POLITECHNIKA GDAŃSKA

Technical University of Gdańsk

Incremental Construction of Finite-State Automata and Transducers, and their Use in the Natural Language Processing

Jan Daciuk

Rozprawa doktorska ETI

May 25, 1998

Abstract

This dissertation states that it is possible to construct minimal deterministic finite-state automata fast and using little memory. Two new construction algorithms are presented. An implementation is discussed. Compared to a similar algorithm by Dominique Revuz, those presented here use far less memory. The thesis states that it is possible to construct automata that guess canonical forms and categories of unknown words much faster than it is done by other algorithms. A new algorithm is given and discussed. An overview of the use of finite-state automata in natural language processing (NLP) is given. A new type of automata is introduced. A method for spelling correction is enhanced so that it can handle Polish words.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The work presented here should be seen as a part of a greater framework. The goal is to create a system for computer-assisted correction of written texts. However, the algorithms presented in this thesis are general, and they can be used in a variety of applications. We are concerned mostly with natural language processing (NLP) applications, but finite automata are useful in other domains, such as compilers, text processors, DNA sequence matching, VLSI design, etc. Most natural language processing (NLP) applications have layered architecture (fig. 1.1). In a real-life NLP system, some layers may be missing. The higher the level is situated on the diagram, the greater the possibility that it is missing. This happens for two reasons. The first is that the lower levels are much more easier to implement. The second one is that we can do without them in many applications without a significant loss of performance. The boundaries between layers are blurred. One cannot perform a good syntactic analysis without the knowledge of semantics, a good semantic analysis without pragmatics, etc. Statistical methods can be used in much the same way on all levels. However, the divisions are useful, because they correspond to different modules, or groups of modules, in NLP software.

The bottom layer is the lexicon and the morphology. The morphology provides information about features of individual words. It is done either by dictionary lookup, or by computation. The set of features depends on the requirements of other system components. The features that are included most

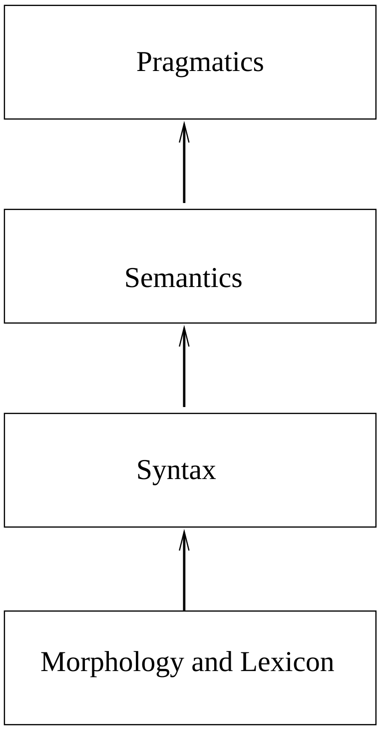


Figure 1.1: NLP layers

frequently are morphological in nature, i.e. such as part-of-speech, number, gender, etc. Other features to be considered are subcategorization and meaning. All algorithms presented in this thesis can be situated in this lowest layer.

The layer can be implemented in a variety of ways. The method chosen here is to use finite-state automata and transducers. There are two main features of automata that justify the choice:

- small size,
- great speed.

They should be seen jointly in comparison with other methods.

We use deterministic, acyclic finite-state automata (DAFSA). Acyclic automata are particularily suited to represent dictionaries, as they accept finite languages – a finite set of words, which corresponds to a definition of a dictionary. If we extend the notion of a word to represent an inflected form with annotations, we can construct morphological dictionaries – dictionaries that contain not only words, but their canonical forms and morphological categories as well. Deterministic automata are much faster in recognition than their nondeterministic counterparts, as recognizing a word means following a single path in an automaton – no (costly) backtracking is required.

The use of finite-state automata in NLP is not new. What is new is the incremental method of their construction, which lowers the memory requirements. Also, the type of automata we use is original. They do not have final states – they have final transitions. The origins of the work presented here date back to a lecture given by Lauri Karttunen ([Kar94]) on a seminar in Archamps, France.

Chapter 5 discusses a new original method for constructing automata that guess the lexemes and the morphological categories that correspond to unknown inflected word forms. These automata are constructed from already existing morphological lexicons. Although a method for guessing using a rule inference from a lexicon was proposed by Andrei Mikheev ([Mik97]), and the conversion of rules to finite-state transducers proposed by Emmanuel Roche and Yves Schabes ([RS95]), they both use Eric Brill's learning paradigm.

The present dissertation upholds the following theses:

- **It is possible to construct minimal, deterministic, acyclic, finite-state automata using less memory than in all previously known algorithms while maintaining the speed of the construction process.**
- **It is possible to construct automata that guess canonical forms of words and their annotations directly from data for a morphological lexicon. Little or no linguistic knowledge is required for that process. We state that this process can be faster than the previously known algorithm serving the same purpose, and that it maintains similar quality of guessing.**

We realize those objectives by:

- Proposing two new construction algorithms: one for sorted data, one for unsorted data. While both have the same computational complexity, the first one runs faster and makes it possible to perform some operations on the automaton while it is being constructed, and the second one is more flexible as it accepts data coming in arbitrary order, possibly from another process.

- Proposing a new way of constructing a guessing automaton. That method is direct. While in the current implementation the construction of such an automaton is divide into two phases: construction of an automaton holding the data for a morphological lexicon, and application of rules that prune it, these two phases can be performed synchronously.

1.2 Related Work

For an excellent general introduction to automata, see [HU79]. A taxonomy of finite-state automata minimization algorithms can be found in [Wat93b], and a related taxonomy of finite-state automata construction algorithms can be found in [Wat93a]. Both papers discuss finite-state automata in general, and they do not include the algorithms that are described here. The algorithms that are presented here are for deterministic, acyclic FSAs. Since they are specific, they are faster than traditional ones, and they use less memory. They could form a separate branch in Watson's taxonomy. The algorithm for data in arbitrary order was also developed independently by Richard E. Watson and Bruce W. Watson. An alternative way for an incremental construction of automata and transducers was proposed by Dominique Revuz ([Rev91]). While doing a pseudo-minimization during the construction phase, and economizing the use of memory, his algorithms construct the minimal automaton in two steps, the second being the proper minimization. The first stage results in a automaton that is not minimal and that needs more memory than the minimal one.

The classical work on automata and two-level rules in morphology is [KK94]. Although that paper was published in 1994, its unpublished versions were in circulation from about 15 years earlier. Other fundamental papers from this field are two articles ([Kos83] and [Kos84]) by Kimmo Koskenniemi, all influenced by [KK94]. Two-level rules and their implementation as transducers are discussed in [RRBP92] and [Spr92], probably the most popular books on morphology. A survey on the use of transducers in NLP is presented in [Moh97]. Several articles on the subject are in a book [RS97] edited by Emmanuel Roche and Yves Schabes.

The (physically) smallest automata are produced by using the concepts and algorithms developed by Tomasz Kowaltowski *et al.* ([KLS93b]). They introduce the notion of binary automata, and perform a very efficient optimization on the bit level (see section 4.4).

The morphological analysis of unknown words is discussed by Eric Brill ([Bri93] and [Bri95]), Andrei Mikheev ([Mik97]), and mentioned by Emmanuel Roche and Yves Schabes ([RS95]). Eric Brill tries to discover rules governing the assignment of morphological categories to words. The assignment is based on prefixes and endings, and on context. They need an annotated corpus to learn the rules. Various rules generated from templates are applied to the corpus in turns, the best one, i.e. the one that gives categories that are closest to those in corpus is chosen and applied to the corpus, and the process continues until new rules do not improve the results. Emmanuel Roche and Yves Schabes ([RS95]) show how the rules can be transformed into a subsequential transducer which results in significant improvement in processing speed, and provides compact representation of rules. Andrei Mikheev ([Mik97]) uses a morphological lexicon instead of corpus to learn the rules using Brill's algorithm. Statistical techniques (e.g. [WMS+93]) have also been used for this purpose.

Spelling correction has a rich bibliography. Three classical works are [Dam64], [Pet80], and [DLS83]. Those papers deal with isolated words. Systems that use context in spelling correction have also been described ([CH83], [CDK+91], [CDGK92], [CGM93], [HJM+82], [JHMR83], [KS81]). Shallow parsing for spelling correction was advocated in [Atw87]. A pioneering work on the use of finite-state methods in spelling correction was done by Kemal Oflazer ([OG94] and [Ofl96]).

Perfect hashing by using finite-state automata is described in a tutorial ([Roc95]), which was presented at the ACL'95 conference.

1.3 Overview of the Dissertation

Chapter 2 introduces basic concepts and definitions that are used in the rest of this work. They are based mostly on [HU79], [Wat93a], [Wat93b], [RS95], and [Moh97]. A new type of automata is presented and discussed.

Chapter 3 describes input data formats for programs that build automata. That format is not interpreted by those programs, but it is interpreted by applications that use automata.

Chapter 4 presents two algorithms for constructing deterministic, acyclic finite state automata. The algorithms differ by their efficiency and by the need to sort input data. The problem of representation of automata (i.e. storing them) is discussed.

Chapter 5 describes an algorithm, or a family of algorithms for constructing a finite-state automaton that can be used for guessing lexemes and categories of inflected forms. The algorithm uses the data for an existing morphological lexicon. The data is prepared in a special way, an automaton is built out of it, and then the algorithms prune it to obtain some generalizations and a smaller size.

Once the automata are built, they can be used for a variety of applications in the NLP domain. Chapter 6 discusses the algorithms that use automata for spelling correction, diacritic restoration, morphological analysis, and perfect hashing.

The dissertation ends with a summary.

1.4 Acknowledgements

Many people contributed (in various forms) to this thesis. They are: Susan Armstrong (ISSCO, Université de Genève), Pierrette Bouillon (ISSCO, Université de Genève), Thierry Etchegoyhen (LATL, Université de Genève), Martin Kay (Xerox), Maghi King (ISSCO, Université de Genève), Henryk Krawczyk (Technical University of Gdańsk), Christopher Laenzlinger (LATL, Université de Genève), Sabine Lehmann (ISSCO, Université de Genève), Jacek Martinek (Poznań University of Technology), Tomasz Obrębski (Poznań University of Technology), Dominique Petitpierre (ISSCO, Université de Genève), Michael Rosner (University of Malta), Graham Russel (ISSCO, Université de Genève), Paweł Siwak (Poznań University of Technology), Stanisław Szejko (Technical University of Gdańsk), Zygmunt Vetulani (Adam Mickiewicz University in Poznań), Bruce W. Watson (Ribbit Software, Canada), Richard E. Watson (Ribbit Software, Canada), Eric Wehrli (LATL, Université de Genève).

I would also like to express my gratitude to the Swiss Federal Scholarship Commission for providing me with a scholarship that covered my stay at ISSCO, Geneva, and to the Faculty of Electronics, Telecommunications and Informatics of the Technical University of Gdańsk for supporting my research with grants.

Chapter 2

Basic Definitions

There are many definitions of automata and transducers. The differences between them consist mainly in the choice of symbols describing particular items, in the way transitions are defined – as a function, or as a relation, with various orderings of constituents, and whether the ϵ-transitions are distinguished in the definition. Notational differences are not important from the theoretical point of view, but the choice of a particular suite of definitions and notational conventions can simplify the description of a problem, thus contributing to the clarity of explanations. The suite of definitions given in this paper is based on two Watson's papers ([Wat93a] and [Wat93b]), but is converted to the notational convention found in [HU79], which is regarded as the standard reference for automata. However, we use different (equivalent) definitions where it enhances the clarity of our discourse. The definitions for finite-state transducers are taken mainly from [RS95].

We introduce a new type of automata: finite-state automata with final transitions. A transformation from the classical automata to the automata with final transitions is provided. We also discuss deterministic acyclic finite-state automata with final transitions. These automata are used in our implementation of the algorithms presented in this dissertation.

2.1 Finite State Automata

A finite-state automaton is a device that can be in one of a finite number of *states*. In certain conditions, it can switch to another state. This is called a *transition*. When the automaton starts working (when it is switched on), it can be in one of its initial states. There is also another important subset of states of the automaton: the final states. If the automaton is in a final state when it stops working, it is said to *accept* its input. The input is a sequence of symbols. The interpretation of the symbols depends on the application; they usually represent events, or can be interpreted as "the event that a particular data became available". The symbols must come from a finite set of symbols, called the *alphabet*. If a particular symbol in a particular state triggers a transition from that state to another one, that transition is *labeled* with that symbol. The labels of transitions can contain one particular symbol that is not in the alphabet. A transition is labeled with ϵ (not present in the alphabet) if it can be traversed with no input symbol.

