is approached.

One way of looking at the problem with steepest ascent is that it considers the same search directions many times. We would prefer an algorithm which considered each possible search direction only once, in each iteration taking a step of exactly the right length in a direction orthogonal to all previous search directions.

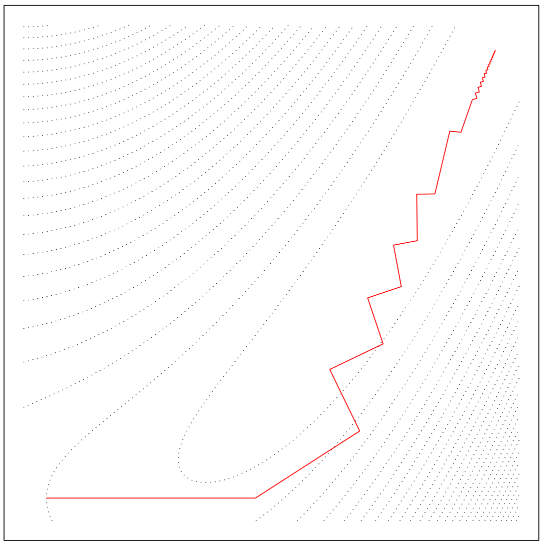


Figure 12.2: Steepest ascent in two dimensions

This intuition underlies *conjugate gradient* methods (see, e.g., Shewchuk, 1994) which choose a search direction which is a linear combination of the steepest ascent direction and the previous search direction. The step size is selected by an approximate line search, as in the steepest ascent method. Several non-linear conjugate gradient methods, such as the *Fletcher-Reeves* (cg-fr) and the *Polak-Ribire-Positive* (cf-prp) algorithms, have been proposed. While theoretically equivalent, they use slightly different update rules and thus show different numeric properties.

12.4 Second order methods

Another way of looking at the problem with steepest ascent is that while it takes into account the gradient of the log-likelihood function, it fails to take into account its curvature, or the gradient of the gradient. The usefulness of the curvature is made clear if we consider a second-order Taylor series approximation of $L(\theta + \delta)$:

$$L(\theta + \delta) \approx L(\theta) + \delta^T G(\theta) + \frac{1}{2}\delta^T H(\theta)\delta \tag{12.3}$$

where H is *Hessian matrix* of the log-likelihood function, the $d \times d$ matrix of its second partial derivatives with respect to θ. If we set the derivative of (12.3) to zero and solve for δ, we get the update rule for *Newton's method*:

$$\delta^{(k)} = H^{-1}(\theta^{(k)})G(\theta^{(k)}) \tag{12.4}$$

Newton's method converges very quickly (for quadratic objective functions, in one step), but it requires the computation of the inverse of the Hessian matrix on each iteration.

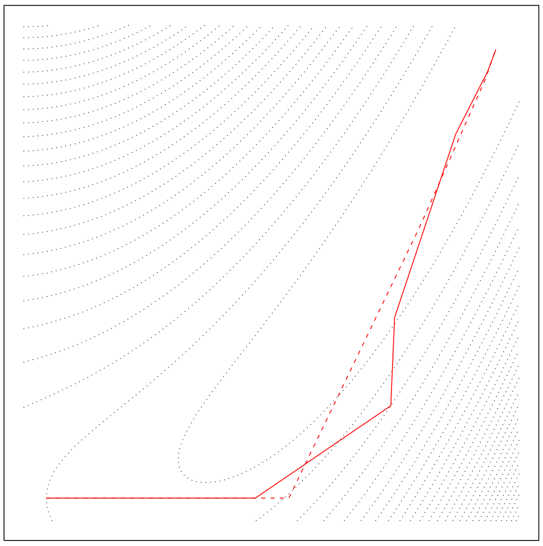


Figure 12.3: Limited memory variable metric method (dashed lines show Newton's method for comparison)

While the log-likelihood function for ME models in (12.1) is twice differentiable, for large scale problems the evaluation of the Hessian matrix is computationally impractical, and Newton's method is not competitive with iterative scaling or first order methods. *Variable metric* or *quasi-Newton* methods avoid explicit evaluation of the Hessian by building up an approximation of it using successive evaluations of the gradient. That is, we replace $H^{-1}(\theta^{(k)})$ in (12.4) with a local approximation of the inverse Hessian $B^{(k)}$:

$$\delta^{(k)} = B^{(k)} G(\theta^{(k)})$$

with $B^{(k)}$ a symmetric, positive definite matrix which satisfies the equation:

$$B^{(k)} y^{(k)} = \delta^{(k-1)}$$

where $y^{(k)} = G(\theta^{(k)}) - G(\theta^{(k-1)})$.

Variable metric methods also show excellent convergence properties and can be much more efficient than using true Newton updates, but for large scale problems with hundreds of thousands of parameters, even storing the approximate Hessian is prohibitively expensive. For such cases, we can apply *limited memory variable metric* methods, which implicitly approximate the Hessian matrix in the vicinity of the current estimate of $\theta^{(k)}$ using the previous m values of $y^{(k)}$ and $\delta^{(k)}$. Since in practical applications values of m between 3 and 10 suffice, this can offer a substantial savings in storage requirements over variable metric methods, while still giving favorable convergence properties (see Figure 12.3).[1]

[1]For algorithmic details and theoretical analysis of first and second order methods, see, e.g., Nocedal (1997) or Nocedal and Wright (1999).

12.5 Comparing estimation techniques

The performance of optimization algorithms is highly dependent on the specific properties of the problem to be solved. Worst-case analysis typically does not reflect the actual behavior on actual problems. Therefore, in order to evaluate the performance of the optimization techniques sketched in previous section when applied to the problem of parameter estimation, we need to compare the performance of actual implementations on realistic data sets (Dolan and Moré, 2000; Benson, McInnes, and Moré, 2000; Dolan and Moré, 2002).

As a basis for the implementation, we have used PETSc (the "Portable, Extensible Toolkit for Scientific Computation"), a software library designed to ease development of programs which solve large systems of partial differential equations (Balay et al., 2001; Balay et al., 1997; Balay et al., 2002). PETSc offers data structures and routines for parallel and sequential storage, manipulation, and visualization of very large sparse matrices.

For any of the estimation techniques, the most expensive operation is computing the probability distribution q and the expectations $E_q[f]$ for each iteration. In order to make use of the facilities provided by PETSc, we can store the training data as a (sparse) matrix F, with rows corresponding to events and columns to features. Then given a parameter vector θ, the unnormalized probabilities $\dot{q}_\theta$ are the matrix-vector product:

$$\dot{q}_\theta = \exp F\theta$$

and the feature expectations are the transposed matrix-vector product:

$$E_{q_\theta}[f] = F^T q_\theta$$

By expressing these computations as matrix-vector operations, we can take advantage of the high performance sparse matrix primitives of PETSc.

For the comparison, we implemented both Generalized and Improved Iterative Scaling in C++ using the primitives provided by PETSc. For the other optimization techniques, we used TAO (the "Toolkit for Advanced Optimization"), a library layered on top of the foundation of PETSc for solving non-linear optimization problems (Benson et al., 2002). TAO offers the building blocks for writing optimization programs (such as line searches and convergence tests) as well as high-quality implementations of standard optimization algorithms (including conjugate gradient and variable metric methods).

Before turning to the results of the comparison, two additional points need to be made. First, in order to assure a consistent comparison, we need to use the same stopping rule for each algorithm. For these experiments, we judged that convergence was reached when the relative change in the log-likelihood between iterations fell below a predetermined threshold. That is, each run was stopped when:

$$\frac{|L(\theta^{(k)}) - L(\theta^{(k-1)})|}{L(\theta^{(k)})} < \epsilon \qquad (12.5)$$

where the relative tolerance $\epsilon = 10^{-7}$. For any particular application, this may or may not be an appropriate stopping rule, but is only used here for purposes of comparison.

dataset	classes	contexts	features	non-zeros
rules	29,602	2,525	246	732,384
lex	42,509	2,547	135,182	3,930,406
summary	24,044	12,022	198,467	396,626
shallow	8,625,782	375,034	264,142	55,192,723

Table 12.1: Datasets used in experiments

Finally, it should be noted that in the current implementation, we have not applied any of the possible optimizations that appear in the literature (Lafferty and Suhm, 1996; Wu and Khudanpur, 2000; Lafferty, Pereira, and McCallum, 2001) to speed up normalization of the probability distribution q. These improvements take advantage of a model's structure to simplify the evaluation of the denominator in (11.4). The particular data sets examined here are unstructured, and such optimizations are unlikely to give any improvement. However, when these optimizations are appropriate, they will give a proportional speed-up to all of the algorithms. Thus, the use of such optimizations is independent of the choice of parameter estimation method.

12.6 Experiments

To compare the algorithms described in §12.1, we applied the implementation outlined in the previous subsection to four training data sets (described in Table 12.1) drawn from the domain of natural language processing. The 'rules' and 'lex' datasets are examples of stochastic attribute value grammars from the experiments described in section 11.2.2. The 'summary' dataset is part of a sentence extraction task (Osborne, to appear), and the 'shallow' dataset is drawn from a text chunking application (Osborne, 2002). These datasets vary widely in their size and composition, and are representative of the kinds of datasets typically encountered in applying ME models to NLP classification tasks.

The results of applying each of the parameter estimation algorithms to each of the datasets is summarized in Table 12.2. For each run, we report the KL divergence between the fitted model and the training data at convergence, the prediction accuracy of fitted model on a held-out test set (the fraction of contexts for which the event with the highest probability under the model also had the highest probability under the reference distribution), the number of iterations required, the number of log-likelihood and gradient evaluations required (algorithms which use a line search may require several function evaluations per iteration), and the total elapsed time (in seconds).[2]

There are a few things to observe about these results. First, while IIS converges in fewer steps the GIS, it takes substantially more time. At least for this imple-

[2]The reported time does not include the time required to input the training data, which is difficult to reproduce and which is the same for all the algorithms being tested. All tests were run using one CPU of a dual processor 1700MHz Pentium 4 with 2 gigabytes of main memory at the Center for High Performance Computing and Visualisation, University of Groningen.

Dataset	Method	KL Div.	Acc	Iters	Evals	Time
rules	gis	5.124×10^{-2}	47.00	1186	1187	16.68
	iis	5.079×10^{-2}	43.82	917	918	31.36
	steepest ascent	5.065×10^{-2}	44.88	224	350	4.80
	conjugate gradient (fr)	5.007×10^{-2}	44.17	66	181	2.57
	conjugate gradient (prp)	5.013×10^{-2}	46.29	59	142	1.93
	limited memory var. metric	5.007×10^{-2}	44.52	72	81	1.13
lex	gis	1.573×10^{-3}	46.74	363	364	31.69
	iis	1.487×10^{-3}	42.15	235	236	95.09
	steepest ascent	3.341×10^{-3}	42.92	980	1545	114.21
	conjugate gradient (fr)	1.377×10^{-3}	43.30	148	408	30.36
	conjugate gradient (prp)	1.893×10^{-3}	44.06	114	281	21.72
	limited memory var. metric	1.366×10^{-3}	43.30	168	176	20.02
summary	gis	1.857×10^{-3}	96.10	1424	1425	107.05
	iis	1.081×10^{-3}	96.10	593	594	188.54
	steepest ascent	2.489×10^{-3}	96.33	1094	3321	190.22
	conjugate gradient (fr)	9.053×10^{-5}	95.87	157	849	49.48
	conjugate gradient (prp)	3.297×10^{-4}	96.10	112	537	31.66
	limited memory var. metric	5.598×10^{-5}	95.54	63	69	8.52
shallow	gis	3.314×10^{-2}	14.19	3494	3495	21223.86
	iis	3.238×10^{-2}	5.42	3264	3265	66855.92
	steepest ascent	7.303×10^{-2}	26.74	3677	14527	85062.53
	conjugate gradient (fr)	2.585×10^{-2}	24.72	1157	6823	39038.31
	conjugate gradient (prp)	3.534×10^{-2}	24.72	536	2813	16251.12
	limited memory var. metric	3.024×10^{-2}	23.82	403	421	2420.30

Table 12.2: Results of comparison.

mentation, the additional bookkeeping overhead required by IIS more than cancels any improvements in speed offered by accelerated convergence. This may be a misleading conclusion, however, since a more finely tuned implementation of IIS may well take much less time per iteration than the one used for these experiments. However, even if each iteration of IIS could be made as fast as an iteration of GIS (which seems unlikely), the benefits of IIS over GIS would in these cases be quite modest.

Second, note that for three of the four datasets, the KL divergence at convergence is roughly the same for all of the algorithms. For the 'summary' dataset, however, they differ by up to two orders of magnitude. This is an indication that the convergence test in (12.5) is sensitive to the rate of convergence and thus to the choice of algorithm. Any degree of precision desired could be reached by any of the algorithms, with the appropriate value of ϵ. However, GIS, say, would require many more iterations than reported in Table 12.2 to reach the precision achieved by the limited memory variable metric algorithm.

Third, the prediction accuracy is, in most cases, more or less the same for all of the algorithms. Some variability is to be expected—all of the data sets being considered here are badly ill-conditioned, and many different models will yield the same likelihood. In a few cases, however, the prediction accuracy differs more substantially. For the two SAVG data sets ('rules' and 'lex'), GIS has a small advantage over the other methods. More dramatically, both iterative scaling methods perform very poorly on the 'shallow' dataset. In this case, the training data is very sparse. Many features are nearly 'pseudo-minimal' in the sense of Johnson et al. (1999), and so receive weights approaching $-\infty$. Smoothing the reference probabilities would likely improve the results for all of the methods and reduce the observed differences. However, this does suggest that gradient-based methods are robust to certain problems with the training data.

12.7 Conclusion

The most significant lesson to be drawn from these results is that, with the exception of steepest ascent, gradient-based methods outperform iterative scaling by a wide margin for almost all the datasets, as measured by both number of function evaluations and by the total elapsed time. And, in each case, the limited memory variable metric algorithm performs substantially better than any of the competing methods.

Chapter 13

Word Sense Disambiguation

A major problem in natural language processing is that of lexical ambiguity, be it syntactic or semantic. A word's *syntactic* ambiguity can be resolved by applying part-of-speech taggers which predict the syntactic category of a word in texts with high levels of accuracy (see for example (Brill, 1995) or (Brants, 2000)). The problem of resolving *semantic* ambiguity, which is generally known as word sense disambiguation (WSD), has proved to be more difficult than syntactic disambiguation.

Disambiguation involves two major steps: First, the possible senses for every ambiguous word have to be determined. This can either be achieved through an inventory of senses (e.g. Machine Readable Dictionaries (MRDs)), listing equivalents in a different language (bilingual dictionaries) or grouping features, categories and/or associated words. In a second step, the appropriate sense has to be assigned to ambiguous words using information about the context (linguistic and extra-linguistic) as well as external knowledge sources.