It is convenient to present automata as directed graphs. The vertices denote states. They are portrayed as small circles. The transitions form the edges – arcs with arrows pointing from the source state (the state where the transition originates) to the target state. They are labeled with symbols. Unless it is clear from the context, the initial states have short arrows that point to them from "nowhere". The final states are represented as two concentric circles.

The example automaton (fig. 2.1) contains 6 states labeled A, B, C, D, E and F. The state labeled A is initial. The states A, C, and D are final. The automaton contains no cycles, as it is impossible to leave a state, follow some transitions, and return in this way to the same original state.

Starting in one of the initial states, the automaton processes a sequence of input symbols. In a given state, it checks if the next input symbol matches any of the labels of the transitions that go out of the state. If so, the automaton follows it to the target state of the transition. If ϵ is among the labels, the transition it labels is followed without reading the input. There may be more than one transition with the same label, or a transition with a matching label and a transition labeled with ϵ. In that case, the decision which path to choose is arbitrary. If there is no transition that matches the input symbol (in particular, if there is no ϵ-transition), the automaton stops and rejects the input. The input is accepted when all input is read and matched by transitions and the automaton is in a final state.

A set of sequences of symbols gathered by pursuing all paths from initial states to the final ones is called the *language* accepted by the automaton. E.g. the example automaton (fig. 2.1) accepts the language $\{\epsilon, a, ac, ae, bc\}$. The symbol ϵ, meaning an empty input, is accepted because the initial state A is also the final one. a is accepted, because there is a transition from A to

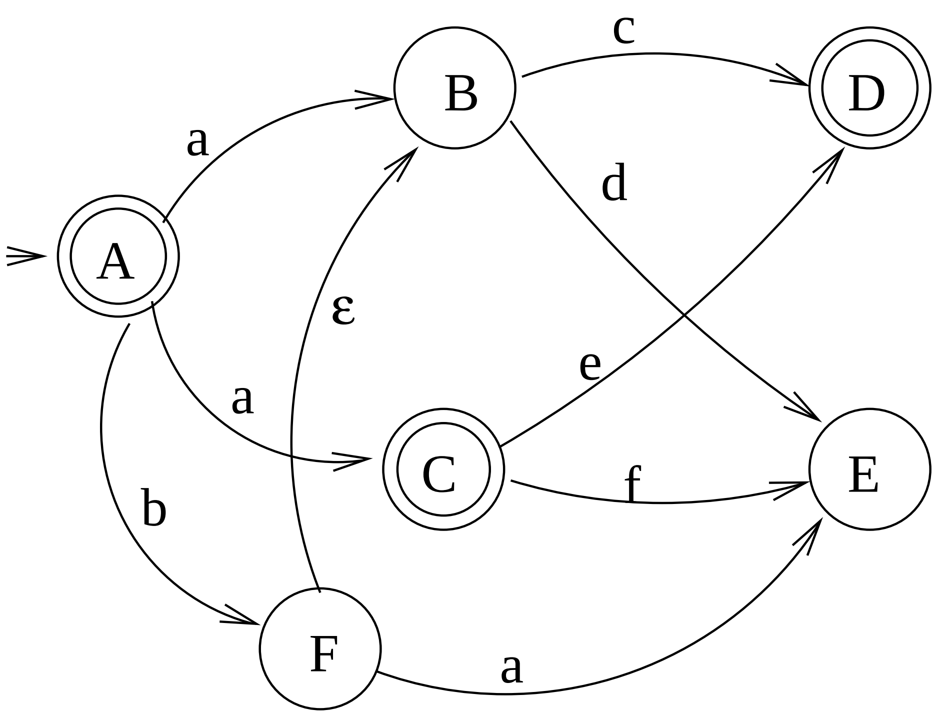


Figure 2.1: Sample finite-state automaton

C labeled with a, and C is a final state. The input bc is accepted, because the control in the automaton can go from A to F on a, and then it can follow the ϵ-labeled transition to B, and then a transition labeled c to D, which is a final state. The sequences: {ad, ae, af, ba} are all rejected, because E is not a final state. The language of the automaton in fig. 2.1 is finite, because the automaton contains no cycles.

An automaton that has no ϵ-labeled transitions is called ϵ-free. An automaton that has one initial state, is ϵ-free, and for each state there is at most one transition labeled with the same symbol is called a *deterministic finite-state automaton*. The automaton from fig. 2.1 can be transformed into the deterministic automaton from fig. 2.2.

A state is said to be *reachable* from another state iff there is a sequence of transitions leading to the state from that other state. An automaton is said to be *start-useful* iff every state is reachable from any of its initial states. An automaton is *useful* iff it is start-useful and for every state there is at least

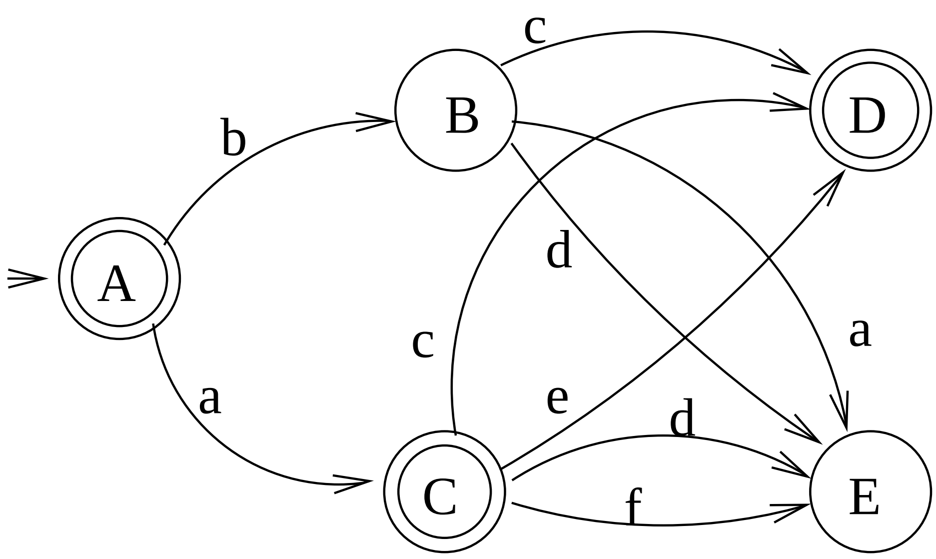


Figure 2.2: Deterministic version of the sample automaton

one final state reachable from that state. The automaton from the fig. 2.2 is not useful; its useful version is presented in fig. 2.3. That version accepts the same language.

An automaton is said to be *acyclic* when it contains no cycles, i.e. it is not possible to arrive at the same state twice when following the transitions. All automata in figures presented in this dissertation are acyclic.

Formally, an automaton is a 5-tuple $M = (Q, \Sigma, \delta, i, F)$, where Q is a set of states, $i \in Q$ is a set of initial (or starting) states, $F \subseteq Q$ is a set of final states, Σ is the alphabet, and δ is a partial mapping $\delta : Q \times (\Sigma \cup \epsilon) \not\longrightarrow \mathcal{P}(Q)$ denoting transitions. We extend the δ mapping to $\delta^* : Q \times \Sigma^* \not\longrightarrow \mathcal{P}(Q)$. The size of the automaton, $|M|$, is equal to the number of states $|Q|$. We define the language accepted by the automaton M to be:

$$\mathcal{L}(M) \equiv \{x \in \Sigma^* | \delta^*(q_0, x) \cap F \neq \emptyset\}$$

Let the mapping $\overrightarrow{\mathcal{L}}_M \in Q \longrightarrow \mathcal{P}(\Sigma^*)$ be the *right language* of a state q in M (the set of all strings over Σ^* that will take one from state q to any final state of M using the extended transition relation δ^*):

$$\overrightarrow{\mathcal{L}}_M\,(q) \equiv \{x \in \Sigma^* | \delta^*(q, x) \cap F \neq \emptyset\}$$

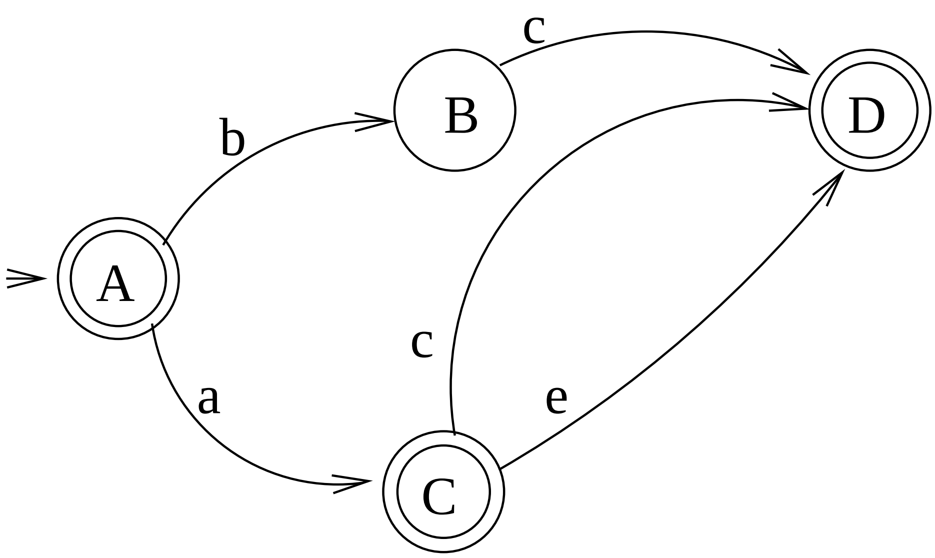


Figure 2.3: Useful version of the sample deterministic automaton

The consequence of the Myhill-Nerode theorem ([HU79]) is that among many automata that accept a given language there is one (up to isomorphism) that has a minimal number of states. It is called a *minimal automaton.* The automaton in fig. 2.3 is minimal.

The property of minimality of is defined as follows ([Wat93a], [Wat93b], [Wat95]):

$$Min(M) \equiv (\forall_{M':M' \in FSA \wedge \mathcal{L}_{FSA}(M) = \mathcal{L}_{FSA}(M')} |M| \leq |M'|)$$

We will use, however, an alternative definition of minimality of $M = (Q, \Sigma, \delta, S, F)$ ([Wat93a], [Wat93b]):

$$Minimal(M) \equiv (\forall_{q_1, q_2 : q_1 \in Q \wedge q_2 \in Q \wedge q_1 \neq q_2} \overrightarrow{\mathcal{L}}(q_1) \neq \overrightarrow{\mathcal{L}}(q_2)) \wedge Useful_s(M)$$

$Useful_s$ is a property specifying that all states can be reached from the start state, and it is defined as follows:

$$Useful_s(M) \equiv (\forall_{q \in Q} \exists_{x \in \Sigma^*} q \in \delta^*(q_0, x))$$

Of particular interest for the NLP community are deterministic acyclic finite state automata (DAFSA) , because they are best suited for representing dictionaries (of various sorts). Whenever we refer to an automaton (or

FSA) throughout this paper, we mean an acyclic, deterministic finite state automaton (DAFSA), unless explicitly stated otherwise. Several definitions given above can be rewritten in a simpler way: a deterministic, finite-state automaton is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a set of states, $q_0 \in Q$ is the starting state, $F \subseteq Q$ is a set of final states, Σ is the alphabet, and δ is a partial mapping $\delta : Q \times \Sigma \not\longrightarrow Q$ denoting transitions. We extend the δ mapping to $Q \times \Sigma^* \not\longrightarrow Q$ as in [HU79]. The definitions of the language accepted by an automaton, the right language of a state, and of the Useful$_s$ property for DAFSA become simpler:

$$\mathcal{L}_{FSA}(M) = \{x \in \Sigma^* | \ \delta(q_0, x) \in F\}$$

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

$$Useful_s(M) \equiv (\forall_{q \in Q} \exists_{x \in \Sigma^*} \delta^*(q_0, x) = q)$$

It is possible to use alternative definition of an FSA: $M = (Q, \Sigma, i, E, F)$, where the meaning of Q, Σ, i, and F is the same as above, and E is a relation $Q \times \Sigma \times Q$ - an alternative expression of the δ function from above. This alternative definition may be more useful in descriptions below.

The algorithms described in this paper operate on classical DAFSA (deterministic acyclic finite-state automata). However, our implementation uses a different kind of automata: *finite-state automata with final transitions (FSA-FT)* . We define such automaton as $M = (Q, \Sigma, i, E, F)$, where Q is the finite set of states, Σ is the alphabet, $i \subseteq Q$ is the set of the initial states of the automaton, $E \subseteq Q \times \Sigma \times Q$ is a set of transitions, and $F \subseteq E$ is the set of final transitions. We say that the FSA-FT accepts a word if for each symbol from the input a transition was found, and the last transition it traversed while recognizing the word was final.

We define a transformation of a classical finite-state automaton into a finite-state automaton with final transitions: for every FSA $M = (Q, \Sigma, i, E, F)$ we find an FSA-FT $M' = (Q', \Sigma, i', E', F')$, where $Q' = Q \cup \{i'\}$, i' is the new initial state, $E' = E \cup \{(i', \epsilon, q_i) : q_i \in i$, and $F' = \{(q_i, a, q_j) \in E : q_j \in F\}$.

Let $\hat{F}$ to be the transitive closure of F defined by the following recursive relation:

- if $e \in F$ then $e \in \hat{F}$
- if $(q, a, q') \in E, (q', a', q'') \in \hat{F}$ then $(q, aa', q'') \in \hat{F}$

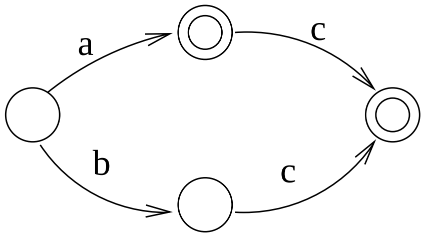
Now we can define the language of the FSA-FT to be:

$$\mathcal{L}_{FSA-FT}(M) \equiv \{x \in \Sigma^* | \; \exists_{q_1 \in \{i\}} \exists_{q_2 \in Q} (q_1, x, q_2) \in \hat{F}\}$$

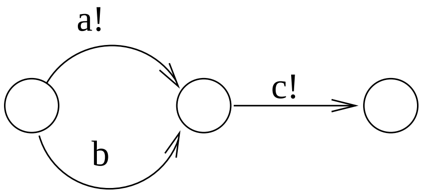
For every deterministic acyclic finite-state automaton $M = (Q, \Sigma, q_0, E, F)$ that does not accept empty input (written as ϵ), i.e. $\epsilon \notin L(M)$, or $q_0 \notin F$, we can find a deterministic acyclic finite-state automaton with final transitions (DAFSA-FT) $M' = (Q, \Sigma, q_0, E, F')$, $\mathcal{L}(M') = \mathcal{L}(M)$, where $F' = \{(q_i, a, q_j) \in E : q_j \in F\}$. We will call this transformation *moving markers.* It preserves the number of states and the number of transitions. Because such a transformation exists, and it is defined for the minimal automata, we can write that $|Minimal(M')| \leq |Minimal(M)|$.

Minimal deterministic acyclic finite-state automata with final transitions can indeed be smaller than their classical counterparts (see fig. 2.4). The definition of minimality for an FSA-FT is the same as for the classical FSA, but the definition of the *right language* of a state in FSA-FT must change:

$$\overrightarrow{\mathcal{L}}_{FSA-FT}(M) \equiv \{w \in \Sigma^* | \; \exists_{q_i \in Q} (q, w, q_i) \in \hat{F}\}$$



(a) Classical DAFSA accepting $\{a, ac, bc\}$



(b) DAFSA-FT accepting the same language. Exclamation point marks final transitions

Figure 2.4: Classical minimal deterministic acyclic FSA and the corresponding FSA with final transitions

Note that for DAFSA-FT $\forall_{q \in Q}\ \epsilon \notin \overrightarrow{\mathcal{L}}\ (q)$. This is different from the classical automata, where $\forall_{q \in F}\ \epsilon \in \overrightarrow{\mathcal{L}}\ (q)$. That difference is very important. It means that the recognition of a word is moved from a final state in the classical automaton to the transition *reaching that state* in the FSA-FT, resulting in the removal of the empty string (ϵ) from the right language of that automaton. Let $M = (Q, \Sigma, q_0, E, F)$ be a DAFSA (a classical one), and $M' = (Q', \Sigma, q_0, E, F')$ be a DAFSA-FT obtained by the transformation *moving markers.* We can write:

$$\overrightarrow{\mathcal{L}}_{DAFSA-FT}\ (q) = \overrightarrow{\mathcal{L}}_{DAFSA}\ (q) \setminus \{\epsilon\}$$

If we can find two states in the classical FSA that have the same right languages except for the empty string, we can merge the states together in a deterministic acyclic finite-state automaton with final transitions. This makes the resulting automaton smaller, i.e. it has less states and less transitions than the corresponding classical finite-state automaton.

Note that as most implementations of automata have implicit states (only transitions are represented explicitly), this new kind of automata saves space at no cost.

The automata with final transitions can be seen as Mealy's automata where the property of being final is the output label. Therefore, they can be perceived more as a notational convention than a new kind of automata. However in transducers, that property would have to be concatenated with the output labels.

2.2 Finite State Transducers

Transducers are automata that have transitions labeled with two symbols. One of the symbols represents input, the other – output. Transducers translate (or transduce) strings. In automata theory (see [HU79]) they are called Mealy's automata.

Transducers can be seen as automata with transitions labeled with symbols from $\Sigma 1 \times \Sigma 2$, where $\Sigma 1$ and $\Sigma 2$ are the alphabets of input and output, respectively. With such a perception in mind, all the definitions for automata hold for transducers. The alphabets $\Sigma 1$ and $\Sigma 2$ are frequently the same, so the definition ([RS95]) becomes simply $T = (\Sigma, Q, i, F, E)$, where Σ is a finite alphabet, Q is a finite set of states or vertices, $i \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $E \subseteq Q \times (\Sigma \cup \epsilon) \times \Sigma^* \times Q$ is the set of transitions or edges. The state *transition function* d that maps $Q \times (\Sigma \cup \{\epsilon\})$ into 2^Q is defined as $d(q,a) = \{q' \in Q | \exists_{w' \in \Sigma^*} (q, a, w', q') \in E\}$. The *emission function* δ maps $Q \times (\Sigma \cup \{\epsilon\}) \times Q$ into 2^{Σ^*}, and is defined as $\delta(q, a, q) = \{w' \in \Sigma^* | (q, a, w', q') \in E\}$.

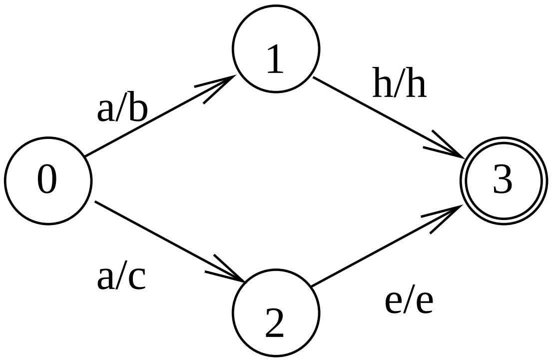


Figure 2.5: Simple transducer (adopted from [RS95])

Figure 2.5 shows a transducer T=({a,b,c,e,h}, {0,1,2,3}, 0, {3}, {(0,a,b,1), (0,a,c,2), (1,h,h,3), (2,e,e,3)}). It translates *ab* to *bh* and *ae* to *ce*.

Transducers translate or *transduce* strings. A transducer T is said to represent the transducing mapping f from Σ^* to Σ^*. Let $\hat{E}$ be the transitive closure of E defined by the following recursive relation ([RS95]):

- if $e \in E$ than $e \in \hat{E}$
- if $(q, a, b, q'), (q', a', b', q'') \in \hat{E}$ then $(q, aa', bb', q'') \in \hat{E}$

Then the transducing mapping $f(w) = w'$ iff $\exists_{q \in F} (i, w, w', q) \in \hat{E}$. One writes $f = |T|$. In the example figure, $|T|(ah) = bh$ and $|T|(ae) = ce$. If, for some input w, more than one output is allowed, then f is called a *rational transduction*. Otherwise it is called a *rational function*.

Because transducers are used as a device to compute a function (or to translate one string into another), a different definition of being deterministic is used. In that definition, a transducer is said to be deterministic iff both the transition function d and the emission function δ lead to sets containing at most one element. More formally, $\forall_{q \in Q} \forall_{a \in \Sigma} |d(q,a)| \leq 1 \wedge |\delta(q, a, q')| \leq 1 \wedge |d(q, \epsilon)| = \emptyset \wedge |\delta(q, \epsilon, q')| = \emptyset$. Traditionally, such deterministic transducers are called *subsequential* (sequential transducers are deterministic transducers for which all states are final). For subsequential transducers, the transducing function can be computed by deterministically following a single path in the transducer.

The transducer in figure 2.5 is not subsequential; it computes a rational function. Because both transitions leaving the first state have a as their input labels, ($d(0, a) = \{1, 2\}$) one cannot determine which path to follow without looking at further input.

Subsequential transducers are defined as seven-tuples $T = (\Sigma, Q, i, F, \otimes, *, \rho)$, where the meaning of Σ, Q, i, and F is the same as in simple transducers described above, $\otimes$ is the deterministic state transition function that maps $Q \times \Sigma$ on Q, $*$ is the deterministic emission function that maps $Q \times \Sigma$ on Σ^* (this function should not be confused with the Kleene's star, which is written as a superscript), and the final emission function ρ maps F on Σ^*. One writes $q \otimes a = q'$, $q * a = w'$ and $\rho(w) = w$. Figure 2.6 shows a subsequential transducer that accepts the same language as the transducer from fig. 2.5.

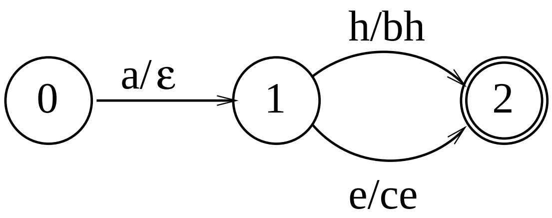


Figure 2.6: Subsequential transducer - a transformation of the transducer from fig. 2.5 (taken from [RS95])

Subsequential transducers are faster in recognition tasks, but may take more space. They are faster, because one always follows a single path in the automaton, and there is no need to go back, as it is the case with simple transducers. They may take more space as one needs to store strings instead of characters (symbols) in transitions.

Chapter 3

Data for Applications

In this chapter, we discuss the problem of what should the automata contain (what languages they should accept) in various applications. The construction algorithms perceive data as lists of words, with words being a sequence of symbols from a given alphabet. They do not interpret the symbols themselves. However, the programs that use the automata do interpret them, and they expect the words in automata to be formed in a particular way.

The data formats discussed here are appropriate for the algorithms presented in the next two chapters, i.e. they are suitable for the incremental construction of automata and transducers. Although this does not mean they cannot be used with other construction algorithms, this does not mean they are suitable for every construction algorithm either. In fact, there is a whole family of methods ([Wat93a], also see section 4.1) that build automata from regular expressions. Another widely used method for building lexical transducers ([KK94], [Kar94]) is to merge transducers representing lexemes and two-level rules by using intersection and composition. That method makes it possible to construct transducers directly form morphological descriptions of lexicon entries. Finally, our algorithms deal with lexicalized data, while automata and transducers can be used as an implementation of translating rules. For example, two-level rules are compiled into transducers ([Kos83], [Kos84]), and simple automata can be used to model phonological phenomena ([BE94]).

The disadvantage of the data format presented here is that the data takes more space than e.g. regular expressions. However, it can be used on-the-fly as it is produced, i.e. occupying no space at all. In addition, this format is flexible, as it does not assume that a particular tool was used to form the data.

We believe that the method for coding the lexemes by specifying how many characters from the inflected form are present in it, and then appending the suffix of the inflected form is a common knowledge among researchers in the field. However, we are not aware of the use elsewhere of the methods for coding prefixes and infixes that we describe here.

3.1 Spelling and Restoration of Diacritics

There are various algorithms for spelling correction and restoration of diacritics, so they may require various data. Most of them perform a simple dictionary lookup, and if the word is not found, they arrive with a list of similar words. These algorithms need a simple list of words for their lexicons. Thus, a word in the language accepted by the automaton is a word of the natural language without additional annotations.