The only way to assign the meaning of a word in a particular usage is thus to examine its context. For instance, the English word *bank*—an extensively cited example of lexical ambiguity—can refer to the bank of a river or to the pecuniary institution. For this reason, a computer program analyzing the sentence "The boy leapt from the bank into the cold water" will need to decide which reading of 'bank' was intended, in order to be able to come up with the correct meaning for the sentence. The overall goal of word sense disambiguation systems is to attribute the correct sense(s) to words in a text.

13.1 Approaches to WSD

There are three ways to approach the problem of assigning the correct sense(s) to ambiguous words in context: a *knowledge-based approach*, which uses an explicit lexicon (MRDs, Thesauri), *corpus-based disambiguation*, where the relevant information about word senses is gathered from training on a large corpus, or, third alternative, a *hybrid approach* combining aspects of the aforementioned methodologies (see (Ide and Véronis, 1998) for a more thorough discussion).

A corpus-based approach has the advantage that text material is easily accessible. The possible means used to attribute senses to ambiguous words are then *distributional information* and *contextwords*. Distributional information about an ambiguous word is the frequency distribution of its senses. Contextwords are the words found to the right and/or the left of a certain word, thus collocational information.

There are two possible approaches to corpus-based WSD systems: *Supervised* and *unsupervised WSD*. Supervised approaches use a training and a testing phase. During training on a disambiguated corpus, probabilistic information about contextwords as well as distributional information about the different senses of an ambiguous word are collected. In the testing phase, the sense with the highest probability computed on the basis of the training data (contextwords) is chosen. Training and evaluating such an algorithm presupposes the existence of sense-tagged corpora. Unsupervised algorithms, on the other hand, are applied to raw text material and annotated data is only needed for evaluation.

The major difficulties of a corpus-based approach are the need for manual sense-tagging and data sparseness. So far there has not been a lot of sense-tagged material made publicly available, and even for English the corpora are still very (if not too) small. One approach to solve the problem is to manually sense-tag corpora using e.g. (Euro)WordNet hierarchies. Another, less time consuming, possibility is the application of unsupervised machine learning techniques (Schütze, 1998) to WSD (although the ‘evaluation problem’ stays the same, see section 13.2).

The difficulty of data sparseness for WSD lies in the fact that there is a disparity in frequency among different senses of an ambiguous word. *Smoothing* is used to ensure that infrequent data or unseen data is treated properly. Class-based (Yarowsky, 1992) and similarity-based (Karov and Edelman, 1996; Karov and Edelman, 1998) models try to overcome data sparseness by generalizing over classes of words.

13.2 Attempt at Evaluation: Senseval

Evaluation is an important matter within the discipline of NLP in general, and in WSD in particular. To evaluate means to compare the results of a particular system with what is seen as correct solution to the problem. In WSD, sense-tagged corpora are needed for evaluation. So far, reliable evaluation data can only be produced through hand-annotation which is very time and expertise-intensive as well as dependent on the skills of the annotator(s).[1]

Another difficulty of evaluating WSD systems with regard to each other is that different lexicons with different sense inventories are used. This means that there is no basis on which to compare the systems. Also, different additional knowledge sources might be employed by different systems which does not facilitate comparison either.

A first attempt within WSD to setup a common task for several systems in order to allow for evaluation is Senseval. Senseval-1, held in 1998, was “the first open,

[1]A measure for the quality of hand-annotated text has been established, the *Inter-Tagger Agreement* (ITA). See (Kilgarriff, 1998a) for an extensive discussion over the production of Gold Standard datasets.

community-based evaluation exercise for WSD programs" in which 18 systems participated (Kilgarriff and Rosenzweig, 2000). The setup allowed for supervised and unsupervised systems to participate, and included a coarse and fine-grained level of sense distinctions.

Several choices regarding task design, corpus and dictionary used had to be made. The task was chosen to be a *lexical* task which means that only a (small) set of previously chosen ambiguous words is disambiguated. An *all-words* approach, in contrast, would mean annotating all ambiguous (content) words in a given corpus. The HECTOR lexical database (Atkins, 1993) was chosen for corpus and dictionary since this database had not been widely used in WSD before and was readily available. The results of Senseval-1 showed the state-of-the-art for supervised (fine-grained) WSD to be 78% correct. Unfortunately, no precise results on unsupervised systems are reported. It is only stated that for unsupervised systems "scores were both lower and more variable" (although of the 18 participating systems 10 were supervised and 8 were unsupervised).

After the success of Senseval-1, Senseval-2 was started in 2000, broadening the task to different languages, to a choice between lexical or all-words disambiguation, as well as to a more flexible framework (See (Edmonds and Cotton, 2001) for an overview).[2]

The results for the Senseval-2 English lexical sample task show a much lower state-of-the-art disambiguation rate for supervised (fine-grained) WSD, namely 64% correct. This amounts to a drop in performance of around 14% in comparison to the Senseval-1 results. According to (Kilgarriff, 2001) the difference is due to the different lexicon. For the Senseval-2 task WordNet was used as sense inventory. This choice was motivated by the fact that WordNet is very widely used (not only in WSD) and has become almost a *de facto* standard. The biggest drawback with using WordNet, however, is that some of the sense distinctions are not clear and/or well-motivated due to the fact that WordNet is organized around groups of words with similar meanings (so called *synsets*), and not around words (as in a dictionary). If the sense distinctions are not clear o start with, the task of disambiguating is obviously more difficult which explains the lower results.

In the context of Senseval-2, the first sense-tagged corpus for Dutch was made available (see (Hendrickx and van den Bosch, 2001) for a detailed description) which underlines the importance of Senseval for this project. Since the release of the data, new experiments are being conducted using real ambiguous words (see section 13.6)—in contrast to the preliminary experiments presented in the next section.

13.3 Preliminary Experiments

In this section, we will report on three preliminary experiments that have been carried out during the first phase of the project. They all used a supervised WSD algorithm (see section 13.3.2) which was trained on either the European Corpus

[2]The data for various languages is available from `http://www.sle.sharp.co.uk/senseval2/`.

Initiative (ECI) corpus of Dutch[3] or on the Senseval-1 corpus[4]. Since there is no disambiguated material available for the Dutch ECI corpus (which means that evaluation of results is not possible), we artificially created such data using pseudowords.

13.3.1 Pseudowords

The technique of pseudowords consists of introducing a form of artificial ambiguity in (untagged) corpora. First of all, two or more words, *sensewords*, are chosen. Training then takes place on the disambiguated corpus, collecting probabilities for the chosen sensewords. For testing, all occurrences of the sensewords are replaced by a non-existing word, a *pseudoword*. The goal is then to recover the correct senseword for every pseudoword introduced in the corpus.

Suppose we chose the sensewords ‘aantal’ and ‘tijd’ and combined them to form the pseudoword ‘aantijd’. The original sentences (1) and (2)—which are used in training as well as in evaluation—will then become test sentences (3) and (4).

(1) Hun *aantal* groeit en volgens justitie lijkt aan die groei geen einde te komen.

(2) Tot die *tijd* blijven de stellingen betrokken.

(3) Hun *aantijd* groeit en volgens justitie lijkt aan die groei geen einde te komen.

(4) Tot die *aantijd* blijven de stellingen betrokken.

Gale, Church, and Yarowsky (1992b) used pseudowords to overcome the “testing material bottleneck”, as well as Schütze (1992) and Schütze (1998), who tried to escape the need for hand-labeling using artificial ambiguous words for evaluation purposes.

13.3.2 Naive Bayes Classification

In the case of the preliminary experiments reported here, we chose to work with a *naive Bayes classifier* (Duda and Hart, 1973) because it is easy to implement, performs relatively well, is rather fast and is used fairly often.

In addition to that, a Bayes classifier uses only distributional information and contextwords to compute probabilities which corresponds to only using information which is available from the corpus itself without the need of any additional material, such as a dictionary or the like. The contextwords are assumed to be independent of position and of each other—they constitute a *bag of words*—which corresponds to the Bayes independence assumption.

First, the disambiguation algorithm is trained on part of the unambiguous corpus, attributing probabilities to the contextwords found to the right and the left of

[3]The ECI is a digitally available multilingual corpus distributed by ELSNET which contains material on a number of European languages, among others Dutch. See `http://www.elsnet.org/eci.html` for a complete listing of available languages and ordering information.

[4]Publicly available at `http://www.itri.brighton.ac.uk/events/senseval1/ARCHIVE/resources.html`.

the senseword(s) for various context window sizes. This is done using Bayes rule

$$P(s_k|c) = \frac{P(c|s_k)}{P(c)} P(s_k)$$

where s_k is sense k of ambiguous word w in context $c = \{c_1, \ldots, c_n\}$, the contextwords within the specified context window. "Training" as used here amounts to counting which senses are used most often in a given context.

Testing takes place on the ambiguous text where the algorithm selects the most probable senseword for each pseudoword according to Bayes decision rule

$$\text{Decide } s' \text{ if } P(s'|c) > P(s_k|c) \text{ for } s_k \neq s'$$

Finally, the computed sensewords are compared to the original sensewords in the disambiguated corpus and the percentage of correctly disambiguated instances of pseudowords is calculated.

Despite its relatively 'naive' approach, the Naive Bayes classifier performs relatively well, especially in comparison with other, more sophisticated approaches (Mooney, 1996; Escudero, Màrquez, and Rigau, 2000).

As has been pointed out in section 13.1, sparse data is a problem in corpus-based WSD. If a contextword has not been seen with a particular sense of an ambiguous word in the training data, the probability $P(v_j)$ of contextword v_j in the context of all senses s_k of ambiguous word w will be 0. This means that no choice can be made using the naive Bayes Classification algorithm explained above.

In such a case, smoothing techniques are applied. In the experiments described in sections 13.3.5, 13.3.3 and 13.3.4, a fixed penalty of $-\log 0.01$ (corresponding to a probability of $p = 0.01$) has been used. Possible extensions would be to use more sophisticated smoothing techniques, e.g. Good-Turing.

13.3.3 First Experiment: Varying Corpus Size

In a first experiment, we looked at the changes of performance in the classification algorithm used depending on corpus size. When working with statistical methods, changes in corpus size/training instances are expected to be reflected in changes of performance (Langley, Iba, and Thompson, 1992). The usual assumption is that the bigger the corpus the better the performance.

Settings: Corpus and Pseudowords

The corpus used in this experiment was the ECI Corpus of Dutch which contains approximately 3 Million words of raw text. The corpus includes transcripts of radio programs, newspaper articles, magazine issues, and some technical texts.

Choosing high frequency nouns, six pseudowords were created, four of which consist of two sensewords and two of which consist of 3 sensewords. Table 13.1 gives an overview over the sensewords chosen as well as their frequency and the frequency baseline.[5]

[5]There are two possibilities to calculate the baseline for WSD Systems: the *random baseline*, which chooses a possible sense at random, or the *frequency baseline*, where the most frequent sense is always chosen. Usually the frequency baseline lies higher than the random baseline, which is why it is a more representative lower bound.

Pseudow.	Senseword 1		Senseword 2		Senseword 3		Baseline
	word	frequ.	word	frequ.	word	frequ.	
aantijd	aantal	1995	tijd	1747			53.31%
lagem	land	2991	gemeente	1018			74.60%
nedmin	nederland	2675	minister	3155			54.11%
prespol	president	2356	politie	1568			60.04%
neduir	nederland	2675	duitsland	719	irak	2818	45.36%
plonbe	plan	1059	onderwijs	960	beleid	908	36.18%

Table 13.1: Overview Pseudowords

Underlying Assumptions

In the experiment described, we departed from two underlying assumptions: Topic coherence and ‘All information’. The idea of topic (or discourse) coherence states that words usually keep the same sense within a paragraph or document (Gale, Church, and Yarowsky, 1992a; Yarowsky, 1993).[6] The size of the context window used was thus restrained to paragraphs, which means that if the window size on either side was bigger than the paragraph boundary, everything beyond that boundary was not taken into account.[7]

Furthermore, no stoplist was used in the reported experiments. One of the working hypotheses was to test whether taking into consideration all available context information including function words could produce good results. In the case of nouns with different articles, for instance, working with a stoplist would definitely be counter-productive. Also, for words with a very different syntactic distribution, like ‘nederland’ and ‘minister’, prepositions and articles are good indicators for a certain ‘sense’.

Results and Evaluation

In the reported experiment, results were ten times cross-validated. The context window was restricted to 3 words to the left and the right of the pseudoword. We take a similar approach to (Chodorow, Leacock, and Miller, 2000) choosing a fixed context window size of ± 3. Similar results can be observed when different context sizes are used.

The results obtained (see table 13.2) clearly show that more training instances do help improve the performance of the naive Bayes classification algorithm used. The overall performance of the algorithm is quite good, especially considering the fact that the results are solely based on statistical information.

[6]Krovetz (1998) has shown that this is only (partially) true for homonymous senses, but is not the case for polysemous words.

[7]There was a big variation in paragraph lengths (1-15 sentences). It is not quite clear yet what sort of noise is introduced through this fact.

Pseudoword	Baseline	0.5M Words	1.5M Words	3M Words
aantijd	53.31	80.32	84.08	**84.97** (+31.66)
lagem	74.60	78.54	80.08	**82.65** (+08.05)
nedmin	54.11	84.45	83.18	**85.02** (+30.91)
prespol	60.04	73.99	79.09	**83.21** (+23.17)
neduir	45.36	65.83	66.38	**70.83** (+25.47)
plonbe	36.18	58.32	67.25	**67.70** (+31.52)

Table 13.2: Results with varying corpus size (in %), optimal performance per row in bold

Pseudoword	Baseline	all	0.6	0.7	0.8	0.9	1
aantijd	53.31	84.97	**85.03**	84.97	79.83	76.64	72.11
lagem	74.60	82.65	82.65	82.62	**82.88**	81.83	81.60
nedmin	54.11	85.02	**85.09**	84.25	82.85	71.66	69.49
prespol	60.04	83.21	**83.43**	81.97	81.15	79.23	78.63

Table 13.3: Results with varying thresholds (in %), optimal performance per row in bold

13.3.4 Second Experiment: Varying Thresholds for Contextwords

In a different experiment, we looked at the use of contextwords. The main idea was to only use contextwords of a certain informative value (expressed through placing a threshold on the probability of each contextword) and to find the cutoff at which the amount of data still used in the disambiguation process and the informative value converge. The thresholds represent how well a particular contextword helps to disambiguate an ambiguous word/pseudoword. A threshold of 1.0 means that a contextword is only used for disambiguation if the probability of contextword v_j given sense k of ambiguous word w is 1 ($p(v_j|s_k) = 1$).