More advanced algorithms check the word's context. This is done with a form of syntactic analysis, which can be full or shallow. Syntactic analysis requires morphological data, such as that described in the next section.

3.2 Morphological Lexicon

The specific form of data for morphological lexicon depends on the kind of automata to be used (simple automata – acceptors, or transducers).

In morphology, there are two levels: the strings come from the surface level, and the lexical level. The surface string is an inflected form of a word. E.g. the string *spała* is an inflected form (one of many) of the word *spać*. The lexical string is the corresponding lexeme with morphological annotations. The lexeme is the word in a form that can be found as the main entry for that word in a dictionary. E.g. the lexeme corresponding to the inflected form *spała* is *spać*. The annotations describe the properties of a word form, e.g. for the word *spała*, we could note that it is a verb in past tense, third person, female, singular, imperfect.

We may need morphology for various purposes. If it is tagging (annotating a corpus with part-of-speech labels), we do not need lexemes; only categories matter. For stemming, which is widely used in e.g. information retrieval, we retain lexemes, but suppress categories.

3.2.1 With Transducers

The whole concept of transducers consists in translating one string into another. Therefore, it is possible to translate the surface string into the lexical string, and vice versa. The translation can either be stored in a lexicon, or generated from translation rules. The latter approach can be found in numerous sources, e.g. [KK94], [Kos83], [Kos84], [RRBP92], [RS95], and [Spr92]. We do not describe it here, and we do not use it.

The lexicon-based method (see [Kar94] or [Moh94a]) is very simple. The paths in a transducer describe translations of strings (fig. 3.1).

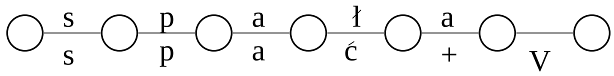


Figure 3.1: Morphology of *spała* as a path in a transducer. Categories kept to a minimum due to lack of space (only part-of-speech)

Therefore, the format of data can be as simple as possible. In our implementation, one line of input data has 3 HT-separated fields (HT means horizontal tabulation character, aka. TAB): inflected form, lexeme, and categories (fig. 3.2).

```
spała spać V[aspect=imperf mode=ind tense=past num=sg pers=3 gender=fem]
```

Figure 3.2: Transducer input data for "spała"

The surface and lexical strings may not be of the same length. Special filler characters must be used to pad the shorter string. They can be used, however, inside the strings as well. By inserting the filler characters at well-chosen places one can reduce the size of the transducer by increasing the number of identical transitions. The well-chosen places are those that allow for alignment of corresponding segments in both strings.

Note that such representation leads to small devices, as the beginnings of words consist mainly of pairs of identical characters, and the endings have the same mappings of surface ending (the ending of the inflected form) to lexical ending. Also, the analysis is simple, as no decoding is necessary.

3.2.2 With Automata – Acceptors

Simple automata (acceptors) can be used for storing a morphological lexicon. A single word of the language accepted by the automaton should represent morphological information for one inflected word form. We no longer have the two-level structure embedded in the transducer, therefore we need to express that structure inside the word of the language accepted by the automaton.

We divide the word into two parts: the inflected form, and annotations separated from the inflected form by an annotation separator. The annotations may contain lexemes and morphological categories. They have their own internal structure.

In order to minimize the size of the automaton, there should be an order imposed on all categories that can appear in the same entry. The categories should always be written in the same order. Categories that are present in most entries should be written last, i.e. the first category should be the most specific, and the last the most general.

Including lexemes in full could inflate the lexicon to a horrible size. This is because automata compress the beginnings and endings of strings, but are less good at compressing the inner parts of strings. The word's stem (or root) may be specific to that word. If not, it occurs in few words (i.e. few lexemes). The beginnings are compressed because there are many words that share a few letters at their starts. The endings are shared with many words. But when we put the inflected form and the lexeme one after another, we have twice the central part of the stem, with an ending (each time different) in between. The sequence of those three items: the stem, the ending, and once again the stem is unique, and it cannot be shared!

There is a solution to this problem. As the inflected form and the lexeme share the word-specific information, i.e. the stem, it needs to occur once. It is sufficient to indicate how many letters from the end of the inflected form should be deleted, and what should be added to the stem to obtain the lexeme. We use a code to indicate the number of letters that should be deleted from the end of the inflected form. "A" means nothing is to be deleted, "B" – 1 letter, "C" – to letters, and so on. The code is followed by the ending of the corresponding lexeme (fig. 3.3).

Inflected forms may have prefixes that are not present in lexemes. As the beginning of inflected form is different from the corresponding lexeme, whole lexemes must be stored, and so they inflate automata (see fig. 3.4). The prefixes may influence categories.

```
spała+Cć+V[aspect=imperf mode=ind tense=past num=sg pers=3 gender=fem]
```

Figure 3.3: Format of input data for "spała" for a simple automaton – acceptor

```
najszybszy+Kszybki+Adj[deg=sup gender=masc num=sg case=nom]
```

Figure 3.4: Prefixes inflate automata by enforcing storing lexemes in full form

In order to decrease the influence of the prefixes on the size of the automaton, a method similar to that described above can be used. A code (the name *prefix code* will be used here) may be introduced to indicate how many letters from the beginning of the inflected form should be deleted. “A” means there are no prefixes, “B” – 1-letter prefix, “C” – 2-letter prefix, etc. The code is inserted in front of the code indicating how many letters to delete from the end of the word. Figure 3.5 shows the morphological data for *najszybszy* – the same as in fig. 3.4 – but this time with prefixes coded to save space. The first “D” after the annotation separator means that the first 3 (i.e. “D” - “A”) letters from *najszybszy* should be deleted, giving *szybszy*. The second “D” says that 3 letters from the end of *szybszy* should be deleted, giving *szyb*. Finally, *ki* is appended to *szyb* giving *szybki* – the lexeme.

```
najszybszy+DDki+Adj[deg=sup gender=masc num=sg case=nom]
```

Figure 3.5: Coded prefix in data for morphological analysis with automata – acceptors

For the Polish morphology containing 1748097 words and expressions, the automaton with coded prefixes was 15.6% smaller than that without coded prefixes. For French, the automaton with coded prefixes would be bigger than the original one, because French does not have flectional prefixes.

Word structure can be more complicated. In German, the verbs can have two parts. The first part is loosely connected to the second one, and can be separated from it by an infix, or can be put separately somewhere else in a sentence. For example, *eingeladen* and *ladet ein* are two valid forms of the verb *einladen*. In order to cope with the problem, a code and a special annotation need to be introduced. The code should be inserted after the prefix code when the prefix code says that there is a prefix, i.e. when it is different from "A". The code (the name *infix code* will be used here) specifies the offset from the beginning of the inflected form of the characters to be deleted. The convention is the same as for other codes, i.e. "A" means the letters to be deleted are right at the beginning (true prefix), "B" means that the letters to be deleted start at the second letter of the inflected form, and so on. Figure 3.6 shows the entry for *eingeladet*. The first annotation separator is followed by 3 capital letters: "C" – 2 characters are to be deleted, "D" – the characters to be deleted start after the third character of the inflected form, "B" - one character should be deleted form the end of the inflected form. The string *ki* that follows the capital letters is the suffix of lexeme.

```
eingeladet+CDBn+PP[num=sg gender=masc case=nom deg=pos]
```

Figure 3.6: Coded infix in morphological data

The special annotation is similar to the method used to represent prefixes in the next section. Between the inflected form and the lexeme, another annotation separator can be used to introduce the *removable prefix*. Figure 3.7 shows how to indicate that *ladet* can be a part of *ladet ein* (this *ein* can be separated form *ladet* by many words).

```
ladet+ein+ABn+Verb[mode=ind tense=pres num=sg pers=3]
```

Figure 3.7: Removable prefix coding in an automaton for morphological analysis

3.3 Morphological Analysis of Unknown Words

For most of flectional languages, such as Polish, French, or German, the part that carries the information about the categories of the word, and about the ways of obtaining the canonical form, is contained in the ending of the inflected form. As we intended to recognize word endings, our lexicon is constructed from inverted words, i.e. the last letter of a word is the first letter of the string stored in the lexicon, the penultimate one – the second one, and so on. The beginning of a word is marked with a special character to make it possible to distinguish complete words from suffixes. We append annotations at the end of the inverted word. Another special character (an annotation separator) separates words from annotations.

As before, we use the term "annotation" to refer to everything past the annotation separator. Again, the annotations may include lexemes and morphological categories. If lexemes are to be included, they need the same (or similar) coding scheme as shown in the previous subsection.

The annotations are not inverted. E.g. the word *spała*, which could have annotations *Cć+V[aspect=imperf mode=ind tense=past num=sg pers=3 gender=fem]*, may be represented as shown in fig. 3.8.

```
ałaps_+Cć+V[aspect=imperf mode=ind tense=past num=sg pers=3 gender=fem]
```

Figure 3.8: Input data for "spała" for a morphological guesser

Prefixes introduce problems similar to those in automata representing lexicons. If lexemes are to be guessed, they cannot be represented in full form, as they would inflate automata. Exactly the same coding scheme as that described in the previous subsection can be used to solve that problem.

We cannot assume that categories of inflected forms depend only on suffixes. This is true e.g. for French, but there are many other languages that do not follow that rule. In Polish, superlatives of adjectives and adverbs are formed by adding a prefix "naj" to corresponding comparatives. In German, past participles are formed by adding a prefix "ge", and a suffix "t" that is used in other forms.

Prefixes also inflate guessing automata, because the parts that distinguish between various annotations are at beginnings of words. Transitions that code them cannot be removed, and neither can all the transitions that precede them. So where prefixes are used, whole words are stored in the automaton.

There is, however, a solution to this problem. We augment the annotations so that we can move there the prefixes from inflected forms. To separate prefixes from other annotations we use another annotation separator. E.g. for a Polish adjective "szybszy" ("faster") we have the input data shown in fig. 3.9.

$\boxed{\text{yzsbyzs\_++Dki+Adj[deg=sup gender=masc num=sg case=nom]}}$

Figure 3.9: Input data for "szybszy" for a morphological guesser

This means that "szybszy" is an inflected form of an adjective in comparative degree with given categories, that this form has no prefix, and that the corresponding lexeme is formed from "szybszy" by removing "D'' − "A'' = 3 letters from the end, and appending "ki". The data superlative form of the same adjective is given in fig. 3.10.

$\boxed{\text{yzsbyzs\_+naj+Dki+Adj[num=sg gender=masc case=nom deg=sup]}}$

Figure 3.10: Input data for "najszybszy" for a morphological guesser

This means that the inflected form is formed from "szybszy" by prepending the prefix "naj" (thus giving "najszybszy"), and that the lexeme is formed from "szybszy" by removing 3 last letters and adding "ki". Note that endings of "szybszy" and "najszybszy" are the same, but the forms have different morphological categories.

By moving prefixes to annotations, we achieved the situation where the inflected form (without the prefix) has the same beginning as the corresponding lexeme (at least in regular words), so to obtain the lexeme from the inflected form we need to cut some letters from the end of the inflected form and append some others, just as if there were no prefixes.

Unfortunately, this approach does not solve all problems with the German past participle. Some verbs have prefixes that can be separated from stems. In such cases the prefix "ge" becomes an infix, e.g. the past participle for "einladen" is "eingeladet". The solution is the same as in case of data for morphological analysis: an additional code (the *infix code*) specifies the offset of the infix from the beginning of the word. The code is inserted in front of the infix. If there is no infix, and the inflected form contains a prefix, the code says that the offset is zero, i.e. the code is "A". The code for *eingeladet* is given in fig. 3.11. The 3 capital letters after the first annotation separator specify what and where should be deleted from the inflected form before the suffix n could be appended to obtain the lexeme: "C" means there are 2 characters to be deleted near the beginning of the inflected form, "D" – the characters to be deleted are after the third character of the inflected form, "B" - the last character of the inflected form should be deleted.

The removable prefix can be removed form the inflected form, and put somewhere else in a sentence: in "er ladet sie ein", *ladet* and *ein* come from the lexeme

tedalegnie_+CDBn+PP[num=sg gender=masc case=nom deg=pos]

Figure 3.11: Coded infixes in data for a morphological guesser

einladen. The data for the guesser can take that into account in a similar way to the handling of prefixes. The question is whether this should be done in the dictionary for the guesser. There are many prefixes that behave in that way, and they could be listed in many guesses, making the output less clear. It seems that the problem should rather be solved at the syntactic level.