The corpus and overall settings were the same as in the experiment reported in section 13.3.3. Only the four pseudowords consisting of two senses were used.

Results and Evaluation

As the results in table 13.3 show there is no clear cutoff value at which the performance of the algorithm improves for all pseudowords. A tendency can be observed that using a threshold of 0.6 (which means that all contextwords are used *except* those which are (almost) equally likely to occur with both senses of a given ambiguous word) works best.

13.3.5 Third Experiment: Pseudowords vs. Real Ambiguous Words

In the last experiments reported, we investigated whether disambiguating pseudowords is comparable to the task of disambiguating real ambiguous words and we reached the conclusion that these two tasks are *not* identical (Gaustad, 2001).

Outline of the Problem

The idea to compare the task of disambiguating real ambiguous words to disambiguating artificially ambiguous words arose from our work on supervised WSD for Dutch. Since there are no sense-tagged corpora available for Dutch, another means of testing algorithms has to be used. An obvious solution is the use of pseudowords: they are easily created, only raw text material is needed and any supervised algorithm can be tested. The one question that remained unanswered was whether using pseudowords would yield results comparable to real WSD and whether the seemingly 'easy way out' could really be seen as equivalent to the disambiguation of real ambiguous words.

Unfortunately, there has not been a lot of work on pseudowords and, to the best of our knowledge, no work at all on their usefulness in testing word sense disambiguation systems. The major problem involved in this comparison is to find a valid setting for a comparison: the elements to be compared—pseudowords and real ambiguous words —are too different from each other to be compared directly. Schütze (1998) explains it in the following way: "[The better performance on pseudowords] can be explained by the fact that pseudowords have two focused senses—the two word pairs they are composed of." Real ambiguous words, on the other hand, consist of sub-senses that are difficult to identify for humans as well as for computers.

Way of Proceeding

A direct comparison of the task of WSD and the task of disambiguating pseudowords is not possible. The only way to compare these two tasks is to *indirectly* compare their results on the same corpus, using the same algorithm and general settings. The comparison does have its limitations: Although we use the same settings for both tasks, the difference between them lies in the actual words (or pseudowords) to be disambiguated. There is no measure to express their differences or similarities. This is precisely why there is no possibility of a direct comparison.

We decided to proceed in two steps. First, real ambiguous words were chosen from the Senseval-1 corpus making use of the dictionary entries as well as the training and testing material provided. Only nouns which were not ambiguous regarding part of speech and for which there was training data were taken into account.

In a second step, we chose the sensewords of a pseudoword according to the frequency distribution of the senses of the real ambiguous words that were tested. Among the possible sensewords that exhibited the same frequency distributions as the real ambiguous words and which fulfilled the constraint of having approximately the same baseline, an arbitrary selection was made.

If the results of this second task are significantly different from the results of the first task on the same corpus, this will show that the results involving pseudowords depend entirely on the choice of sensewords. This means that the disambiguation of pseudowords is *not* identical to the real WSD task. Note that if one does not have access to sense-tagged corpora, no information about the distribution of the senses of real ambiguous words is available, which means that it is not really a comparable setup!

Settings: Corpus and Ambiguous Words/Pseudowords

The corpus used in this experiment are the English Senseval-1 resources. The advantage of using this material is that it is (lexically) sense-tagged for a number of real ambiguous words which means that the evaluation data for real ambiguous words is at hand. Furthermore, there have been numerous publications on the construction of the material, on choices made regarding annotation, on inter-annotator agreement, etc. ((Kilgarriff, 1998b; Kilgarriff and Rosenzweig, 2000), and see also section 13.2), which allow for a thorough understanding of the real world disambiguation task. This is an important precondition to being able to extensively compare this task to nearly the same task using pseudowords.

The perhaps most important factor in this comparison is the choice of elements of comparison, in this case the ambiguous words and the sensewords chosen to constitute the different pseudowords.

The choice of ambiguous words depended, on the one hand, on the available Senseval-1 material (evaluation data). On the other hand, we only selected nouns which were not ambiguous in part-of-speech.[8] No stemming was used. The ambiguous words and their senses[9] chosen for the experiments can be seen in table 13.4[10].

The main criteria for choosing the sensewords constituting the pseudowords were their frequency in the corpus as well as their part of speech. For the comparison with each ambiguous word, five pseudowords were made up. The distribution of these pseudowords' sensewords was chosen to be as similar as possible to the distribution of the different senses of the ambiguous words. An overview (including frequencies) of the pseudowords and the corresponding ambiguous word is given in table 13.4.

Results and Evaluation

The results in table 13.5 clearly show that the performance of the naive Bayes classification algorithm used is significantly better on pseudowords than on real ambiguous words. A possible reason for this is the relatedness of sense distinctions in real ambiguous words whereas the sensewords that constitute pseudowords have two very clearly distinct senses.

A possible explanation for the fact that the performance on real ambiguous words is considerably bad—it constantly fails to reach the baseline—is that there is not enough training data. Note that the baseline of most ambiguous nouns in the Senseval-1 corpus is relatively high which means that one sense accounts for most occurrences of the ambiguous word. This makes the disambiguation task

[8]A number of ambiguous words in the Senseval-1 material had to be simultaneously part-of-speech and lexically disambiguated, e.g. *bet*, *giant*, *promise*. There were also cases with no training material provided (*disability*, *hurdle*, *rabbit*, *steering*) which were not taken into account given that we worked with a supervised algorithm.

[9]The senses were taken from the Senseval-1 dictionary entries. Only the coarse-grained distinctions were taken into account.

[10]Since the sense *hairsh* does not occur in the testing data, we decided to only consider two senses for *shirt* and, consequently, for the pseudowords.

	Amb./Ps.word	Senses/S.words	Freq. train	Freq. test	Baseline
Ambig. word	accident	crash	1058	248	92.88%
		chance	178	19	
Pseudowords	timwe	time	722	306	91.90%
		weekend	73	27	
	yeatra	year	708	307	92.47%
		traffic	86	25	
	peolang	people	673	268	92.10%
		language	54	23	
	woan	world	422	187	92.12%
		animal	39	16	
	goveq	government	396	184	92.35%
		equipment	31	15	
Ambig. word	behavior	social	969	267	95.70%
		of thing	29	12	
Pseudowords	peostan	people	673	268	93.40%
		standards	41	19	
	tima	time	722	306	95.33%
		machine	49	15	
	yeagro	year	708	307	95.34%
		growth	58	15	
	wodat	world	422	187	94.92%
		data	36	10	
	gopay	government	396	181	95.26%
		payment	30	9	
Ambig. word	excess	aglut	103	108	58.06%
		of or after poss	65	67	
		surplus	10	9	
		too much	73	2	
Pseudowords	womuconba	world	422	187	58.62%
		music	231	97	
		concert	43	16	
		battle	42	19	
	gopoemch	government	396	184	57.64%
		police	218	98	
		empire	37	16	
		champion	45	19	
	dacipapro	day	373	161	57.71%
		city	211	83	
		palace	37	16	
		protection	45	19	
	pemanora	people	673	268	58.64%
		man	377	154	
		noise	33	16	
		railway	33	19	
	heterite	head	349	150	58.37%
		team	162	72	
		river	42	16	
		technology	34	19	
Ambig. word	shirt	t-shirt	132	73	57.06%
		garment	336	105	
Pseudowords	schoclu	school	178	87	59.02%
		club	140	72	
	mastre	market	190	89	58.55%
		street	158	63	
	cimon	city	211	83	58.04%
		month	130	60	
	coufam	country	201	91	57.96%
		family	117	66	
	wogia	women	189	91	58.33%
		giants	140	65	

Table 13.4: Overview ambiguous words and corresponding pseudowords

	Basel.	Results	Difference
accident	92.88	84.45	**- 8.43**
timwe	91.90	91.56	- 0.34
yeatra	92.47	91.77	- 0.70
peolang	93.10	91.88	- 0.59
woan	92.12	93.44	+ 0.97
goveq	92.35	91.33	- 1.14
mean			- 0.40 [± 0.89]
behaviour	95.70	84.95	**-10.75**
peostan	93.40	92.99	- 0.41
tima	95.33	95.64	+ 0.31
yeagro	95.34	94.04	- 1.30
wodat	94.92	93.79	- 1.13
gopay	95.26	96.36	+ 1.10
mean			- 0.29 [± 1.24]
excess	58.06	50.35	**- 7.71**
womuconba	58.62	71.86	+13.24
gopoemch	57.64	72.92	+15.28
dacipapro	57.71	73.98	+16.27
pemanora	58.64	73.00	+14.36
heterite	58.37	74.39	+16.02
mean			+15.03 [± 1.55]
shirt	58.98	57.50	**- 1.48**
schoclu	59.02	72.79	+13.77
mastre	58.55	74.83	+16.28
cimon	58.04	78.69	+20.65
coufam	57.96	63.91	+ 5.95
wogia	58.33	72.22	+13.89
mean			+14.1 [± 6.6]

Table 13.5: Pseudowords vs. Real Ambiguous Words: Results (in %)

comparatively harder and might be a possible explanation for the bad performance on real ambiguous words.

We conclude from the results that the task of disambiguating pseudowords is comparable only in a limited way to the task of disambiguating real ambiguous words. The results on pseudowords will usually be better which might lead to false assumptions about the performance of a given algorithm on the real problem.

The results obtained from disambiguating artifical ambiguous words differ greatly from the results of real ambiguous words. This indicates that pseudowords cannot be taken as a substitute for testing with real ambiguous words.

Testing of WSD algorithms is very difficult without evaluation data. The assumption that artificially created ambiguous words are a good substitute for real ambiguous words is *not* valid, as has been shown by the experiment reported in section 13.3.5. Thus the initial problem—wanting to test algorithms for languages without sense tagged corpora—remains.

13.4 Research Questions

Based on the research review and the conducted preliminary experiments, the following questions concerning WSD are raised.

1. What kind of linguistic information is most useful for WSD?
2. How can one successfully combine statistical approaches to WSD with linguistic information?
3. How can the interplay between corpus, linguistic information sources and disambiguation proper be optimised?

The research plans are to implement a WSD algorithm which makes use of different types of linguistic information (e.g. part-of-speech, dependency relations, selectional restrictions) in combination with statistical methods.

Statistical approaches have proved to be successful and rather efficient, but intuitively one would think that the addition of linguistic information should lead to increases in performance[11]. In WSD, it has not yet been systematically investigated whether this is the case.

A major question that will be investigated is what kind of linguistic information is most useful for word sense disambiguation (see section 13.5). Also, different linguistic information might be useful depending on the syntactic category and other characteristics of the target word, as different disambiguation strategies might be needed for different groups or classes of target words.

It seems reasonable to include information made available by the Alpino grammar and parser developed in the context of the PIONIER-project described in this report. This 'collaboration' also facilitates the integration of the developed WSD system into the final NLP tool for Dutch.

[11]For example, Wilks and Stevenson (1997) state that: "Our intuition is that word sense disambiguation can be most effectively carried out combining [different, orthogonal] knowledge sources."

13.5 Linguistic Information for WSD

There are different sources of knowledge which can be used in WSD. As has been said above, the use of linguistic knowledge for a WSD system has not been thoroughly investigated yet, although it might lead to significant increases in performance.[12] Types of linguistic knowledge to be possibly used in WSD are detailed in the following sections. The linguistic information will be used in addition to statistical information (e.g. bigrams, see (Pedersen, 2001)) and a statistics-based algorithm (see section 13.3.2).

13.5.1 Morphological Information

Lemmatising Lemmatising is the process of stripping words from morphological information and only keeping root forms. Generalizing over different morphological realizations of words might be an advantage when working with verbs. For nouns (and probably also adjectives and adverbs) it might be disadvantageous not to include morphological information. Important facts about e.g. senses which are only applicable in case of plural might be lost. There is also the danger of overgeneralization which might lead to not being able to distinguish between different words anymore because they are shortened to the same basic form.

13.5.2 Syntactic Information

Part-of-speech (POS) POS tags contain syntactic information at word level. They can be used in the following way. Senses that do not match the POS of the ambiguous word in context can be eliminated. This reduces the number of senses that have to be considered in the disambiguation process.

Subcategorization Frames Subcategorization frames provide valency information for different categories of words (nouns, verb, adjectives). They are typically associated with selectional restrictions, i.e. restrictions on the meaning of grammatical complements, and thus they could be useful e.g. in determining which senses of nouns are appropriate as complements of a certain sense of a verb given a particular syntactic construction, and vice versa.

Dependency Relations Dependency relations hold between constituents in a sentence which are dependent on each other. Since dependency structures do not necessarily reflect surface (syntactic) structure, they are a valuable source of information when full parses are unavailable. Being able to identify dependencies between constituents could be used to determine which words might contain more important disambiguation clues than others.

[12]There are attempts to integrate linguistic knowledge into WSD systems (see e.g. Ng and Lee's (1996) exemplar-based approach which uses POS, stemming and verb-object syntactic relations), but, to the best of our knowledge, no systematic research has been conducted yet.

13.5.3 Semantic Information

WordNet WordNet is a lexicon where the similarity between words is represented hierarchically. Lexical information is organized by semantic properties. In addition to semantic information, relations, such as for instance hyponymy or hyperonymy, between words and concepts are also defined. Different approaches to WSD have used information from WordNet (Leacock, Chodorow, and Miller, 1998; Hawkins, 1999). An asset is that WordNet provides semantic information attached to lexical items. One of the difficulties with the architecture of this lexicon, on the other hand, is the fact that different POS are contained in separate hierarchies. Also, related words cannot be clustered easily since they might be classified far away from each other (referred to as 'tennis problem', see section 3.5 in (Hawkins, 1999)).

Selectional Restrictions Selectional restrictions are restrictions on the meaning of grammatical complements. In (Resnik, 1993; Resnik, 1997) selectional restrictions are used to resolve syntactic ambiguity. Resnik's hypothesis is that many lexical relationships reflect underlying conceptual relationships and that statistical disambiguation strategies should take those into account. Prior to using selectional restrictions the grammatical links between words must already have been identified. A drawback of this approach is that selectional restrictions are only useful for the resolution of broad sense ambiguity since fine-grained senses often belong, or are used with, the same semantic class.

13.5.4 Pragmatic Information

Topical information A thesaurus usually contains information about subject areas, often called pragmatic codes. Using these codes can help to optimize the choice of senses: the goal is to achieve the greatest overlap of pragmatic codes within a text with paragraphs as basic units (Wilks and Stevenson, 1997; Wilks and Stevenson, 1998). Yarowsky (1992), for instance, uses Roget's thesaurus to identify salient contextwords that appear with an ambiguous word belonging to a certain category. These can then be used to resolve ambiguity since they provide evidence for a particular category.

13.6 Follow-Up Experiments

13.6.1 Experiments on Senseval-2 Data

The experiments described in the first part of this chapter on word sense disambiguation have shown that the use of pseudowords to investigate WSD is not a viable option. Since the sense-tagged dataset for Dutch has been made available in the context of Senseval-2, we have started a first batch of experiments systematically investigating the influence of different sources of linguistic information on disambiguation accuracy.

The statistical classifier used in these experiments is a maximum entropy classifier. Maximum entropy is a general technique for estimating probability distributions from data. If nothing about the data is known, it involves selecting the most uniform distribution where all events have equal probability. In other words, it means selecting the distribution which maximizes the entropy.

If data is available, labeled training data is seen as a number of features which are used to derive a set of constraints for the model. This set of constraints characterizes the class-specific expectations for the distribution. So, while the distribution should maximize the entropy, the model should also satisfy the constraints imposed by the labeled training data. A maximum entropy model is thus the model with maximum entropy of all models that satisfy the set of constraints derived from the training data.

The maximum entropy model is built using the following formula:

$$p(c|x) = \frac{1}{Z} \exp\left(\sum_i \lambda_i f_i(x,c)\right)$$

where $f_i(x,c)$ is the number of times feature i is used to find class c for event x, and the weights λ_i are chosen to maximize the likelihood of the training data and maximize the entropy of p.

The main advantage of maximum entropy modeling over a Naive Bayes classifier is that dependencies between different features are taken into account. Also, heterogeneous and overlapping sources of information can be integrated into the statistical model. Furthermore, good results have been produced in other areas of NLP research using maximum entropy techniques (Berger, Pietra, and Pietra, 1996).

In order to be able to systematically investigate the influence of different sources of linguistic information in WSD, we set up the following scheme for the first experiments on the Senseval-2 data for Dutch.

First, the Senseval-2 data is tokenized, lemmatized and part-of-speech-tagged (using a Brill-style tagger for Dutch (Drenth, 1997) with only the main categories of the WOTAN tag set (Berghmans, 1994)). In a second step, linguistic information is tested on its value for WSD. The kinds of information included in these first experiments will be: the ambiguous word itself, the lemma of the ambiguous word, its part-of-speech, context words to the left and the right of the ambiguous word (either full word-forms or lemmas), as well as dependency relations for heads and dependents. Varying the amount of information included, different feature sets are built. The ambiguous word and its lemma constitute the core information which is always included in the feature set. Every feature set is tested using the maximum entropy classifier and the classification accuracy is evaluated against the frequency baseline (see footnote 5 in section 13.3.3).

13.6.2 Bootstrapping

The production of sense-tagged training material for supervised algorithms is very time- and expertise-intensive. The same is true for evaluation data for any algorithm, be it supervised or unsupervised. In order to overcome this problem partly, *bootstrapping* can be used.

Bootstrapping means that a small corpus is sense-tagged by hand and used as seed information for a classifier. Iteratively, large amounts of unlabeled data are then labeled using the seed information, and the newly labeled data is in turn used as input to build a new classifier. In this way, labeled data can be acquired quickly. Its quality is assured through hand-correction (which is a lot less time-consuming than hand-labeling all the data). This method has been applied to WSD with good results (Yarowsky, 1995).

In the context of this project, the idea is to use Alpino to annotate sentences with a seed collection of senses, hand correct the output, increase the seed collection and retrain the classifier using the newly labeled data set and the larger seed collection. This incremental method of semi-automatic annotation will provide us with more training and testing data for the final WSD system. Also, this method is a lot less time and labor consuming than hand-annotation

References

Abeille, Anne. 1995. The flexibility of french idioms: a representation with lexicalized tree adjoining grammar. In Martin Everaert, Erik-Jan van der Linden, Andre Schenk, and Rob Schreuder, editors, *Idioms: Structural & Psychological Perspectives*. Lawrence Erlbaum Associates, New Jersey.

Abney, Steven P. 1997. Stochastic attribute-value grammars. *Computational Linguistics*, 23:597–618.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman. 1974a. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company.

Aho, Alfred V., John E. Hopcroft, and Jeffrey D. Ullman. 1974b. *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers. Principles, Techniques and Tools*. Addison Wesley.

Angluin, D. 1988. Identifying languages from stochastic examples. Technical Report 614, Yale University.

Atkins, Sue. 1993. Tools for computer-aided corpus lexicography: The Hector project. *Acta Linguistica Hungarica*, 41:5–72.

Baayen, R. H., R. Piepenbrock, and H. van Rijn. 1993. *The CELEX Lexical Database (CD-ROM)*. Linguistic Data Consortium, University of Pennsylvania, Philadelphia, PA.

Balay, Satish, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Lois Curfman McInnes, and Barry F. Smith. 2001. PETSc home page. http://www.mcs.anl.gov/petsc.

Balay, Satish, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. 1997. Efficienct management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press.

Balay, Satish, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. 2002. PETSc users manual. Technical Report ANL-95/11–Revision 2.1.2, Argonne National Laboratory.

van der Beek, Leonoor, Gosse Bouma, Robert Malouf, and Gertjan van Noord. 2002. The alpino dependency treebank. In *Computational Linguistics in the Netherlands*. to appear.

Beesley, Kennneth R. and Lauri Karttunen. 2002. *Finite-State Morphology: Xerox Tools and Techniques*. Studies in Natural Language Processing. Cambridge University Press.

Belz, Anja. 2000. Multi-syllable phonotactic modelling. In Lauri Karttunen Jason Eisner and Alain Thériault, editors, *Proceedings of SIGPHON 2000: Finite-State Phonology*. Luxembourg.

Benson, Steven, Lois Curfman McInnes, and Jorge Moré. 2000. GPCG: A case study in the performance and scalability of optimization algorithms. Technical Report ANL/MCS-P768-0799, Argonne National Laboratory. http://www.mcs.anl.gov/home/more/papers/gpcg.ps.gz.

Benson, Steven J., Lois Curfman McInnes, Jorge J. Moré, and Jason Sarich. 2002. TAO users manual. Technical Report ANL/MCS-TM-242–Revision 1.4, Argonne National Laboratory.

Berger, Adam, Stephen Della Pietra, and Vincent Della Pietra. 1996. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–72.

Berger, Adam L., Stephen A. Della Pietra, and Vincent J. Della Pietra. 1996. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71.

Berghmans, Johan T. 1994. WOTAN—een automatische grammaticale tagger voor het Nederlands. Master's thesis, K.U. Nijmegen.

Bird, Steven and T. Mark Ellison. 1994. One-level phonology: Autosegmental representations and rules as finite automata. *Computational Linguistics*, 20(1):55–90.

Blaheta, Don and Mark Johnson. 2001. Unsupervised learning of multi-word verbs. In *39th Annual Meeting and 10th Conference of the European chapter of the Association for Computational Linguistics (ACL39)*, pages 54–60, Toulouse, France. CNRS.

Blattner, Meera and Tom Head. 1977. Single-valued a-transducers. *Journal of Computer and System Sciences*, 15(3):328–353.

Boros, M., W. Eckert, F. Gallwitz, G. Görz, G. Hanrieder, and H. Niemann. 1996. Towards understanding spontaneous speech: Word accuracy vs. concept accuracy. In *Proceedings of the Fourth International Conference on Spoken Language Processing (ICSLP 96)*, Philadelphia.

Bouma and Kloosterman. 2002. Querying dependency treebanks in XML. In *Proceedings of the Third international conference on Language Resources and Evaluation (LREC)*, pages 1686–1691, Gran Canaria, Spain.

Bouma, Gosse. 2000. Argument realization and Dutch R-pronouns: Solving Bech's problem without movement or deletion. In Ronnie Cann, Claire Grover, and Philip Miller, editors, *Grammatical Interfaces in HPSG*. CSLI Publications, pages 51–76.

Bouma, Gosse. 2001a. Extracting dependency frames from existing lexical resources. In *Proceedings of the NAACL Workshop on WordNet and Other Lexical Resources: Applications, Extensions and Customizations*, Somerset, NJ. Association for Computational Linguistics.

Bouma, Gosse. 2001b. Finite state methods for hyphenation. In Lauri Karttunen, Kimmo Koskenniemi, and Gertjan van Noord, editors, *Finite State Methods in Natural Language Processing, Extended Abstracts, ESSLLI Workshop*, pages 29–33, Helsinki.

Bouma, Gosse, Frank van Eynde, and Dan Flickinger. 2000. Constraint-based lexica. In F. van Eynde and D. Gibbon, editors, *Lexicon Development for Speech and Language Processing*. Kluwer, pages 43–76.

Bouma, Gosse, Rob Malouf, and Ivan Sag. 2001. Satisfying constraints on adjunction and extraction. *Natural Language and Linguistic Theory*, 19:1–65.

Bouma, Gosse and Gertjan van Noord. 1998. Word order constraints on verb clusters in German and Dutch. In Erhard Hinrichs, Tsuneko Nakazawa, and Andreas Kathol, editors, *Complex Predicates in Nonderivational Syntax*. Academic Press, New York, pages 43–72.

Bouma, Gosse, Gertjan van Noord, and Robert Malouf. 2001. Wide coverage computational analysis of Dutch. In W. Daelemans, K. Sima'an, J. Veenstra, and J. Zavrel, editors, *Computational Linguistics in the Netherlands 2000*.

Bouma, Gosse and Begoña Villada Moirón. 2002. Corpus-based acquisition of collocational prepositional phrases. In *Computational Linguistics in the Netherlands*. to appear.

Brandt Corstius, H. 1978. *Computer-taalkunde*. Coutinho, Muiderberg.

Brants, Thorsten. 2000. TnT—a statistical part-of-speech tagger. In *Proceedings of the 6th Applied Natural Language Processing Conference ANLP-2000*, Seattle, WA.

Breidt, Elisabeth, Frederique Segond, and Giuseppen Valetto. 1996. Local grammars for the description of multi-word lexemes and their automatic recognition in texts. In *COMPLEX96*, Budapest.

Brent, Michael. 1993. From grammar to lexicon: Unsupervised learning of lexical syntax. *Computational Linguistics*, 19(2):243–262.

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

Briscoe, Ted and John Carroll. 1997. Automatic extraction of subcategorization from corpora. In *Proceedings of the 5th ACL Conference on Applied Natural Language Processing*, pages 356–363, Washington, DC.

Briscoe, Ted and John Carroll. 1998. Can subcategorization probabilities help a statistical parser? In *Proceedings of the ACL/SIGDAT workshop in Very Large Corpora*, Montreal.

Calder, Jo. 2000. Thistle and interarbora. In *Proceedings of the 18th International Conference on Computational Linguistics (COLING)*, pages 992–996, Saarbrücken.

Campbell, L.L. 1970. Equivalence of Gauss's principle and minimum discrimination information estimation of probabilities. *Annals of Mathematical Statistics*, 41:1011–1015.

Caraballo, Sharon A. and Eugene Charniak. 1998. New figures of merit for best-first probabilistic chart parsing. *Computational Linguistics*, 24(2).

Carpenter, Bob. 1992. *The Logic of Typed Feature Structures*. Cambridge University Press, New York.

Carrasco, Forcada, and Santamaria. 1996. Inferring stochastic regular grammars with recurrent neural networks. In *ICGI: International Colloquium on Grammatical Inference and Applications*.

Carrasco, Rafael C. and Mikel L. Forcada. 2002. Incremental construction and maintenance of minimal finite-state automata. *Computational Linguistics*, 28(2), June.

Carrasco, Rafael C. and Jose Oncina. 1999. Learning deterministic regular grammars from stochastic samples in polynomial time. *Theoretical Informatics and Applications*, 33(1):1–19.

Carroll, Glenn and Mats Rooth. 1998. Valence induction with a head-lexicalized PCFG. In *Proceedings of the 3rd Conference on Empirical Methods in Natural Language Processing*, Granada, Spain.

Carroll, J. and E. Briscoe. 1996. Apportioning development effort in a probabilistic LR parsing system through evaluation. In *Proceedings of the ACL SIGDAT Conference on Empirical Methods in Natural Language Processing*, pages 92–100, University of Pensylvania, Philadelphia, PA.

Carroll, John, Ted Briscoe, and Antonio Sanfilippo. 1998. Parser evaluation: A survey and a new proposal. In *Proceedings of the first International Conference on Language Resources and Evaluation (LREC)*, pages 447–454, Granada, Spain.

Carter, David. 1997. The TreeBanker: A tool for supervised training of parsed corpora. In *Proceedings of the ACL Workshop on Computational Environments For Grammar Development And Linguistic Engineering*, Madrid.

Charniak, E., G. Carroll, J. Adcock, A. Cassandra, Y. Gotoh, J. Katz, M. Littman, and J. McCann. 1996. Taggers for parsers. *Artificial Intelligence*, 85(1-2).

Chen, Stanley F. and Ronald Rosenfeld. 1999. A gaussian prior for smoothing maximum entropy models. Technical Report CMU-CS-99-108, Computer Science Department, Carnegie Mellon University.

Chi, Zhiyi. 1998. *Probability models for complex systems.* Ph.D. thesis, Brown University.

Chodorow, Martin, Claudia Leacock, and George Miller. 2000. A topical/local classifier for word sense identification. *Computers and the humanities*, 34(1-2):115–120.

Church, Kenneth W. and Ramesh Patil. 1982. Coping with syntactic ambiguity or how to put the block in the box on the table. *Computational Linguistics*, 8:139–149.

Church, Kenneth Ward and Patrick Hanks. 1990. Word association norms, mutual information & lexicography. *Computational Linguistics*, 16(1):22–29.

Collins, Michael. 1999. *Head-Driven Statistical Models for Natural Language Parsing.* Ph.D. thesis, University Of Pennsylvania.

Corley, Steffan, Martin Corley, Frank Keller, Matthew W. Crocker, and Shari Trewin. 2001. Finding syntactic structure in unparsed corpora. *Computers and the Humanities*, 35(2):81–94.

Culicover, Peter W. 1976. *Syntax.* Academic press, New York.

Daciuk, Jan. 1998. *Incremental Construction of Finite-State Automata and Transducers, and their Use in the Natural Language Processing.* Ph.D. thesis, Technical University of Gdańsk.

Daciuk, Jan. 2000a. Experiments with automata compression. In M. Daley, M. G. Eramian, and S. Yu, editors, *Conference on Implementation and Application of Automata CIAA'2000*, pages 113–119, London, Ontario, Canada, July. University of Western Ontario.

Daciuk, Jan. 2000b. Finite-state tools for natural language processing. In *COLING 2000 Workshop on Using Tools and Architectures to Build NLP Systems*, pages 34–37, Luxembourg, August.