3.4 Perfect Hashing

Hashing maps words from a given set of n words into numbers (addresses). If the mapping is a bijection into a set $1 \dots n$ (or $0 \dots n-1$), the hashing is said to be "perfect".

It is possible to number the words of the language accepted by an automaton. The numbers associated with the words depend on the order in which they are stored in the automaton. The order is (roughly) lexicographical, i.e. to determine which of two words should be first, one compares the codes of their first letters. The smaller code means the corresponding word is stored first. If the codes are equal, the comparison is repeated with the words without their first letters, and so on, recursively. For example, in the automaton that stores the French regular endings of the verbs of the first group (fig. 4.3, page 31), if the words are numbered from 1, then *aient* receives the number 1, *ais* – 2, *ait* – 3, *ant* – 4, *assent* – 5, and so on. As the numbers result from the way the words are stored, all that is needed as input data for the algorithm that builds a perfect hashing automaton is a simple list of words. To match the order of words in the automaton, that list should be lexicographically sorted. For more information on perfect hashing, see section 6.4, page 77.

What we mean by "words" depends on the application. The words can be e.g. a list of entries in a dictionary holding the description of meanings of words, or a list of entries in an encyclopedia. In the first example, we can use a list of all inflected forms instead of the entries, which would provide a possibility of looking up the inflected forms as well. However, we are not limited to entire words being words in the natural language we process. The words may contain annotations similar to those described in the subsection 3.2.2. If we put categories as annotations in our dictionary example, we can provide full morphological analysis of inflected forms!

Chapter 4

Construction Algorithms

4.1 Taxonomy

The taxonomy presented in fig. 4.1 is from [Wat93a], with an additional branch for deterministic acyclic automata. Bruce Watson prepares a new, extended taxonomy, containing more algorithms than those shown here.

The figure shows two families of construction algorithms. Σ-algebras are a generalization of regular expressions. The algorithms constructing finite-state automata using regular expression date from the early days of computer science. The same results can be obtained in a variety of ways, as the diagram shows. Another family of methods is based on the Myhill-Nerode theorem (or at least the methods that belong to it can be derived from that theorem). Incremental algorithms for acyclic automata are added as a separate branch in the Myhill-Nerode tree. In the diagram, the algorithm for data in arbitrary order is derived from the algorithm for sorted data. This is in line with the development of those algorithms by the author. Note that the passage through the sorted data algorithm is not necessary to arrive at the version for data in arbitrary order; actually, Richard Watson and Bruce Watson developed the same algorithm without considering the version for sorted data.

4.2 Incremental Construction

A trie is a deterministic acyclic finite-state automaton (DAFSA) whose transition graph is a trie with the initial state as the root and all leaves being final. Let us imagine an FSA in a form of a trie (figure 4.2, for example). We can see that many subtrees are isomorphic. The minimal DAFSA (fig. 4.3) is the one where,

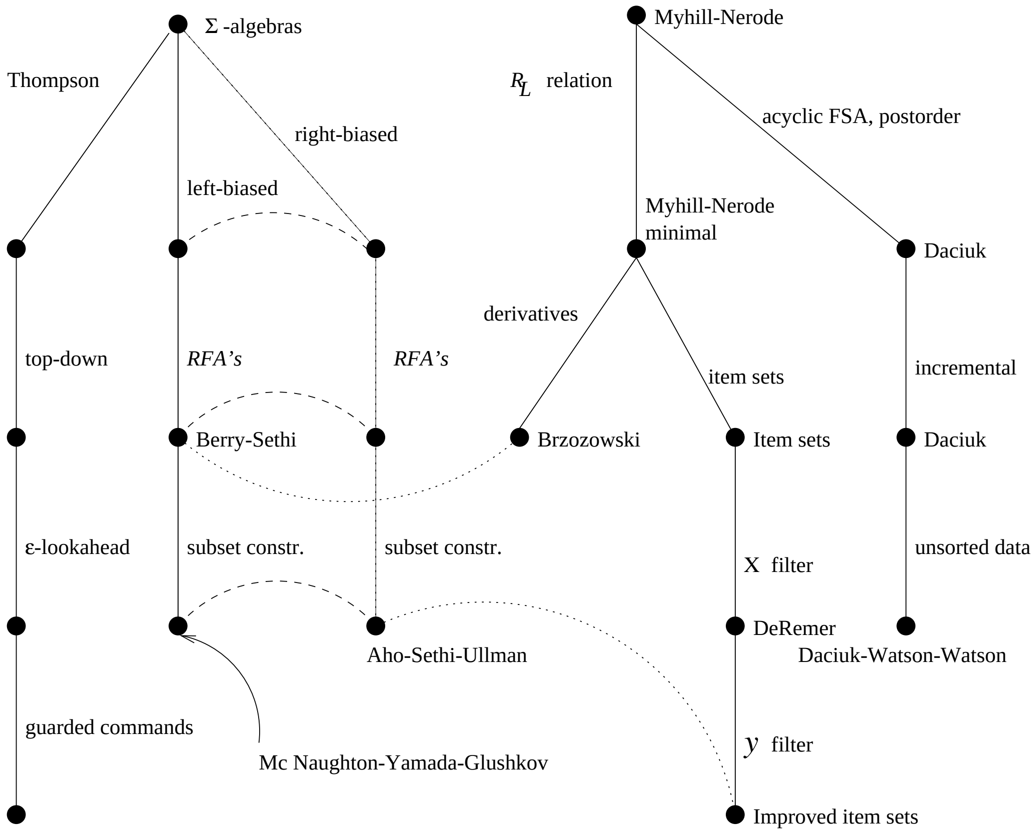


Figure 4.1: The family trees of finite automata constructions taken from [Wat93a] with modifications. Solid edges denote refinements of the solution (and therefore explicit relationships between constructions). They are labeled with the name of the refinement

for all isomorphic trees, only one copy is kept, (pointers to all pairwise isomorphic subtrees are replaced by pointers to their unique copy).

Traditionally, to obtain a minimal DAFSA one would first create a DAFSA for the language (not necessarily minimal), and then minimize it using any of a number of well known algorithms (see [Wat93b] and [Wat95]). Usually the first stage is done by building a trie. Algorithms that minimize DAFSA can be fairly effective in their use of memory, but the size of the original automaton can be huge, although some effort towards decreasing memory requirements for it has been reported ([Rev91]). We present a way to reduce these intermediate memory requirements and total construction time by constructing the minimal automaton

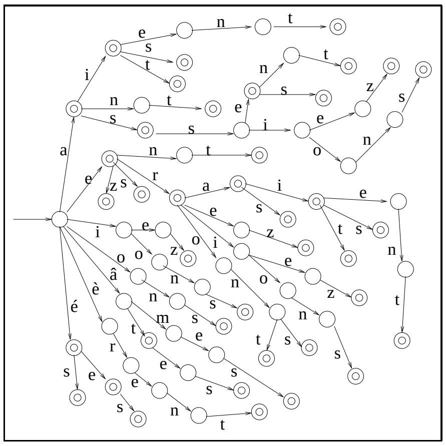


Figure 4.2: A trie whose language is the French regular endings of verbs of the first group

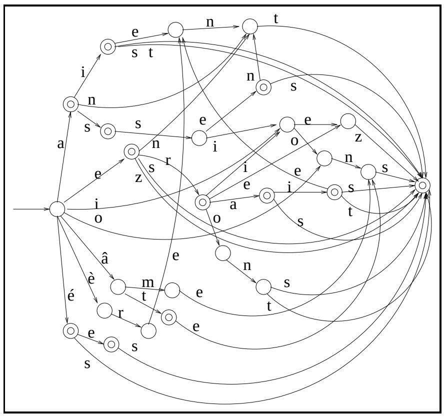


Figure 4.3: The unique minimal DAFSA whose language is the French regular endings of verbs of the first group

incrementally (word by word, maintaining an invariant of minimality) as soon as new data arrives, thus avoiding ever having a trie in memory.

The central part of all FSA minimization algorithms is a classification of states ([Wat93b], [Wat95]). The states of a DAFSA are partitioned into classes of abstractions whose representatives are the states of the minimal DAFSA. Assuming the original dictionary does not have any useless states ($Useful_s(M)$ is $true$), then we can see (by the definition of minimality we use) that each state in the minimal DAFSA must have a unique right language. Since this is a necessary and sufficient condition for minimality, we can use equality of right languages as our equivalence relation for our classes ([Wat93b], [Wat95]). Using the given definition of right languages, it is easily shown that equality of right languages is an equivalence relation (reflexive, symmetric and transitive). We will denote two states, p and q, belonging to the same equivalence class by $p \equiv q$.

Let us walk through a minimization of the trie in figure 4.2 using the algorithm given in [HU79]. Initially, only pairs of states where one is final, and the other is not, can be recognized and marked as belonging to different classes. Pairs of states that have a different number of out transitions or the same number with different labels, can be marked as different. Then pairs of states that have transitions labeled with the same symbols, but leading to states that have already been considered and are marked as different, can immediately be marked as different.

4.2.1 From Sorted Data

Let us traverse the trie (see fig 4.2) with the postorder method and see how the partition can be performed. We start with the (lexicographically) first leaf. All states up to the first forward-branching state (state with more than one outgoing transition) must belong to different classes. We can put them into a register of states so that we can find them easily. There will be no need to replace them by other states. Looking at the next branches, and starting from their leaves, we first need to know whether a given state belongs to the same class or not. Newly considered states belong to the same class as a representative of an established class if and only if:

1. they are either both final, or both non-final
2. they have the same number of outgoing transitions,
3. corresponding transitions have the same labels,
4. corresponding transitions lead to the same states, and
5. states reachable via outgoing transitions are the sole representatives of their classes.

The last condition is satisfied by the postorder method used to traverse the trie. If all the conditions are satisfied, the state is replaced by another state found in the register. If not, then it must be a representative of a new class, so it must be put into the register.

This procedure leads to a minimal automaton. Note that all leaves belong to the same class. If a pair of states have equivalent transitions (the same number of transitions, with the same labels, and the same target states), and if the target states are the sole representatives of their classes, then the right language of each pair of target states is the same. Hence our intuition about equivalence of states is supported by our alternate definition of minimal automata.

All leaves have the same right language, $\{\epsilon\}$, which is the empty string. If any pair of states have the same right language, then that language can be represented as

$$\begin{aligned}\vec{\mathcal{L}}(q) &= \{a\,\vec{\mathcal{L}}(q')|\ a \in \Sigma \wedge \exists_{q' \in Q}[\delta(q,a) = q']\} \cup \{\epsilon\},\ q \in F \\ &= \{a\,\vec{\mathcal{L}}(q')|\ a \in \Sigma \wedge \exists_{q' \in Q}[\delta(q,a) = q']\},\ q \notin F\end{aligned}$$

Note that this is simply the inductive definition of the right language of a state (using δ instead of δ^*). This confirms that the states must have an equal number of transitions that have the same labels, and lead to states that have the same right languages. They also must be both either final, or non-final (the difference between final and non-final states is that the final ones have ϵ in their right language).

Now we want to add new words to our automaton gradually while maintaining the minimality property. We need to know how to synchronize two processes: one that adds new words to the automaton, the other that minimizes it. There are two crucial questions to be answered. Firstly, which states are subject to change when new words are added? Secondly,is there a way to add new words to the automaton such that we minimize the number of states that may need to be changed during the addition of a word? Looking at the figures 4.2 and 4.3, it becomes clear that in order to reproduce the same order of traversal of states, the data must be lexicographically sorted. Note that in order to this, Σ must be ordered. Farther investigation reveals that when we add words in this order, only the states that need to be traversed to accept the last word added to the automaton may change when words are added. All the rest of the automaton remains unchanged. This happens because states are changed by removing them or by appending new transitions. We call a *common prefix* the longest prefix of the new word that is also a prefix of a word already in the automaton. New transitions can only be added to the last state that belongs to the common prefix of new word added to the automaton. Since the words are lexicographically sorted, the common prefix can contain only states that represent the beginning of the last word added to the automaton (the whole word if the last word is a prefix of the new word). Each new word can either reuse the states from the common prefix, or it can add new states. The states may be removed only from the path that represents the last word added to the automaton (the part that does not contain the common prefix). When a new word is added, and the last word is not a prefix of the new word, then states in that part are subject to either removal (they are replaced by other identical states) or they are registered as new states. That discovery leads us to the algorithm given in fig 4.4.