Daciuk, Jan. 2001. Computer assisted enlargement of morphological dictionaries. In Lauri Karttunen, Kimmo Koskenniemi, and Gertjan van Noord, editors, *Proceedings of Finite State Methods in Natural Language Processing 2001, 13th European Summer School in Logic, Language and Information ESSLLI 2001*, Helsinki, Finland, August.

Daciuk, Jan. 2002. Comparison of construction algorithms for minimal, acyclic, deterministic, finite-state automata from sets of strings. In *Seventh International Conference on Implementation and Application of Automata CIAA '2002*, Tours, France, July.

Daciuk, Jan, Stoyan Mihov, Bruce Watson, and Richard Watson. 2000. Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1):3–16, April.

Daciuk, Jan and Gertjan van Noord. 2001. Finite automata for compact representation of language models in nlp. In Burce Watson and Derick Wood, editors, *Proceedings of the 6th Conference on Implementations and Applications on Automata (CIAA)*, pages 45–55, Pretoria, South Africa.

Daelemans, Walter and Antal van den Bosch. 1992. Generalization performance of backpropagation learning on a syllabification task. In *Connectionism and Natural Language Processing. Proceedings Third Twente Workshop on Language Technology*, pages 27–38.

Dalrymple, Mary, Stuart M. Shieber, and Fernando C.N. Pereira. 1991. Ellipsis and higher order unification. *Linguistics and Philosophy*, 14:399–452.

Darroch, J. and D. Ratcliff. 1972. Generalized iterative scaling for log-linear models. *Ann. Math. Statistics*, 43:1470–1480.

Dassow, Jürgen, Gheorge Paun, and Arto Salomaa. 1997. Grammars with controlled derivations. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages Vol.2 Linear Modeling: Background and Application.* Springer, pages 101–154.

Della Pietra, Stephen, Vincent Della Pietra, and John Lafferty. 1997. Inducing features of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19:380–393.

Deming, W.E. and F.F. Stephan. 1940. On a least squares adjustment of a sampled frequency table when the expected marginals are known. *Annals of Mathematical Statistics*, 11:427–444.

Dolan, Elizabeth D. and Jorge Moré. 2000. Benchmarking optimization software with COPS. Technical Report ANL/MCS-246, Argonne National Laboratory. http://www-unix.mcs.anl.gov/ more/cops/bcops.ps.gz.

Dolan, Elizabeth D. and Jorge J. Moré. 2002. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91:201–213.

Drenth, Erwin W. 1997. Using a hybrid approach towards Dutch part-of-speech tagging. Master's thesis, Alfa-Informatica, University of Groningen.

Duda, R. O. and P. E. Hart. 1973. *Pattern Classification and Scene Analysis.* John Wiley & Sons, New York.

Dunning, Ted. 1993. Accurate methods for the statistics of surprise and coincidence. *Computational linguistics*, 19(1):61—74.

Dupont, Pierre. 1997. Grammatical inference: Formal and heuristic methods. presentation slides, May.

Edmonds, Philip and Scott Cotton. 2001. Senseval-2: Overview. In *Proceedings of Senseval-2, Second International Workshop on Evaluating Word Sense Disambiguation Systems*, pages 1–5, Toulouse.

Eisner, Jason. 1997. Efficient generation in primitive optimality theory. In *35th Annual Meeting of the Association for Computational Linguistics and 8th Conference of the European Chapter of the Association for Computational Linguistics*, pages 313–320.

Ellison, Mark T. 1994. Phonological derivation in optimality theory. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING)*, pages 1007–1013, Kyoto.

Escudero, Gerard, Lluís Màrquez, and German Rigau. 2000. On the portability and tuning of supervised word sense disambiguation systems. Technical Report LSI-00-30, Software Department (LSI), Technical University of Catalonia (UPC).

Evert, Stefan and Brigitte Krenn. 2001. Methods for the qualitative evaluation of lexical association measures. In *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics*, Toulouse, France.

van Eynde, Frank. 1999. Major and minor pronouns in Dutch. In Gosse Bouma, Erhard W. Hinrichs, Geert-Jan M. Kruijff, and Richard T. Oehrle, editors, *Constraints and Resources in Natural Language Syntax and Semantics*. CSLI Publications, Stanford, pages 137–152.

Firoiu, Laura, Tim Oates, and Paul R. Cohen. 1998a. Learning deterministic finite automaton with a recurrent neural network. In *Proceedings of the 4th International Colloquium on Grammatical Inference - ICGI '98*, volume 1433, pages 90–101. Springer-Verlag.

Firoiu, Laura, Tim Oates, and Paul R. Cohen. 1998b. Learning regular languages from positive evidence. In *Proceedings of the Twentieth Annual Meeting of the Cognitive Science Society*, pages 350–355.

Fischer, Ingrid and Martina Keil. 1996. Parsing decomposable idioms. In *Proceedings of the 16th International Conference on Computational linguistics*, pages 388—393, Copenhagen, Denmark.

Foster, George. 2000. A maximum entropy/minimum divergence translation model. In K. Vijay-Shanker and Chang-Ning Huang, editors, *Proceedings of the 38th Meeting of the Association for Computational Linguistics*, pages 37–44, Hong Kong, October.

Frank, Robert and Giorgio Satta. 1998. Optimality theory and the computational complexity of constraint violability. *Computational Linguistics*, 24:307–315.

Gale, Bill, Kenneth Church, and David Yarowsky. 1992a. One sense per discourse. In *Proceedings of the ARPA Workshop on Speech and Natural Language Processing*, pages 233–237.

Gale, Bill, Kenneth Church, and David Yarowsky. 1992b. Work on statistical methods for word sense disambiguation. In *AAAI Fall Symposium on Probabilistic Approaches to Natural Language*, pages 54–60, Cambridge, MA.

Gaustad, Tanja. 2001. Statistical corpus-based word sense disambiguation: Pseudowords vs. real ambiguous words. In *Companion Volume to the Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics (ACL/EACL 2001) – Proceedings of the Student Research Workshop*, Toulouse.

Geerts, G. and H. Heestermans, editors. 1992. *Van Dale Groot woordenboek der Nederlandse Taal*. Van Dale Lexicografie, Utrecht-Antwerpen.

Gerdemann, Dale and Gertjan van Noord. 1999. Transducers from rewrite rules with backreferences. In *Ninth Conference of the European Chapter of the Association for Computational Linguistics*, Bergen Norway.

Gerdemann, Dale and Gertjan van Noord. 2000. Approximation and exactness in finite state optimality theory. In Jason Eisner, Lauri Karttunen, and Alain Thériault, editors, *Finite-State Phonology. Proceedings of the Fifth Workshop of the ACL SPecial Interest Group in Computational Phonology*, pages 34–45, Luxembourg.

Gildea, Daniel and Daniel Jurafsky. 1996. Learning bias and phonological-rule induction. *Computational Linguistics*, 22(4):497–530.

Gold, E. M. 1996. Language identification in the limit. *Information and Control*, 10:447–474.

Good, I.J. 1963. Maximum entropy for hypothesis formulation, especially for multidimensional contingency tables. *Annals of Mathematical Statistics*, 34:911–934.

Groot, Mila. 2000. Lexiconopbouw: microstructuur. Internal report of the project Corpus Gesproken Nederlands.

Haeseryn, W. and K. Romijn et al., editors. 1997. *Algemene Nederlandse Spraakkunst*. Nijhoff, Groningen.

Hawkins, Paul Martin. 1999. *DURHAM: A Word Sense Disambiguation System*. Ph.D. thesis, Laboratory for Natural Language Engineering, Department of Computer Science, University of Durham.

Hendrickx, Iris and Antal van den Bosch. 2001. Dutch word sense disambiguation: Data and preliminary results. In *Proceedings of Senseval-2, Second International Workshop on Evaluating Word Sense Disambiguation Systems*, pages 13–16, Toulouse.

Hermans, Ben and Marc van Oostendorp, editors. 1999. *The Derivational Residue in Phonological Optimality Theory*, volume 28 of *Linguistik Aktuell/Linguistics Today*. John Benjamins, Amsterdam/Philadelphia.

Hinrichs, Erhard and Tsuneko Nakazawa. 1994. Linearizing AUXs in German verbal complexes. In John Nerbonne, Klaus Netter, and Carl Pollard, editors, *German in Head-driven Phrase Structure Grammar*, Lecture Note Series. CSLI, Stanford, pages 11–38.

Hopcroft, John E. 1971. An $n \log n$ algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*. Academic Press, pages 189–196.

Hopcroft, John E. and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley.

Ide, Nancy and Jean Véronis. 1998. Introduction to the special issue on word sense disambiguation: The state of the art. *Computational Linguistics*, 24(1):1–40.

Jaynes, E.T. 1957. Information theory and statistical mechanics. *Physical Review*, 106,108:620–630.

Jelinek, Frederick. 1998. *Statistical Methods for Speech Recognition*. MIT Press.

Johnson, C. Douglas. 1972. *Formal Aspects of Phonological Descriptions*. Mouton, The Hague.

Johnson, J. Howard and Derick Wood. 1997. Instruction computation in subset construction. In Darrell Raymond, Derick Wood, and Sheng Yu, editors, *Automata Implementation*. Springer Verlag, pages 64–71. Lecture Notes in Computer Science 1260.

Johnson, Mark. 1998. Finite-state approximation of constraint-based grammars using left-corner grammar transforms. In *COLING-ACL '98. 36th Annual Meeting of the Association for Computational Linguistics and the 17th International Conference on Computational Linguistics. Proceedings of the Conference*, pages 619–623.

Johnson, Mark, Stuart Geman, Stephen Canon, Zhiyi Chi, and Stefan Riezler. 1999. Estimators for stochastic "unification-based" grammars. In *Proceedings of the 37th Annual Meeting of the ACL*, pages 535–541, College Park, Maryland.

Kager, René. 1999. *Optimality Theory*. Cambridge UP, Cambridge, UK.

Kaplan, Ronald and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–379.

Kaplan, Ronald M. and Lauri J. Karttunen. 1998. Finite-state encoding system for hyphenation rules. United States Patent 5,737,621.

Karlsson, Fred, Atro Voutilainen, Juha Heikkilä, and Arto Anttila. 1995. *Constraint Grammar: A Language-Independent Framework for Parsing Unrestricted Text*. Mouton de Gruyter, Berlin/New York.

Karov, Yael and Shimon Edelman. 1996. Learning similarity-based word sense disambiguation from sparse data. In *Proceedings of the 4th Workshop on very large corpora*, Copenhagen.

Karov, Yael and Shimon Edelman. 1998. Similarity-based word sense disambiguation. *Computational Linguistics*, 24(1):41–59.

Karttunen, Lauri. 1991. Finite-state constraints. In *Proceedings International Conference on Current Issues in Computational Linguistics*, pages 23–40, Universiti Sains Malaysia, Penang.

Karttunen, Lauri. 1995. The replace operator. In *33th Annual Meeting of the Association for Computational Linguistics*, pages 16–23, M.I.T. Cambridge Mass.

Karttunen, Lauri. 1996. Directed replacement. In *34th Annual Meeting of the Association for Computational Linguistics*, pages 108–115, Santa Cruz.

Karttunen, Lauri. 1998. The proper treatment of optimality theory in computational phonology. In *Finite-state Methods in Natural Language Processing*, pages 1–12, Ankara.

Karttunen, Lauri, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. 1996. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–338. http://www.rxrc.xerox.com/research/mltt/fst/articles/jnle-97/rele.html.

Katz, Jerrold J. 1973. Compositionality, idiomaticity & lexical substitution. In Stephen R. Anderson and Paul Kiparsky, editors, *A Festschrift for Morris Halle*. Holt, Rinehart and Winston, Inc., pages 357—376.

Kehler, Andrew. 1994. Common topics and coherent situations: interpreting ellipsis in the context of discourse coherence. In *Proceedings of the 32th Annual Meeting of the Association for Computational Linguistics (ACL-94)*.

Kempe, André. 2000. Factorization of ambiguous finite-state transducers. In *CIAA 2000. Fifth International Conference on Implementation and Application of Automata. Preproceedings*, pages 157–164, London, Ontario, Canada.

Kempe, André and Lauri Karttunen. 1996. Parallel replacement in the finite-state calculus. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING)*, pages 622–627, Copenhagen, Denmark.

Kiefer, B., H.-U. Krieger, J. Carroll, and R. Malouf. 1999. A bag of useful techniques for efficient and robust parsing. In *Proceedings of the 37th Meeting of the Association for Computational Linguistics*, pages 473–480, College Park, MD.

Kilgarriff, Adam. 1998a. Gold standard datasets for evaluating word sense disambiguation programs. *Computer Speech and Language, Special Issue on Evaluation*, 12(4):453–472.

Kilgarriff, Adam. 1998b. Senseval: An exercise in evaluating word sense disambiguation programs. In *Proceedings of the First International Conference on Language Resources and Evaluation (LREC 1998)*, pages 581–588, Granada.

Kilgarriff, Adam. 2001. English lexical sample task description. In *Proceedings of Senseval-2, Second International Workshop on Evaluating Word Sense Disambiguation Systems*, pages 17–20, Toulouse.

Kilgarriff, Adam and Joseph Rosenzweig. 2000. Framework and results for english Senseval. *Computers and the humanities*, 34(1-2):15–48.

Kiraz, George Anton. 1999. Compressed storage of sparse finite-state transducers. In O. Boldt, H. Jürgensen, and L. Robbins, editors, *Workshop on Implementing Automata WIA99 - Pre-Proceedings*, Potsdam, July.

Kisseberth, Charles. 1970. On the functional unity of phonological rules. *Linguistic Inquiry*, 1:291–306.

Klarlund, Nils. 1998. Mona & fido: The logic-automaton connection in practice. In *Computer Science Logic, CSL '97*, LNCS. LNCS 1414.

Knuth, Donald E. 1998. *The Art of Computer Programming, Volume 3, Sorting and Searching.* Addison Wesley, second edition edition.

Koster, Jan. 1975. Dutch as an SOV language. *Linguistic Analysis*, 1.

Kowaltowski, Tomasz, Cláudio L. Lucchesi, and Jorge Stolfi. 1993. Minimization of binary automata. In *First South American String Processing Workshop*, Belo Horizonte, Brasil.

Kowaltowski, Tomasz, Cláudio L. Lucchesi, and Jorge Stolfi. 1998. Finite automata and efficient lexicon implementation. Technical Report IC-98-02, icunicamp, January.

Krenn, Brigitte. 2000. *The Usual Suspects: Data Oriented Models for the Identification and Representation of Lexical Collocations.* Ph.D. thesis, DFKI & Universitat des Saarlandes.

Krenn, Brigitte and Gregor Erbach. 1994. Idioms and support verb constructions. In John Nerbonne, Klaus Netter, and Carl Pollard, editors, *German in Head-Driven Phrase Structure Grammar.* CSLI, pages 365—395.

Krovetz, Robert. 1998. More than one sense per discourse. In *Proceedings of the ACL SIGLEX Workshop.*

Lafferty, John, Fernando Pereira, and Andrew McCallum. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *International Conference on Machine Learning (ICML).*

Lafferty, John and Bernhard Suhm. 1996. Cluster expansions and iterative scaling for maximum entropy language models. In K. Hanson and R. Silver, editors, *Maximum Entropy and Bayesian Methods.* Kluwer.

Langley, Pat, Wayne Iba, and Kevin Thompson. 1992. An analysis of bayesian classifiers. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92)*, San Jose.

Lapata, Maria. 1999. Acquiring lexical generalizations from corpora: a case study for diathesis alternations. In *Proceedings of 37th ACL*, pages 397—404.

Leacock, Claudia, Martin Chodorow, and George A. Miller. 1998. Using corpus statistics and WordNet relations for sense identification. *Computational Linguistics*, 24(1):147–165.

Liang, Franklin Mark. 1983. *Word Hy-phen-a-tion by Com-put-er.* Ph.D. thesis, Stanford University.

Lin, Dekang. 1998. Extracting collocations from text corpora. In *First workshop on computational terminology*, Montreal, Canada.

Lin, Dekang. 1999. Automatic identification of non-compositional phrases. In *Proceedings of ACL-99*, pages 317–324. University of Maryland.

Lucchiesi, Claudio and Tomasz Kowaltowski. 1993. Applications of finite automata representing large vocabularies. *Software Practice and Experience*, 23(1):15–30, Jan.

Malouf, Robert. 2000. The order of prenominal adjectives in natural language generation. In K. Vijay-Shanker and Chang-Ning Huang, editors, *Proceedings of the 38th Meeting of the Association for Computational Linguistics*, pages 85–92, Hong Kong, October.

Malouf, Robert. 2002. A comparison of algorithms for maximum entropy parameter estimation. In *Proceedings of the Sixth Conference on Natural Language Learning (CoNLL-2002)*, Taiwan. to appear.

Malouf, Robert and Miles Osborne. 2000. A toolkit for robust and efficient maximum entropy language modeling. Computational Linguistics in the Netherlands (CLIN-2000), November.

Manning, Christopher D. and Hinrich Schütze. 1999. *Foundations of Statistical Natural Language Processing.* MIT Press, Cambridge Mass.

Marcus, G. F. 1993. Negative evidence in language acquisition. *Cognition*, 46:53–85.

McCarthy, John and Alan Prince. 1995. Faithfulness and reduplicative identity. In Jill Beckman, Laura Walsh Dickey, and Suzanne Urbanczyk, editors, *Papers in Optimality Theory*. Graduate Linguistic Student Association, Amherst, Mass, pages 249–384. University of Massachusetts Occasional Papers in Linguistics 18.

Mohri, Mehryar. 1994. Compact representations by finite-state transducers. In *32th Annual Meeting of the Association for Computational Linguistics*, pages 204–209, New Mexico State University.

Mohri, Mehryar. 1996. On some applications of finite-state automata theory to natural language processing. *Natural Language Engineering*, 2:61–80. Originally appeared in 1994 as Technical Report, institut Gaspard Monge, Paris.

Mohri, Mehryar. 2000. Minimization algorithms for sequential transducers. *Theoretical Computer Science*, 234:177–201.

Mohri, Mehryar and Richard Sproat. 1996. An efficient compiler for weighted rewrite rules. In *34th Annual Meeting of the Association for Computational Linguistics*, Santa Cruz.

Moon, Rosamund. 1998. *Fixed expressions and Idioms in English. A corpus-based approach.* Clarendom Press, Oxford.

Mooney, Raymond. 1996. Comparative experiments on disambiguating word senses: An illustration of the role of bias in machine learning. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP-96)*, pages 82–91, University of Pennsylvania.

Moortgat, Michael, Ineke Schuurman, and Ton van der Wouden. 2002. CGN syntactische annotatie. internal report corpus Gesproken Nederlands.

Mullen, Tony. 2002. *An Investigation into Compositional Features and Feature Merging for Maximum Entropy-Based Parse Selection.* Ph.D. thesis, University of Groningen.

Mullen, Tony, Robert Malouf, and Gertjan van Noord. 2001. Statistical parsing of dutch using maximum entropy models with feature merging. In J. Tsujii, editor, *NLPRS2001, Proceedings of the Sixth Natural Language Processing Pacific Rim Symposium*, pages 481–486, Tokyo. University of Tokyo Press.

Mullen, Tony and Miles Osborne. 2000. Overfitting avoidance for stochastic modeling of attribute-value grammars. In *Proceedings of the Fourth Conference on Computational Language Learning (CoNLL-2000)*, Lisbon.

Nederhof, M.-J. 2000. Practical experiments with regular approximation of context-free languages. *Computational Linguistics*, 26(1).

Ng, Hwee Tou and Hian Beng Lee. 1996. Integrating multiple knowledge sources to disambiguate word sense: An exemplar-based approach. In *34th Annual Meeting of the Association for Computational Linguistics*, pages 40–47, Santa Cruz, CA.

Ngai, Grace and Radu Florian. 2001. Transformation-based learning in the fast lane. In *Proceedings of the second conference of the North American chapter of the ACL*, pages 40–47, Pittsburgh.

Nicolas, Tim. 1995. Semantics of idiom modification. In Martin Everaert, Erik-Jan van der Linden, Andre Schenk, and Rob Schreuder, editors, *Idioms: Structural & Psychological Perspectives*. Lawrence Erlbaum Associates, New Jersey.

Nocedal, Jorge. 1997. Large scale unconstrained optimization. In A. Watson and I. Duff, editors, *The State of the Art in Numerical Analysis*. Oxford University Press, pages 311–338.

Nocedal, Jorge and Stephen J. Wright. 1999. *Numerical Optimization*. Springer, New York.

van Noord, Gertjan. 1997a. An efficient implementation of the head corner parser. *Computational Linguistics*, 23(3):425–456. cmp-lg/9701004.

van Noord, Gertjan. 1997b. FSA Utilities: A toolbox to manipulate finite-state automata. In Darrell Raymond, Derick Wood, and Sheng Yu, editors, *Automata Implementation*. Springer Verlag, pages 87–108. Lecture Notes in Computer Science 1260.

van Noord, Gertjan. 1999. FSA6 reference manual. The *FSA Utilities* toolbox is available free of charge under Gnu General Public License at http://www.let.rug.nl/˜vannoord/Fsa/.

van Noord, Gertjan. 2000a. Empirical aspects of finite state language processing; bijlage 1: Onderzoeksplan. Appendix to research proposal.

van Noord, Gertjan. 2000b. The treatment of epsilon moves in subset construction. *Computational Linguistics*, 26(1):61–76.

van Noord, Gertjan. 2001. Robust parsing of word graphs. In Jean-Claude Junqua and Gertjan van Noord, editors, *Robustness in Language and Speech Technology*. Kluwer Academic Publishers, Dordrecht.

van Noord, Gertjan and Gosse Bouma. 1994. Adjuncts and the processing of lexical rules. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING)*, pages 250–256, Kyoto. cmp-lg/9404011.

van Noord, Gertjan and Gosse Bouma. 1997a. Dutch verb clustering without verb clusters. In Patrick Blackburn and Maarten de Rijke, editors, *Specifying Syntactic Structures*. CSLI Publications / Folli, Stanford, pages 213–243.

van Noord, Gertjan and Gosse Bouma. 1997b. Hdrug, A flexible and extendible development environment for natural language processing. In *Proceedings of the EACL/ACL workshop on Environments for Grammar Development, Madrid*.

van Noord, Gertjan, Gosse Bouma, Rob Koeling, and Mark-Jan Nederhof. 1999. Robust grammatical analysis for spoken dialogue systems. *Journal of Natural Language Engineering*, 5(1):45–93.

van Noord, Gertjan and Dale Gerdemann. 2000. An extendible regular expression compiler for finite-state approaches in natural language processing. In O. Boldt and H. Juergensen, editors, *Automata Implementation. 4th International Workshop on Implementing Automata, WIA '99*. Springer. Springer Lecture Notes in Computer Science 2214.

van Noord, Gertjan and Dale Gerdemann. 2001. Finite state transducers with predicates and identities. *Grammars*, 4:263–286.

Nunberg, Geoffrey, Ivan A. Sag, and Thomas Wasow. 1994. Idioms. *Language*, 70(3):491–539.

Oflazer, Kemal. 1999. Dependency parsing with an extended finite state approach. In *37th Annual Meeting of the Association for Computational Linguistics*, pages 254–260.

Oncina, J., P. García, and E. Vidal. 1993. Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15:448–458.

Oostdijk, Nelleke. 2000. The Spoken Dutch Corpus: Overview and first evaluation. In *Proceedings of Second International Conference on Language Resources and Evaluation (LREC)*, pages 887–894.

Osborne, Miles. 2000. Estimation of stochastic attribute-value grammars using an informative sample. In *Proceedings of the Eighteenth International Conference on Computational Linguistics (COLING 2000)*.

Osborne, Miles. 2002. Shallow parsing using noisy and non-stationary training material. *Journal of Machine Learning Research*, 2:695–719.

Osborne, Miles. to appear. Using maximum entropy for sentence extraction. In *Proceedings of the ACL 2002 Workshop on Automatic Summarization*, Philadelphia.

Paardekooper, P.C. 1962. Voorzetsel-uitdrukkingen. *Nieuwe Taalgids*, 55:3–9.

Paardekooper, P.C. 1973. Grensproblemen bij v-z-uitdrukkingen. *Nieuwe Taalgids*, 66:137–145.

Pantel, Patrick and Dekang Lin. 2000. An unsupervised approach to prepositional phrase attachment using contextually similar words. In K. Vijay-Shanker and Chang-Ning Huang, editors, *Proceedings of the 38th Meeting of the Association for Computational Linguistics*, pages 101–108, Hong Kong, October.

Pearce, Darren. 2001. Synonymy in collocation extraction. In *WordNet and Other lexical resources: applications, extensions & customizations (NAACL 2001)*, pages 41–46, Pittsburgh. Carnegie Mellon University.

Pearce, Darren. 2002. A comparative evaluation of collocation extraction techniques. To appear in 'Third International Conference on Language Resources and Evaluation', May.

Pedersen, Ted. 1996. Fishing for exactness. In *Proceedings of the South Central SAS User's Group (SCSUG-96) Conference*, Austin, TX.

Pedersen, Ted. 2001. A decision tree of bigrams is an accurate predictor of word sense. In *Proceedings of the Second Conference of the North American Chapter of the Association for Computational Linguistics*, Pittsburgh, PA.

Perlmutter, David M. and Scott Soames. 1979. *Syntactic Argumentation & the structure of English*. University of California Press, California.

Perrin, Dominique. 1990. Finite automata. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science. Volume B: Formal Models and Semantics*. Elsevier and the MIT Press, pages 1–57.

Pollard, Carl and Ivan Sag. 1994. *Head-driven Phrase Structure Grammar.* University of Chicago / CSLI.

Prince, A. and P. Smolensky. 1995. *Optimality Theory: Constraint Interaction in Generative Grammar.* MIT Press, Cambridge, MA.

Prince, Alan and Paul Smolensky. 1993. Optimality theory: Constraint interaction in generative grammar. Technical Report TR-2, Rutgers University Cognitive Science Center, New Brunswick, NJ. MIT Press, To Appear.

Prins, Robbert and Gertjan van Noord. 2001. Unsupervised pos-tagging improves parsing accuracy and parsing efficiency. In *Proceedings of the Seventh International Workshop on Parsing Technologies (IWPT)*, pages 154–165, Beijing, China.

Ratnaparkhi, Adwait. 1998. *Maximum entropy models for natural language ambiguity resolution.* Ph.D. thesis, University of Pennsylvania.

Resnik, Philip. 1993. *Selection and Information: A Class-based Approach to Lexical Relationships*. Ph.D. thesis, Institute for Research in Cognitive Science IRCS, University of Pennsylvania.

Resnik, Philip. 1997. Selectional preferences and sense disambiguation. In *ACL SIGLEX Workshop on Tagging Text with Lexical Semantics: Why, What, and How?*, pages 52–57, Washington, D.C.

Reutenauer, C. 1993. Subsequential functions: Characterizations, minimization, examples. In *Proceedings of the International Meeting of Young Computer Scientists*, Berlin. Springer. Lecture Notes in Computer Science.

Revuz, Dominique. 1991. *Dictionnaires et lexiques: méthodes et algorithmes.* Ph.D. thesis, Institut Blaise Pascal, Paris, France. LITP 91.44.

Riehemann, Susanne. 1997. Idiomatic constructions in HPSG. Presented at the Fourth International Conference on Head-Driven Phrase Structure Grammar.

Riehemann, Susanne. 2001. *A constructional approach to idioms and word formation.* Ph.D. thesis, Stanford University.

Riehemann, Susanne Z. and Emily Bender. 1999. Absolute constructions: on the distribution of predicative idioms. In S.Bird, A. Carnie, J. Haugen, and P. Norquest, editors, *WCCFL 18 Proceedings*, pages 476—489, Somerville. Cascadilla Press.

Riezler, Stefan, Detlef Prescher, Jonas Kuhn, and Mark Johnson. 2000. Lexicalized stochastic modeling of constraint-based grammars using log-linear measures and em. In *Proceedings of the 38th Annual Meeting of the ACL*, pages 480–487, Hong Kong.

Roche, Emmanuel. 1995. Finite-state tools for language processing. In *ACL'95*. Association for Computational Linguistics. Tutorial.

Roche, Emmanuel and Yves Schabes. 1995. Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics*, 21(2):227–253.

Roche, Emmanuel and Yves Schabes. 1997a. Deterministic part-of-speech tagging with finite-state transducers. In Emmanuel Roche and Yves Schabes, editors, *Finite state language processing*. MIT Press, Cambridge, Mass., pages 205–239.

Roche, Emmanuel and Yves Schabes. 1997b. Introduction. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*. MIT Press, Cambridge, Mass.

Sag, Ivan. 1997. English relative clause constructions. *Journal of Linguistics*, 33(2):431–484.

Sag, Ivan, T. Baldwin, F. Bond, A. Copestake, and D. Flickinger. 2001. Multiword expressions: a pain in the neck for nlp. LinGO Working Paper No. 2001-03.

Sag, Ivan A. and Thomas Wasow. 1999. *Syntactic Theory: A Formal Introduction*. CSLI Publications, Stanford CA.

Sailer, Manfred. 2000. *Combinatorial Semantics & Idiomatic Expressions in Head-Driven Phrase Structure Grammar*. Ph.D. thesis, University of Tuebingen.