The function *common_prefix* finds the longest prefix (of the word to be added) that is a prefix of a word already in the automaton.

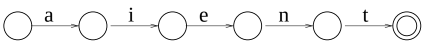
The function *add_suffix* creates a branch extending out of the automaton that represents the suffix of the word being added (the portion of the word which is left after the common prefix has been cut). The last state of this branch is marked as final. The function *last_child* returns the state reached by the (lexicographically) last transition which is going out from the argument state. Since the data is lexicographically sorted, *last_child* returns the state reached by the transition added most recently from the state (during the addition of the previous word). Each state has a marker that indicates whether or not it is already registered, because it is necessary to determine which states have already been processed. Some parts of the automaton are left for further treatment (replacement or reg-

```
Register ← ∅
while there is a next word
    Word ← next word
    CommonPrefix ← common_prefix(Word)
    LastState ← δ*(q0, CommonPrefix)
    CurrentSuffix ← Word_{length(CommonPrefix)+1...length(Word)}
    if has_children(LastState)
        replace_or_register(LastState)
    add_sufix(LastState, CurrentSuffix)
replace_or_register(q0)

function common_prefix(Word)
    return Word_{1...n} : n = max i : ∃_{q∈Q} δ(q0, Word_{1...i}) = q

function replace_or_register(State)
    Child ← last_child(State)
    if not marked_as_registered(Child)
        if has_children(Child)
            replace_or_register(Child)
        if ∃_{q∈Register} q ≡ Child
            delete_branch(Child)
            last_child(State) ← q
        else
            Register ← Register ∪{Child}
            mark_as_registered(Child)
```

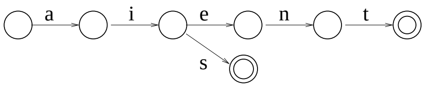
Figure 4.4: Incremental construction algorithm for sorted data

istering) until some other word is added so that those states no longer belong to the path in the automaton that accepts the new word.. That marker is read with *marked_as_registered*, and set with *mark_as_registered*. Finally, *has_children* returns *true* if there are outgoing transitions from the state, and *delete_branch* deletes its argument state and all states that can be reached from it (if they are not already marked as registered).

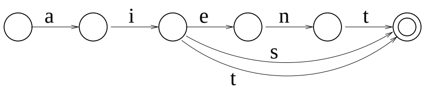
Memory requirements for this algorithm are at the maximum towards the end of processing. Memory is needed for the minimized automaton that is under construction, the call stack, and for the register of states. The memory for the automaton is proportional to the number of states and the total number of transitions. The memory for the register of states is proportional to the number of states, and can be freed once the construction is complete.



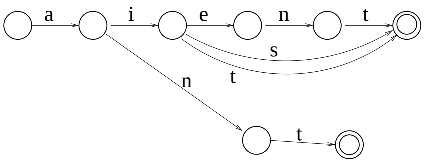
(a) Automaton with *aient*



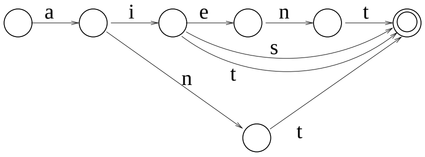
(b) The common prefix for *ais* found, the rest appended



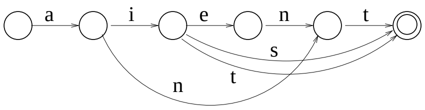
(c) The last (final) state in the path representing *ais* replaced with equivalent state already in the automaton; the same done with *ait*



(d) The common prefix for *ant* in the automaton found, the rest appended



(e) The last state in the path representing *ant* replaced with an equivalent state already in the automaton



(f) The penultimate state in the path representing *ant* replaced with an equivalent state already in the automaton

Figure 4.5: Examples of construction process for sorted data

The main loop of the algorithm runs m times, where m is the number of words to be accepted by the automaton. The function common_prefix executes in $\mathcal{O}(|w|)$

time, where $|w|$ is the maximum word length. The function replace_or_register executes recursively at most $|w|$ for each word. In each recursive call, there is one register search, and there may be one register insertion. The pessimistic time complexity of the search is $\mathcal{O}(\log n)$, where n is the number of states in the (minimized) automaton. The pessimistic time complexity of adding a state to the register is also $\mathcal{O}(\log n)$. By using a hash table to represent the register, the average time complexity of those operations can be made constant. Since all children of a state are either replaced or registered, *delete_branch* executes in constant time. So the pessimistic time complexity of the entire algorithm is $\mathcal{O}(m|w|\log n)$, while an average time complexity of $\mathcal{O}(m|w|)$ can be achieved.

Figure 4.5 shows examples of the construction process for data: *aient, ais, ait* and *ant*.

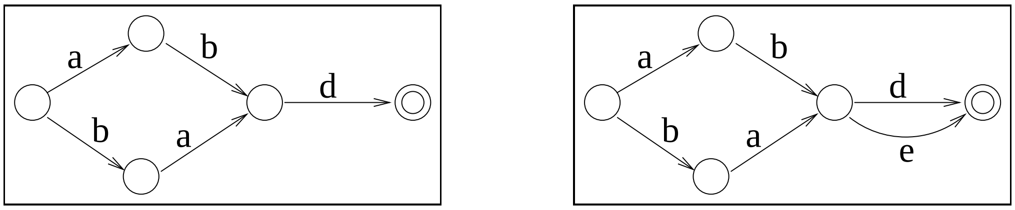
4.2.2 From Data in Arbitrary Order

Sometimes it is difficult or impossible to sort the input data before construction of an automaton (for example, there is not enough disk space to store and sort data or the data comes from another program or physical source. An incremental automaton building algorithm may still be very useful in those situations, although unsorted data makes it more difficult to merge the trie building process, and the minimization process. There are two extreme solutions:

- do not minimize what may be changed later. That is, do not immediately determine the equivalence class of a new state, but do so at the end of the algorithm. This leads to the traditional method of building a trie, and minimizing it afterwards,

- minimize everything on-the-fly, possibly changing a states equivalence class each time a word is added. Before actually constructing a new state in the automaton, we first determine if it would be in the equivalence class of a pre-existing state. In addition, we might need to change the equivalence classes of previously constructed states since their right languages might have changed. This leads to an incremental construction algorithm.

The description of the second case is included here. Every time a word is added to the automaton, it is minimized. This could lead to modifications of the parts of the automaton that had previously been minimized. Similar to the algorithm for sorted data, when a new word is added, the common prefix is searched for in the automaton. However, one cannot assume that the path containing the prefix will not be changed. If there are states in the path traversed by the word in the pre-existing automaton that are targets of more than one incoming transition (known



(a) Automaton containing words: *abd* and *bad*

(b) Word *bae* added to an automaton with *abd* and *bad*

Figure 4.6: The result of adding the word *bae* to a minimized automaton containing *abd* and *bad*. The automaton inadvertently contains *abe* as well.

as *confluence states*, then appending another transition to the last state in the path would accidentally add more words than desired (see fig. 4.6).

To avoid generation of such spurious words, all states in the common prefix from the first state that has more than one incoming transition must be cloned. *Cloning* is the process of creating a new state which has outgoing transitions with the same labels and to the same destination states as a given state. If we compare the minimal automaton to an equivalent trie, we note that a confluence state can be seen as a root of several original isomorphic subtrees merged into one (as described in the previous section). One of the isomorphisms needs to be modified, so it must be separated from the others by cloning its root. The rest of the word must be appended. The new subtree is then modified, so its states must be removed from the register, and either re-registered, or replaced by another (equivalent) state already in the register. If there are no confluent states in the common prefix (up to a confluence state), then adding the rest of the word does not differ from the algorithm for sorted data.

When the process of traversing the common prefix (up to a confluence state) and adding the suffix is complete, the modifications are not over. The previous states in the prefix path may need to be changed, because the right languages of those states may have changed. We must recalculate the equivalence relation for all states on the path of the new word. The process could actually make the new minimal automaton smaller. The resulting algorithm is shown in fig. 4.7.

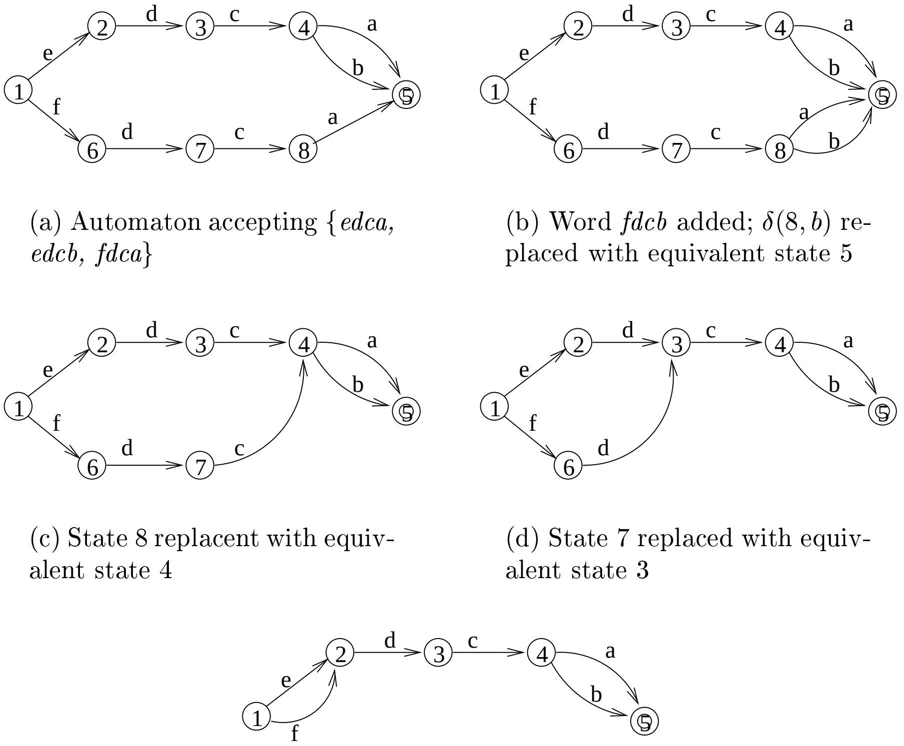
The *replace_or_register* procedure from the previous algorithm needs to be modified slightly. As new words are added in arbitrary order, and one can no longer assume that the last child is the one that has been added or modified last. Now, all children of a state must be checked, not only the most recently altered

Register $\leftarrow \emptyset$
while there is a next word
    Word $\leftarrow$ next word
    CommonPrefix $\leftarrow$ common_prefix(Word)
    FirstState $\leftarrow$ first confluence state
    if FirstState $= \emptyset$
        LastState $\leftarrow \delta(q_0,$ CommonPrefix$)$
    else
        LastState $\leftarrow clone(\delta(q_0,$ CommonPrefix$))$
    CurrentSuffix $\leftarrow Word_{length(CommonPrefix)+1...length(Word)}$
    addpostfix(LastState, CurrentSuffix)
    if FirstState $\neq \emptyset$
        CurrentIndex $\leftarrow$ (length(x): $\delta(q_0, x) = FirstState$)
        for i **from** length(CommonPrefix) - 1 **downto** CurrentIndex
            CurrentState $= clone(\delta(q_0, CommonPrefix_{1...i}))$
            $\delta(CurrentState, CommonPrefix_i) \leftarrow LastState$
            replace_or_register(CurrentState)
            LastState $\leftarrow$ CurrentState
    else
        CurrentIndex $\leftarrow$ length(CommonPrefix)
    repeat
        CurrentIndex $\leftarrow$ CurrentIndex - 1
        CurrentState $\leftarrow \delta(q_0, Word_{1...CurrentIndex})$
        OldState $\leftarrow$ LastState
        mark_as_not_registered(LastState)
        Register $\leftarrow$ Register $\setminus \{LastState\}$
        replace_or_register(CurrentState)
        nochange $\leftarrow$ OldState $=$ LastState
    until nochange

Figure 4.7: Incremental construction algorithm from data in arbitrary order

child. However, only one child may need treatment, so the execution time is of the same order. In the previous algorithm, *add_suffix* is never passed ϵ as an argument. In this version, it may happen. The effect is that the current state should be marked as final since the common prefix is already the entire word.