Sarkar, Anoop and Daniel Zeman. 2000. Automatic extraction of subcategorization frames for Czech. In *Proceedings of the 18th International Conference on Computational Linguistics (COLING 2000)*, Saarbrücken.

Schenk, Andre. 1994. *Idioms & collocations in compositional grammars*. Ph.D. thesis, Universiteit Utrecht.

Schulte im Walde, Sabine, Helmut Schmid, Mats Rooth, Stefan Riezler, and Detlef Prescher. 2001. Statistical Grammar Models and Lexicon Acquisition. In Christian Rohrer, Antje Rossdeutscher, and Hans Kamp, editors, *Linguistic Form and its Computation*. CSLI Publications, Stanford, CA.

Schulze, Bruno Maximilian, Ulrich Heid, Helmut Schmid, Anne Schiller, Mats Rooth, Gregory Grefenstette, Jean Gaschler, Annie Zaenen, and Simone Teufel. 1994. Decide. MLAP-Project 93-19 D-1b I, STR and RXRC, November.

Schütze, Hinrich. 1992. Context space. In *AAAI Fall Symposium on Probabilistic Approaches to Natural Language*, pages 113–120, Cambridge, MA.

Schütze, Hinrich. 1998. Automatic word sense disambiguation. *Computational Linguistics*, 24(1):97–123.

Shewchuk, Jonathan Richard. 1994. An introduction to the conjugate gradient method without the agonizing pain. http://www.cs.cmu.edu/ quake-papers/painless-conjugate-gradient.ps.

Silberztein, Max. 1997. The lexical analysis of natural languages. In Emmanuel Roche and Ives Schabes, editors, *Finite-State Language Processing*. MIT Press, pages 175–203.

Silberztein, Max. 1999. INTEX tutorial notes. In *Workshop on Implementing Automata WIA99 – Pre-Proceedings*, pages XIX–1 – XIX–31.

Skut, W., B. Krenn, and H. Uszkoreit. 1997. An annotation scheme for free word order languages. In *Proceedings of the Fifth Conference on Applied Natural Language Processing*, Washington,DC.

Sojka, Petr. 1995. Notes on compound word hyphenation in TEX. In *TUGboat 1995*. Proceedings of the 16th annual meeting of the Tex users group.

Sterling, Leon and Ehud Shapiro. 1994. *The Art of Prolog*. MIT Press, Cambridge Mass. Second Edition.

Stolcke, A. and S. Omohundro. 1993. Hidden Markov Model induction by Bayesian model merging. In *Advances in Neural Information Processing Systems*, volume 5. Morgan Kaufman.

Tarjan, Robert Endre and Andrew Chi-Chih Yao. 1979. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, November.

Tesar, Bruce. 1995. *Computational Optimality Theory*. Ph.D. thesis, University of Colorado, Boulder.

Thollard, Franck, Pierre Dupont, and Colin de la Higuera. 2000. Probabilistic DFA inference using Kullback-Leibler divergence and minimality. In *Proc. 17th International Conf. on Machine Learning*, pages 975–982. Morgan Kaufmann, San Francisco, CA.

Tutelaers, P. T. H. 1999. Afbreken in TEX, hoe werkt dat nou? available at ftp://ftp.tue.nl/pub/tex/afbreken.

Uit den Boogaart, P. C. 1975. *Woordfrequenties in geschreven en gesproken Nederlands*. Oosthoek, Scheltema & Holkema, Utrecht. Werkgroep Frequentie-onderzoek van het Nederlands.

Van Eynde, F. 1996. Het nederlands en de nieuwe taaltechnologie. vijf voor twaalf? In *Over de toekomst van het Nederlands*. Davidsfonds/Clauwaert, Leuven, pages 25–40.

Vosse, Theo. 1994. *The Word Connection*. Ph.D. thesis, Rijksuniversiteit Leiden.

Voutilainen, Atro. 1998. Does tagging help parsing? a case study on finite state parsing. In *Finite-state Methods in Natural Language Processing*, Ankara.

Walther, Markus. 1999. One-level prosodic morphology. Technical Report 1, Institüt für Germanistische Sprachwissenschaft, Philipps-Universität Marbug. cs.CL/9911011.

Walther, Markus. 2000. Finite-state reduplication in one-level prosodic morphology. In *First Conference of the North American Chapter of the Association for Computational Linguistics*, pages 296–302, Seattle.

Watson, Bruce. 1998. A fast new (semi-incremental) algorithm for the construction of minimal acyclic DFAs. In *Third Workshop on Implementing Automata*, pages 91–98, Rouen, France, September. Lecture Notes in Computer Science.

Watson, Bruce. 2001. An incremental DFA minimization algorithm. In Lauri Karttunen, Kimmo Koskenniemi, and Gertjan van Noord, editors, *proceedings of FSMNLP 2001, ESSLLI workshop*, Helsinki, August. Available at `http://www.let.rug.nl/~vannoord/alp/esslli_fsmnlp/watson.pdf`.

Watson, Bruce W. 1999a. Implementing and using finite automata toolkits. In A. Kornai, editor, *Extended Finite State Models of Language*. Cambridge University Press, pages 19–36.

Watson, Bruce W. 1999b. The OpenFIRE initiative. In Junichi Aoe, editor, *Proceedings of the International Conference on Computer Processing of Oriental Languages*, pages 421–424, Tokushima, Japan.

Wauschkuhn, Oliver. 1995. The influence of tagging on the results of partial parsing in German corpora. In *Fourth International Workshop on Parsing Technologies*, pages 260–270, Prague/Karlovy Vary, Czech Republic.

Weeber, M., R. Vos, and Harald Baayen. 2000. Extracting the lowest-frequency words: Pitfalls and possibilities. *Computational Linguistics*, 26(3):301–317.

Wilks, Yorick and Mark Stevenson. 1997. Combining independent knowledge sources for word sense disambiguation. In *Proceedings of the Conference Recent Advances in Natural Language Processing*, pages 1–7, Tzigov Chark.

Wilks, Yorick and Mark Stevenson. 1998. Word sense disambiguation using optimised combinations of knowledge sources. In *COLING-ACL '98. 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics. Proceedings of the Conference*, pages 1398–1402, Montreal.

Wu, Jun and Sanjeev Khudanpur. 2000. Efficient training methods for maximum entropy language modelling. In *Proceedings of ICSLP2000*, volume 3, pages 114–117, Beijing.

Yarowsky, David. 1992. Word-sense disambiguation using statistical models of Roget's categories trained on large corpora. In *Proceedings of the 15th [sic] International Conference on Computational Linguistics (COLING)*, pages 454–460, Nantes.

Yarowsky, David. 1993. One sense per collocation. In *Proceedings ARPA Human Language Technology Workshop*, Princeton.

Yarowsky, David. 1995. Unsupervised word sense disambiguation rivaling supervised methods. In *33th Annual Meeting of the Association for Computational Linguistics*, pages 189–196, Cambridge, MA.

van Zanten, Gert Veldhuijzen, Gosse Bouma, Khalil Sima'an, Gertjan van Noord, and Remko Bonnema. 1999. Evaluation of the NLP components of the OVIS2 spoken dialogue system. In Frank van Eynde, Ineke Schuurman, and Ness Schelkens, editors, *Computational Linguistics in the Netherlands 1998*, pages 213–229. Rodopi Amsterdam.

Zhu, Song Chun, Ying Nian We, and David Mumford. 1997. Minimax entropy principle and its application to texture modeling. *Neural Computation*, 9:1627–1660.

Appendix A

Publications

A.1 Journal Articles

Leonoor van der Beek, Gosse Bouma, Gertjan van Noord. Een brede computationele grammatica voor het Nederlands. *Nederlandse Taalkunde.* To appear.

Gosse Bouma and Ineke Schuurman. Naar een digitale bibliotheek voor de taalkunde. *Nederlandse Taalkunde* 5 (2). 2000. pp. 177–180.

Gosse Bouma and Ineke Schuurman. De digitale infrastructuur van het Nederlands. *Nederlandse Taalkunde* 5 (2). 2000. pp. 90–94.

Gosse Bouma, Robert Malouf and Ivan Sag, Satisfying Constraints on Extraction and Adjunction, In: *Natural Language and Linguistic Theory*, 19, 2001, pp. 1–65.

Jan Daciuk and S. Mihov and B. Watson and R. Watson, Incremental Construction of Minimal Acyclic Finite-state Automata. *Computational Linguistics*, 26 (1). 2000. pp. 3–16.

Gertjan van Noord, Treatment of Epsilon Moves in Subset Construction. *Computational Linguistics* 26 (1). 2000. pp. 61–76.

Gertjan van Noord and Dale Gerdemann. Finite State Transducers with Predicates and Identity. *Grammars* 4 (3). 2001. pp. 263–286.

A.2 Ph.D. Thesis

Tony Mullen, An Investigation into Compositional Features and Feature Merging for Maximum Entropy-Based Parse Selection. Ph.D. Thesis. University of Groningen 2002.

A.3 Edited Volumes

Jean-Claude Junqua and Gertjan van Noord (editors), *Robustness in Language and Speech Technology.* Kluwer. 2001. ISBN 0-7923-6790-1

Lauri Karttunen, Kimmo Koskenniemi, Gertjan van Noord (editors), *Finite State Methods in Natural Language Processing. FSMNLP 2001.* Extended Abstracts. ESSLLI Workshop, Helsinki. 2001.

A.4 Reviewed Book Chapters

Leonoor van der Beek, Gosse Bouma, Robert Malouf, Gertjan van Noord. The Alpino Dependency Treebank. *Computational Linguistics in the Netherlands CLIN 2001.* To appear.

Gosse Bouma and Frank van Eynde and Dan Flickinger, Constraint-based Lexica. In F. van Eynde and D. Gibbon (eds), *Lexicon Development for Speech and Language Processing.* Kluwer 2000. pp. 43–76.

Gosse Bouma, Gertjan van Noord, Robert Malouf. Alpino: Wide Coverage Computational Analysis of Dutch. In: W. Daelemans, K. Sima'an, J. Veenstra, J. Zavrel (eds), *Computational Linguistics in the Netherlands CLIN 2000.* Rodopi, Amsterdam, 2001. pp. 45–59.

Gosse Bouma and Begoña Villada Moirón. Corpus-based acquisition of collocational prepositional phrases. *Computational Linguistics in the Netherlands CLIN 2001.* To appear.

Gosse Bouma. Argument realization and Dutch R-Pronouns: Solving Bech's problem without movement or deletion. In Ronnie Cann, Claire Grover, and Philip Miller (eds), *Grammatical Interfaces in Head-driven Phrase Structure Grammar.* CSLI Publications, 2000.

Jan Daciuk, Treatment of Unknown Words, In: O. Boldt, and H. Jörgensen, *Automata Implementation,* 4th International Workshop on Implementing Automata, WIA '99 Potsdam, Germany, July 1999, Revised Papers, Springer Verlag, Berlin-New York, etc. 2001, pp. 71–80.

Tanja Gaustad and Gosse Bouma. Accurate Stemming of Dutch for Text Classification. *Computational Linguistics in the Netherlands CLIN 2001.* To appear.

Robert Malouf, Cooperating Constructions. In: E. Francis and L. Michaelis (eds.), *Linguistic Mismatch: Scope and Theory.* CSLI Publications. To appear.

Gertjan van Noord, Dale Gerdemann, An Extendible Regular Expression Compiler for Finite-state Approaches in Natural Language Processing. In: O.Boldt, H.Juergensen (eds), *Automata Implementation. 4th International Workshop on Implementing Automata, WIA '99, Potsdam Germany, July 1999, Revised Papers.* Springer Lecture Notes in Computer Science 2214. 2000.

Gertjan van Noord, Robust Parsing of Word Graphs. In: Jean-Claude Junqua and Gertjan van Noord (editors), *Robustness in Language and Speech Technology.* Kluwer. 2001. ISBN 0-7923-6790-1

A.5 Reviewed Conference Proceedings

Gosse Bouma, A finite state and data-oriented method for grapheme to phoneme conversion. In: *NAACL 2000*, pp 303–310. 2000.

Gosse Bouma, Extracting Dependency Frames from Existing Lexical Resources, In: Moldovan, D , e.a., (red.), *WordNet and Other Lexical Resources: Applications, Extensions and Customizations*, Pittsburgh 3-4 June, 2001, NAACL 2001 Workshop, Association for Computational Linguistics, 2001, pp. 65-70

Gosse Bouma, Finite State Methods for Hyphenation, In: Lauri Karttunen, Kimmo Koskenniemi, Gertjan van Noord (eds), *Finite State Methods in Natural Language Processing*, Extended Abstracts, ESSLLI Workshop, August 20-24 2001, Helsinki, University of Helsinki, Helsinki, 2001, pp. 29-33.

Gosse Bouma and Geert Kloosterman. Querying dependency treebanks in XML. In *Proceedings of the Third international conference on Language Resources and Evaluation* (LREC). Gran Canaria. 2002.

Jan Daciuk and Gertjan van Noord, Finite Automata for Compact Representation of Language Models in NLP, In: Bruce Watson and Derick Wood (eds.), *Proceedings of the 6th Conference on Implementations and Applications on Automata*, 23–25 July 2001, Pretoria, Republic of South Africa, University of Pretoria, Dept. of Computer Science, Pretoria, 2001, pp. 45–55.

Jan Daciuk, Computer-Assisted Enlargement of Morphological Dictionaries, In: Lauri Karttunen, Kimmo Koskenniemi and Gertjan van Noord (eds.), *Finite State Methods in Natural Language Processing,* Extended Abstracts, ESSLLI Workshop, August 20-24 2001, Helsinki, University of Helsinki, Helsinki, 2001, pp. 23-27

Jan Daciuk, Experiments with Automata Compression. In: M. Daley, M. Eramian and S. Yu (eds), *CIAA*, University of Western Ontario, London, Canada. 2000. pp. 113–119.

Jan Daciuk. Finite State Tools. In: R. Zajac (ed), COLING Workshop *Using Toolsets and Architectures to build NLP Systems.* 2000. pp. 34–37.

Jan Daciuk, Comparison of Construction Algorithms for Minimal, Acyclic, Deterministic, Finite-State Automata from Sets of Strings, *Seventh International Conference on Implementation and Application of Automata* CIAA '2002, Tours, France, 2002.

Tanja Gaustad, Statistical Corpus-Based Word Sense Disambiguation: Pseudowords vs. Real Ambiguous Words, In: Miltsakaki, E , e.a., (eds.), *39th Annual Meeting aand 10th Conference of the European Chapter, Companion Volume to the Proceedings of the Conference: Proceedings of the Student Research Workshop and Tutorial Abstracts*, CNRS , Toulouse, 2001, pp. 61–66.

Tanja Gaustad, Extraktion und Verifikaton von Subkategorisierungsmustern für Französische Verben. In: U. Heid, S. Evert, E. Lehmann, C. Rohrer (eds), *Proceedings of the Ninth Euralex International Congress.* University of Stuttgart. 2000. pp. 611–617.

Dale Gerdemann and Gertjan van Noord. Approximation and Exactness in Finite State Optimality Theory. In: Jason Eisner, Lauri Karttunen, Alain Thriault (editors), *SIGPHON 2000, Finite State Phonology. Proceedings of the Fifth Workshop of the ACL Special Interest Group in Computational Phonology.* August 2000, Luxembourg.

Robert Malouf. Markov models for language-independent named entity recognition. In *Proceedings of the Sixth Conference on Natural Language Learning* (CoNLL-2002). Taiwan. To appear.

Robert Malouf. A comparison of algorithms for maximum entropy parameter estimation. In *Proceedings of the Sixth Conference on Natural Language Learning* (CoNLL-2002). Taiwan. To appear.

Tony Mullen, Robert Malouf, and Gertjan van Noord, Statistical Parsing of Dutch using Maximum Entropy Models with Feature Merging, In: Tsujii, J (eds.), *NLPRS2001, Proceedings of the Sixth Natural Language Processing Pacific Rim Symposium,* November 27–30, 2001, University of Tokyo Press, Tokyo, 2001, pp. 481–486.

Robbert Prins and Gertjan van Noord, Unsupervised pos-tagging improves parsing accuracy and parsing efficiency, In: Bunt, H (eds), *Proceedings of the Seventh International Workshop on Parsing Technologies* - IWPT - 2001, 17–19 October, 2001 Peking University, Beijing, China, Tsinghua University Press, Beijing, 2001, pp. 154–165.

Begoña Villada Moirón and Gosse Bouma. A corpus-based approach to the acquisition of collocational prepositional phrases. In: EURALEX 2002, Copenhagen, August 2002. To appear.

Appendix B

Presentations

Leonoor van der Beek, Gosse Bouma, Robert Malouf and Gertjan van Noord. The Alpino Dependency Treebank. *Empirical methods in the new millennium: Linguistically Interpreted Corpora* (LINC 2001), Leuven, August 29 2001.

Leonoor van der Beek, The Alpino Dependency Treebank; three tools for treebanking, CLIN 2001, Twente University, Enschede, November 30 2001.

Leonoor van der Beek, Cleft Sentences. BCN Poster Day. February 2002, Groningen.

Gosse Bouma, Finite State and Data-Oriented Methods for Grapheme to Phoneme Conversion, NAACL 2000, Seattle. May 2000.

Gosse Bouma, Gertjan van Noord, Alpino: A Wide Coverage Computational Grammar of Dutch. CLIN, Tilburg, November 3 2000.

Gosse Bouma, Alpino, A Wide Coverage Computational Grammar for Dutch, LOT winterschool, University of Amsterdam, January 15 2001 [invited].

Gosse Bouma, Extracting Dependency Frames from Existing Lexical Resources, WordNet and Other Lexical Resources Workshop, Pittsburgh, USA, June 3 2001.

Gosse Bouma, Finite State Methods for Hyphenation, Finite State Methods in Natural Language Processing, Helsinki, Finland, August 20-24 2001.

Gosse Bouma and Geert Kloosterman. Querying dependency treebanks in XML. Poster at the Third international conference on Language Resources and Evaluation (LREC), Gran Canaria, 2002.

Jan Daciuk, Finite Automata for Compact Representation of Language Models in NLP, Six Internations Conference on Implementation and Application of Automata, University of Pretoria, South-Africa, July 23-25 2001.

Jan Daciuk, Computer-assisted Enlargement of Morphological Dictionaries, Finite State Methods in Natural Language Processing, Helsinki, Finland, August 13–24 2001.

Jan Daciuk, Gertjan van Noord A Finite-State Library for NLP, CLIN 2001, University of Twente, Enschede, November 30 2001.

Jan Daciuk. Computer-Aided Enlargment of Morphological Dictionaries. Presented at the Natural Language Processing Seminar, The Linguistic Engineering / Formal

Linguistics Group, Linguistic Engineering Group at the Department of Artificial Intelligence, Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland, June 25th, 2001 [invited].

Jan Daciuk. Incremental Construction of Minimal, Deterministic, Acyclic, Finite-State Automata. Presented at the Seminar für Sprachwissenschaft, Tübingen University, Germany, May 24th, 2000 [invited].

Jan Daciuk. Construction of Guessing Automata for Morphological Analysis and Morphological Descriptions. Presented at the Seminar für Sprachwissenschaft, Tübingen University, Germany, May 22nd, 2000 [invited].

Tanja Gaustad, The Best of two Worlds: Word Sense Disambiguation Using Statistics and Linguistics. BCN Ph.D. Retreat, Doorwerth, April 2002.

Tanja Gaustad, Word Sense Disambiguation as Classification Problem. Potchefstroomse Universiteit vir Christelike Hoër Onderwys, Potchefstroom, South Africa. March 2002.

Tanja Gaustad, Gosse Bouma, Accurate Stemming and Email Classification. CLIN 2001, Enschede, November 30 2001.

Tanja Gaustad, Statistical Corpus-Based Word Sense Disambiguation: Pseudowords vs. Real Ambiguous Words, ACL 2001, Toulouse, France, July 9–11 2001.

Tanja Gaustad, Extraction and Verification of Subcategorization Patterns for French Verbs. EURALEX 2000, Stuttgart, August 2000

Tanja Gaustad, Word Sense Disambiguation Using a Naive Bayes Classification Algorithm and Pseudowords. BCN Poster Day, January 2001, Groningen.

Robert Malouf, Tony Mullen, Gertjan van Noord, Probabilistic parsing with the Alpino grammar, CLIN 2001, Enschede, November 30 2001.

Robert Malouf and Miles Osborne. A toolkit for robust and efficient maximum entropy language modeling. CLIN 2000, Tilburg, November 2000.

Robert Malouf, Stochastic Head-Driven Phrase Structure Grammar. Department of computational linguistics, University of Saarbrücken, May 2002. [invited]

Robert Malouf. Mixed categories in constraint-based grammars. Emanuel Vasiliu Lectures in Formal Grammars, University of Bucharest, April 2002. [invited]

Robert Malouf, Stochastic Head-Driven Phrase Structure Grammar. Deparment of Linguistics and Oriental Languages, San Diego State University, December 2001. [invited]

Robert Malouf, Stochastic Head-Driven Phrase Structure Grammar, Human Communication Research Centre, University of Edinburgh, June 2001. [invited]

Robert Malouf, Practical and efficient default unification. Microsoft Research, Redmond, Washington. December 2000. [invited]

Gertjan van Noord and Dale Gerdemann, Finite State Transducers with Predicates and Identity, CLIN Tilburg, November 3, 2000.

Gertjan van Noord, Alpino: Wide Coverage Computational Analysis of Dutch, Computing with LLL Seminar, University of Amsterdam, June 15 2001 [invited].

Gertjan van Noord, with Dale Gerdemann, Invited Speaker at *SIGPHON 2000, Finite State phonology*, Approximation and Exactness in Finite State Optimality Theory. August 2000, Luxembourg.

Gertjan van Noord, Wide Coverage Computational Analysis of Dutch. University of Sussex. Brighton. February 21 2002 [invited].

Gertjan van Noord, Wide Coverage Computational Analysis of Dutch. Johns Hopkins University. Baltimore. April 9 2002 [invited].

Robbert Prins and Gertjan van Noord. Unsupervised POS-Tagging Improves Parsing Accuracy and Parsing Efficiency. BCN Poster Day. February 2002, Groningen.

Begoña Villada Moirón, Gosse Bouma. A corpus-based approach to the acquisition of collocational prepositional phrases. CLIN 2001, Enschede. November 30 2001.

Begoña Villada Moirón. Extraction of collocational prepositions. BCN Poster Day. February 2002, Groningen.

Appendix C

Other Research Activities

Gosse Bouma:

- program committee *Learning Language in Logic*, Lisbon, 2000.
- program committee *International Conference on HPSG*, Berkely, 2000.
- member *Corpusannotatie* NWO project Corpus Gesproken Nederlands.
- member coordinating committee *Elektronisering van de ANS*, Nederlandse Taalunie.
- member *Platform for Taal- en Spraaktechnologie*, Nederlandse Taalunie.
- editorial board *Computational Linguistics*
- program committee *Formal Grammar* 2002.
- program committee *LREC Workshop Beyond Parseval: towards Improved Evaluation Measures of Parsing Systems.*
- program committee ESSLLI 2003.

Jan Daciuk:

- reviewer for Computational Linguistics and Natural Language Engineering
- reviewer for workshop *Finite State Methods in Natural Language Processing.* Helsinki. 2001

Tanja Gaustad:

- organiser TABU-day, one-day conference on general linguistics, University of Groningen, June 22 2001.
- organiser 13th CLIN Meeting (Computational Linguistics in the Netherlands), to be held 29 november 2002, University of Groningen.

Robert Malouf:

- one week lecture at ESSLLI (with Miles Osborne), An Introduction to Stochastic Attribute-Value Grammars. Helsinki august 2001.
- one week lecture at HPSG Summer School, Statistics for Linguists, Trondheim, Summer 2001.
- KNAW Research Fellow as of January 1, 2002.
- HPSG-L electronic mailing list manager
- Reviewer for Computational Linguistics, Language and Computation, Natural Language Engineering, Natural Language and Linguistic Theory

Gertjan van Noord:

- area chair, *COLING*, Saarbrücken. 2000.
- tutorial chair *EACL/ACL* 2001 Toulouse, 2000.
- editorial board *Computer Speech and Language*.
- editorial board of *WEB-SLS*, The European Student Journal of Language and Speech.
- program committee *Workshop Using Toolsets and Architectures to build NLP Systems*. Luxembourg, 2000.
- program committee *Workshop Efficiency in Large-scale Parsing Systems*, Luxembourg. 2000.
- program committee *TAG+ Workshop*, Paris. 2000.
- program committee *TAG+ Workshop*, Venice. 2001.
- program committee *International Conference on HPSG*, Norway. 2001.
- program committee *International Workshop on Naturl Language Understanding and Logic Programming*. Copenhagen, 2002.
- co-chair workshop *Finite State Methods in Natural Language Processing*. Helsinki. 2001
- co-chair *20 years of two-level morphology*. August 2001. Helsinki. 2001.
- one week lecture at the LOT summerschool, Tilburg, June 2000 entitled *Finite State Language Processing*.
- co-promoter of Rob Koeling. Dialogue-Based Disambiguation: Using Dialogue Status to Improve Speech Understanding, 2002. University of Groningen.

Appendix D

Software and other resources

A number of software packages as well as a number of other resources are maintained by members of the Pionier group. These resources are freely availabe to other members of the research community (the detailed conditions of usage may vary).

Adfa Adfa is a program for testing various acyclic automata construction methods. http://www.eti.pg.gda.pl/ jandac/adfa.html

Alpino Treebanks Collection of corpora annotated with CGN dependency structures. http://www.let.rug.nl/ vannoord/trees/

Estimate Estimate is a program for parameter estimation of maximum entropy models. http://www.let.rug.nl/ malouf/maxent/

Fadd . Fadd is a library accessing dictionaries in form of finite-state automata, finite-state perfect hashing functions, and compressed finite-state language models (as produced by the s_fsa program). http://www.eti.pg.gda.pl/ jandac/fadd.html

FSA Utilities This is a collection of utilities to manipulate regular expressions, finite-state automata and finite-state transducers. Manipulations include automata construction from regular expresssions, determinization (both for finite-state acceptors and finite-state transducers), minimization, composition, complementation, intersection, Kleene closure, etc. http://www.let.rug.nl/ vannoord/Fsa

Minim A set of programs for testing automata minimization algorithms, and in particular Daciuk's version of the incremental algorithm by Bruce Watson. http://www.eti.pg.gda.pl/ jandac/minim.html

S_FSA A package of programs for construction and use of finite-state automata for morphological analysis, spelling correction, restoration of diacritics, and perfect hashing. http://www.eti.pg.gda.pl/ jandac/fsa.html

Hdrug Hdrug is an environment to develop logic grammars / parsers / generators for natural languages. http://www.let.rug.nl/ vannoord/Hdrug/

Stemmer/Lemmatiser A dictionary-based stemmer/lemmatiser for Dutch based on CELEX. http://www.let.rug.nl/ tanja/code.html

Appendix E

List of Project Members

Leonoor van der Beek AIO, started April 1 2001. Planned end-date April 1 2005.

Gosse Bouma UD at Alfa-informatica RuG. Some of his teaching is taken over by replacements which are being financed from the Pionier budget.

Jan Daciuk Postdoc. Started February 1 2000. End-date February 1 2003.

Tanja Gaustad AIO, started April 1 2000. Between april 1 2001 and october 1 2001, Tanja worked for an email classification project (*Kennisontwikkeling in Partnerschap* with Bussiness Support Center, Groningen). Therefore, planned end-date October 1 2004.

Robert Malouf Postdoc. From July 1 2001 until January 1 2002. Both before and after this period, Malouf participated in the project.

Tony Mullen Researcher. After his Ph.D. project, Mullen worked for three months for Pionier (January 1 2002 - April 1 2002). During this period he finished his Ph.D.

Robbert Prins AIO, started November 1 2000. Planned end-date November 1 2004.

Gertjan van Noord UHD at Alfa-informatica RuG. Some of his teaching is taken over by replacements which are being financed from the Pionier budget.

Begoña Villada Moirón AIO, started November 1 2000. Planned end-date November 1 2004.

Appendix F

Financial overview

The following table gives the financial situation of the project. All amounts in EURO.

	1999	2000	2001	2002	2003	2004	2005	total
Personnel:								
Daciuk		50166	57907	60417	5055			173545
Gaustad		19344	14167	30978	35573	27695		127757
Prins		4499	27781	29376	32415	30772		124843
Villada		5660	27855	29604	32866	29184		125169
Malouf			20244					20244
Mullen				11001				11001
vd Beek			18576	25394	28470	32813		113456
Teaching		11104	22949	51731	42867	14689	9999	153339
Further Costs	59	88075	10545	13613	13613	13613		139518
Reserved								190522
Total								1179394

- *Teaching* refers to a number of teachers we have employed which take over most of the teaching obligations of Gosse Bouma and Gertjan van Noord.
- *Further Costs* include travel money (for conference visits, etc.) and non-standard hardware. In the first year of the project we invested in a cluster of 7 Alpha Unix machines (64bit).
- *Reserved* includes reservation for the free Post-doc position, as well as additional teaching replacement, a.o.