As before, the main loop executes m times, where m is the number of words accepted by the automaton. The inner loops are executed at most $|w|$ times for each word. Putting a state into the register is $\mathcal{O}(\log n)$, and it may be constant for regular data when using a hash table. The same estimation is valid for a removal from the register. So the time complexity of the algorithm remains the same, but the constant changes. For sorted data, only a small part of the automaton is changed each time a new word is added. For the data in arbitrary order, the changes frequently percolate all the way back to the start state, so processing each word costs more.



(a) Automaton accepting {*edca*, *edcb*, *fdca*}

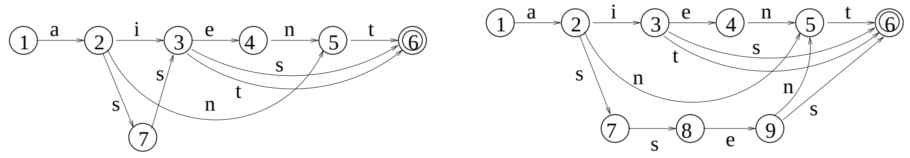
(b) Word *fdcb* added; $\delta(8, b)$ replaced with equivalent state 5

(c) State 8 replacent with equivalent state 4

(d) State 7 replaced with equivalent state 3

(e) Final result

Figure 4.8: Examples of Construction process from unsorted data: changes propagate to the root



(a) Automaton with *aient, ais, ait, ant*, and *assent*

(b) After *asses* has been added, the states 3 and 4 had to be cloned

Figure 4.9: Examples of construction process from unsorted data: cloning

4.3 Algorithm by Dominique Revuz

We know of one algorithm that constructs minimal deterministic acyclic finite-state automata incrementally. Dominique Revuz ([Rev91]) does it in two stages: the first one is called a pseudominimization, the second one – the proper minimization. We give here an account of that algorithm based on [Rev91].

A notion of *the longest common suffix* is introduced. The longest common suffix of a word and a set is the longest common suffix of the word and a suffix of a word of the set.

To simplify the search of the longest common suffix (written $lsc(w, X)$ from French *"le plus long suffixe commun"*), the words are appended in inversed lexicograhical order, in which the comparison between the letters of words is done from right to left. This order makes it possible to enter words one after another so that the longest common prefix is the longest common prefix of the last word with the set of words already entered.

While there is a transition labeled with the current letter (of the word) and the state reached with that transition has only one predecessor, one moves to that state using the transition.

While the transition exists but the target state has more than one predecessor, the target state is duplicated, and the transition is redirected towards the copy. One moves to that copy.

While the longest common suffix is not reached, one creates a new transition labeled with the current letter and a new state, to which one moves using the transition.

Now the longest common suffix is reached.

While the state of the longest common suffix belongs to the path in the automaton used by the prefix of the word, one creates a new transition labeled with the current letter and a new state, to which one moves using the transition.

If the state in the longest common prefix does does not belong to the state used by the prefix of the word, one creates a transition that leads to that state and one moves to that state.

Let d be the last state reached in one of the last two loops. If d is final, then:

- If d has more than one predecessor, one clones d to get d', and d' becomes final.
- If d has only one predecessor, it becomes final.

That was the pseudominimization phase. It is followed by a true minimization phase, which consists in partitioning the states into classes that have the same maximal length of strings in their right languages. The states are sorted in their classes, starting from the class that have the maximal length of string in its right language equal to 0, and proceeding with 1, 2, etc. During the sorting, the equivalent states are eliminated.

4.4 Representation

The size of the automaton, and the speed of its use, depend on the method used for its representation. The differences can be drastic. It should be noted that there is no single representation method suitable for all applications and providing the best characteristics in all aspects. The choice must be seen as a compromise between various requirements. The representations of automata are the same as those for directed graphs (see e.g. [AHU74]).

Two classical methods are the *neighborhood matrix* and *lists*. The neighborhood matrix is a matrix where one axis is labeled with the source state numbers of the transitions, and the other one is labeled with the target state numbers of the transitions. State attributes must be stored elsewhere, i.e. in another vector. The entries in the matrix are the attributes of the transitions, here: labels. Table 4.1 shows such a structure for the example automaton from fig. 2.3 (page 13).

As large automata may have millions of states, and there may be many different transitions between states, such method cannot be effective. A variant of the neighborhood matrix uses all possible symbols as the columns; the target states are values of the matrix. Such matrix is much better than the previous one, both in terms of memory requirements and processing time.

Another traditional approach is to use coincidence lists. Fig. 4.10 shows the example automaton from fig.2.3 (page 13) in form of coincidence lists.

Note that if the transitions can be stored in consecutive memory locations, then there is no need for links to the next transitions. What is needed is the number

	1	2	3	4
1	0	b	a	0
2	0	0	0	c
3	0	0	0	ce
4	0	0	0	0

Table 4.1: Neighborhood matrix for example automaton from fig. 2.3

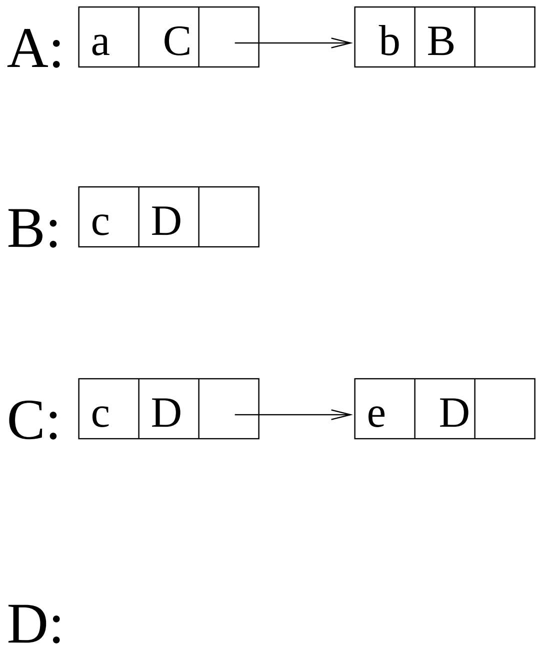


Figure 4.10: Example automaton from fig. 2.3 represented by lists

of transitions for the state. Transitions can be stored as consecutive entries in a vector. If we move the properties: the number of transitions, and the final status of the states to the transitions that lead to them, we no longer need to store the states explicitly; they will be stored implicitly in the transitions. See table 4.2 for an example. Interestingly, moving some state properties to transitions resulted in the creation of a new state. That state exists solely because its dummy transition provides the properties of the initial state.

Since the states in the method shown above are implicit, further compression is possible. Two such methods were explored in the implementation of the automata

transition	label	transitions	final	target	source state
1	0	2	yes	2	(start)
2	a	2	yes	5	(A)
3	b	1	no	4	(A)
4	c	0	yes	0	(B)
5	c	0	yes	0	(C)
6	e	0	yes	0	(C)

Table 4.2: Example automaton from fig.2.3 represented by vectors of transitions

construction algorithms presented in this dissertation:

- storing a smaller state inside a bigger one (the transitions of the smaller state are a subset of the transitions of the bigger one),
- storing a state so that it shares some consecutive transitions of two (or more) states stored in consecutive locations (the state overlaps with two or more other states – all its transitions are represented as transitions of those states).

Note that for the first method to work efficiently, the transitions should be sorted. The order imposed by the sorting affects compression. The best results can be achieved by sorting the transitions on the frequency of occurrences of their labels[1]. Finding the ideal layout of the transitions in each state is NP-hard. Table 4.3 shows the example automaton where one of states (B) was put into another state (C). Assuming that all transitions of a state are ordered, we denote as $\hat{\delta}_i(q)$ the ith transition of the state q. For a given state q_1, the method consists in finding a representation of a state $\hat{\delta}(q_2) = \{\hat{\delta}_1(q_2) \ldots \hat{\delta}_{|\hat{\delta}(q_2)|}\}$ such that

$$|\hat{\delta}(q_2)| > |\hat{\delta}(q_1)| \wedge \exists_{i:i\geq 0 \wedge i+|\hat{\delta}(q_1)|-1\leq|\hat{\delta}(q_2)|} : \forall_{j:j\geq i \wedge j\leq i+|\hat{\delta}(q_1)|} \hat{\delta}_j(q_1) = \hat{\delta}_{j+i}(q_2)$$

The other method can be applied after the first one described above. We take a state represented as a set of its outgoing transitions and look for two other states. A few of the last transitions of first one should be the same as the first transitions of the state we want to compress, and the remaining transitions of the state to be compressed should be the same as the first transitions of the second state we search for. Formally, for a given state q_1, we look for states q_2 and q_3 such that $|\hat{\delta}(q_1)| \geq |\hat{\delta}(q_2)|$ and $|\hat{\delta}(q_1)| \geq |\hat{\delta}(q_3)|$, and $|\hat{\delta}(q_1)| \leq |\hat{\delta}(q_2)| + |\hat{\delta}(q_3)|$, and:

[1]Personal communication with Martin Kay

transition	label	transitions	final	target	source state
1	0	2	yes	2	(start)
2	a	2	yes	4	(A)
3	b	1	no	4	(A)
4	c	0	yes	0	(B,C)
5	e	0	yes	0	(C)

Table 4.3: Example automaton from fig. 2.3 represented by vectors of transitions - state B resides inside state C

$$\exists_{i:i\geq 1\wedge j\leq min(|\hat{\delta}(q_1)|,|\hat{\delta}(q_2)|}\forall_{j:j\geq i\wedge j\leq|\hat{\delta}(q_2)|}\hat{\delta}_j(q_1)=\hat{\delta}_{j-i+1}(q_2)$$

and

$$\forall_{j:j\geq 1\wedge j<|\hat{\delta}(q_1)|}\hat{\delta}_{j+|\hat{\delta}(q_2)|-i+1}(q_1)=\hat{\delta}_j(q_3)$$

In our programs all states that were allowed to take part in such match-making had two transitions.

Tomasz Kowaltowski ([KLS93b]) uses a form of coincidence lists to obtain the best compression of automata. A state of a traditional finite-state automaton is treated as a list of states of a *binary automaton*. Those states (we will call them *binary states* here to avoid confusion with the states of a traditional FSA) correspond to the transitions of the traditional FSA. Each state of the binary automaton has two pointers. The first one leads to the target state of the transition. The second one points to the next transition in the current state of the original FSA. Tomasz Kowaltowski notes that, with such representation of states, it is possible to merge the endings of two lists, i.e. to merge parts of states. Note that in the approach described above, a state can be merged either with two adjacent states, or with only one bigger state that contains all transitions of the smaller state (that bigger state may further be merged with an even bigger state). In Kowaltowski's approach, one smaller state may be merged with several bigger states that cannot be merged one into another.

A *proper state* is a binary state that is either:

- the initial state of the automaton, or
- the target of transition, or
- the target of at least two pointers that chain the binary states of an FSA state (*failure transitions* in Kowaltowski's terminology)

Proper states are visited in order so that a proper state cannot be visited before all proper states that are targets of the transitions of that state are visited. When a proper state is visited, one looks among all states already visited for a state that contains a set of binary states being a subset of the set of transitions of the current state. From the set of states found in that search the state with the biggest number of binary states is chosen. The binary states in the current proper state are reordered so that those that are present in the chosen state are in the end. Then a pointer to them is replaced with a pointer to the chosen proper state.

To maximize the probability that the current state will be matched by parts of other states, its binary states that are not shared with chosen proper state are ordered by the frequency of their occurrence in the proper states that were not visited yet, and contain the binary states of the chosen state. The failure transition of the least frequent binary state in a proper state points to a more frequent binary state, and so on to the most frequent binary state whose failure transition points to the chosen state.

The automaton is represented as a vector of binary states. Judicious coding makes it possible to obtain substantial compression. The symbols (the labels of transitions) are coded on $\lceil \log_2(|\Sigma| + 1) \rceil$ bits. The number of symbols in the language is incremented by one to accommodate for a special symbol that marks the end of the word. Tomasz Kowaltowski notes that most states of an FSA have only one transition, and thus the target of that transition is the previous state (states are numbered from the sink state up to the start state). An extra bit distinguishes between the situation when the target is the previous binary state, or not. In the latter case the target state number i is coded on k bits, where $1 + 2^{k-1} < i \leq 1 + 2^k, i \geq 3$. There are no final states, i.e. no states need to bare the marker of being final, as the end of the word is marked by a special symbol. This means that every final state has a transition labeled with that symbol. In that case, there is no need to code the target state, as it is the same for all transitions labeled with the symbol.

In the case of the failure transition, most target states are either the previous binary state (states are numbered from the end), or none. The third possibility is an arbitrary state (coded as shown above), so it is possible to use one or two bits to distinguish between these 3 situations. Tomasz Kowaltowski reports compression rates (the size of the compressed folded automaton divided by the size of the original automaton) up to 99%.

Dominique Revuz ([Rev91]), and Emmanuel Roche and Yves Schabes ([RS95]) argue for a different representation method. The automaton is stored in a matrix where each row is a state, and each column is labeled with one symbol. There are $|\Sigma|$ columns. If a transition from a given state labeled with a given symbol exists, the entry with the coordinates (state, symbol) contains that symbol and the number of the target state. Otherwise it contains something else.

	a	b	c	d	e
1	a,3	b,2			
2			c,4		
3			c,4		e,4
4					

Table 4.4: Matrix labeled with states and symbols storing the example automaton from fig. 2.3

As the example shows, such a table is sparse, so they recommend using a method described in [TY79]. The method (fig. 4.11) compresses the matrix by trying to arrange its rows so that they overlap. The matrix is compressed to a vector. For each row, an attempt is made to store it in the vector in such a way that the transitions fall into empty spaces left from other rows. The empty spaces appear when the state in question does not have $|\Sigma|$ transitions. For each symbol in Σ being the potential label, if there is no transition labeled with it, an empty space appears.

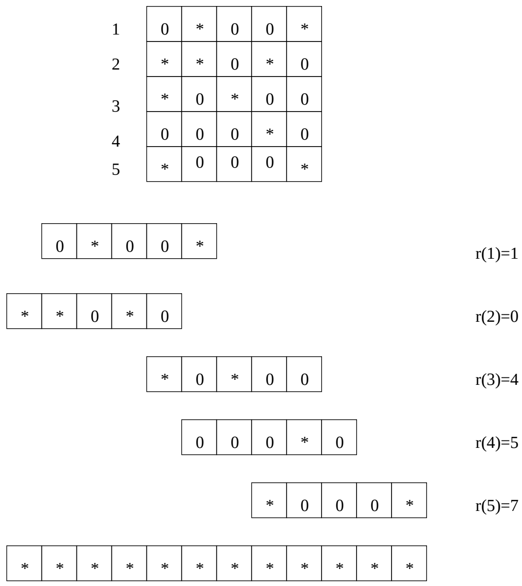


Figure 4.11: Compression of a sparse table (adopted from [TY79])

4.5 Discussion

The algorithms presented here are new and original. They have been developed during the author's stay at ISSCO, Geneva, Switzerland. At the same time, and independently, those algorithms were also developed by Bruce Watson and Richard Watson.

By exploiting particular features of deterministic acyclic FSA as opposed to finite-state automata in general, it was possible to devise a way to build automata more efficiently than it was possible in general case.

An algorithm described in [Rev91] also constructs an automaton from sorted data while performing a sort of minimization on-the-fly. Data is sorted in reverse order, and that property is used to compress the endings of words while building the automaton. This is called a pseudo-minimization, and it must be supplemented by a true minimization afterwards. Although that technique is more economic in its use of memory than standard techniques, we are still left with an automaton that is not minimal in an intermediate state. Revuz writes that for the DELAF dictionary, the automaton after the pseudo-minimization phase was 8 times bigger than the minimal one. Our algorithms need the space for the minimal automaton, and the register. The size of the register is proportional to the number of states in the automaton, and it may have index structures proportional to the logarithm of the number of states. In any case, its size is much smaller than the size of the automaton.

Compared to Revuz'es algorithm, our algorithms use less memory, because they minimize the automaton each time a new word has been added. Our algorithm for sorted data makes it possible to perform some modifications that possibly reduce its size on-the-fly (such as e.g. pruning required for building a morphological guesser) because minimized parts of the automaton are left unchanged till the end of the construction process. The algorithm for data in arbitrary order no longer has that interesting property, but it is more flexible than the one by Dominique Revuz.

Note that it is possible to use the incremental algorithms described above to build deterministic acyclic finite-state automata with final transitions. In fact, the algorithms presented in this chapter are the modified versions of the algorithms for the DAFSA-FT. Both the vectors of transitions, and the sparse table representation used by Dominique Revuz *et al.* can be used with the DAFSA-FT to obtain smaller automata. The gains depend on the characteristics of particular data. Table 4.5 provides some examples. The French words were taken from a morphological lexicon of French from ISSCO, Geneva. It contains 210284 words. The Polish words come from a lexicon developed by Prof. Zygmunt Vetulani from Adam Mickiewicz University in Poznań, Poland. It contains 1012496 words and expressions. The Polish lexicon is less regular. This may be the reason for bigger gains with that data. Note that the gains depend on the percentage of final states in the automaton. It is possible to construct such deterministic acyclic automata with n states that the corresponding DAFSA-FT has $(n-2)/2+2$ state. For large automata of this kind $|DAFSA-FT| \rightarrow n/2$. The automaton in figure 2.4 is such an automaton ($n=4$).

		French	Polish
DAFSA	States	16665	55393
	Transitions	43507	160216
DAFSA-FT	States	16494	54261
	Transitions	43163	157497
Savings	States	1%	2%
	Transitions	0.8%	1.7%

Table 4.5: DAFSA-FT compared with DAFSA dictionaries of French and Polish words

The same algorithms can be used to construct transducers. Labels from both levels can be put one after another (possibly using a filler character where there is no symbol on any level), and then treated as words for an automaton, with the alphabet being $\Sigma_1 \times \Sigma_2$, where Σ_1 and Σ_2 are the alphabets of the levels. The transducers constructed in such a way are not subsequential.

We do not use the representation advocated by Revuz and others (based on [TY79]). That representation actually produces smaller dictionaries. This is achieved not by better placement of states (and their transitions) – this is already optimal in the vector representation we use, but by eliminating the need to store the number of transitions parting from a given state. It is also faster at recognition. However, in applications where one needs to list or to try all descendants of a state, like in spelling correction, diacritic restoration, and (partly) morphological analysis, it is much slower, as one must check all $|\Sigma|$ possibilities at each state.

Chapter 5

Construction of Guessing Automata

5.1 Motivation

A morphological analysis of words in a text is needed in many applications. It constitutes a prerequisite for natural language parsing and all for applications that use it, it is also useful in document retrieval. Such analysis is usually lexicon-based, i.e. it requires a morphological lexicon.

Unfortunately, real-world texts contain correct words that cannot be found in a lexicon. It seems impossible to record all words of a living language in a lexicon, as a lexicon is static in nature, and a language is a living thing – new words are coined continually. Another reason for finding words not present in the lexicon is the Zipf's law[Bri93]. The Zipf's law states that the rank of an element divided by the frequency of occurrence is constant. E.g. in the Brown corpus, two percent of different words[1] account for sixty nine percent of the text. About seventy five percent of different words occur five or fewer times in the corpus. Fifty eight percent of different words occur two or fewer times, and fourth four percent only occur once. The consequence of the Zipf's law is that by doubling the number of words in the lexicon, one gets only a few percents of the coverage of an arbitrary unrestricted text. Therefore, increasing the size of the lexicon is a very costly effort yielding minute results.

New words are also constructed by derivation on compounding. While the analysis of such words is relatively easy, their number is very small compared to the number of potential words formed in the same way. Therefore, it is not practical

[1]Eric Brill uses the terms *word type* and *token*

to store all such derivatives and compounds in the lexicon. In many cases there may be many ways to form a new word, and it is not possible to predict which one would be chosen.

Additionally, texts may contain incorrect words. For purpose of e.g. spelling corrections, the morpho-syntactic categories of a misspelled word may help reduce the list of possible corrections. If the misspelling does not affect the word's flectional ending (and its prefix, if present), these categories may still be easily obtainable from the corrupted version.

It is possible to use textbooks to write rules that associate word endings with specific tags. Actually, it seems to be a natural way to write a category guesser. It is possible to enhance the rules so that not only categories, but lexemes as well are predicted. However, this process requires a considerable amount of linguistic knowledge. Textbook rules do not capture many exceptions that are present because of the nature of human languages, so the rules must be refined over and over again.

Our aim is to

- use an already existing morphological lexicon;
- reduce the size of the lexicon by cutting out useless information (beginnings of words);
- generalize the knowledge contained in the lexicon so that accurate prediction of morphological information for unknown words be possible.

5.2 Method Description

Our method consists of preparing data for the guesser, building a finite state automaton from that data, cutting out useless information, and using the automaton to recognize meaningful word endings. In the automaton, each transition is labeled with one character.

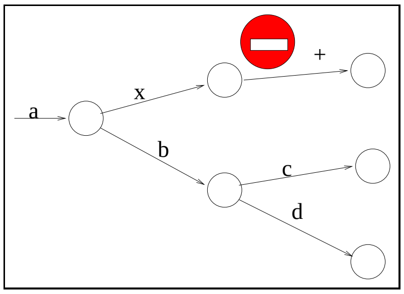
We use a particular format of data that is very important for the way the guesser is constructed. Please refer to the section 3.3 for its detailed description.

5.2.1 Lexicon Reduction

All strings constructed in the way described in section 3.3 are used to build a finite state automaton. As the strings consist of inverted inflected forms, followed by an annotation separator, and by annotations, the resulting automaton can already be used to the annotations from the corresponding inflected form. The inflected form that is to be analyzed should be inverted, and one should look for the resulting string in the automaton. If the word is not in the lexicon, the search fails at a certain state, i.e. no transition labeled with the next letter of the string can be found. In such case, all annotations that can be reached from that state should be printed. Although the automaton could be used for our purposes, it is unnecessarily big, as it contains much useless information.

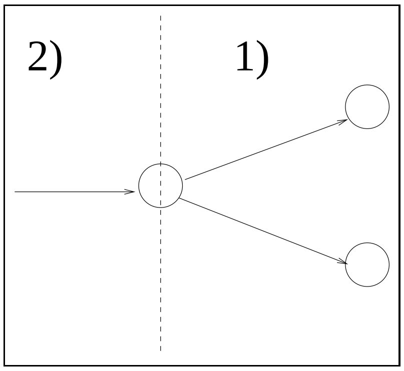
As word endings normally decide what annotation should be associated with a given word, the automaton has a particular structure. For a given word, the first few states have many outgoing transitions. Then, there is a chain of states linked with each other with single transitions. Passing to annotations, a more complicated transition network appearers again. As for any state in the central part, there is only one way leading from the state to the appropriate annotations, it represents no useful information for our purpose. So all states from that part can be pruned, along with the corresponding transitions.

Pruning is governed by the following rules:



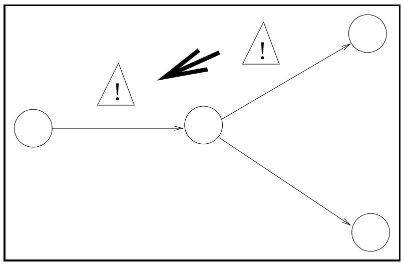
R1. The pruning process does not apply to transitions belonging to annotations (the annotations are separated from the inflected forms by the annotation separator – represented here by "+").

This rule should be obvious, because annotations are what we want to obtain in the recognition phase. The transitions that are pruned belong to the inflected forms, and more precisely: to their beginnings that do not influence the annotations.



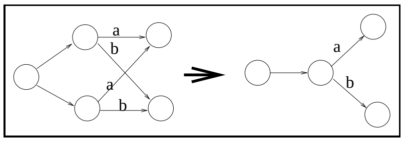
R2. A transition can be removed only if the pruning process has already visited all transitions that can be reached through the target state, except the transitions representing annotations. In other words, the states are visited (and the outgoing transitions pruned if possible) in the post-order method.

This means that we traverse the automaton recursively in depth, cutting unneeded transitions on the way back.



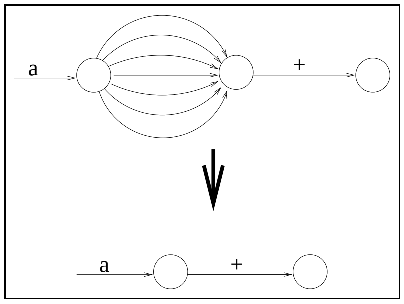
R3. A transition cannot be removed if the target state has a transition that does not belong to annotations, but cannot be removed. The warning sign above a transition in the figure shows that the transitions cannot be removed. We use a different sign than in R1, as the reason the transition cannot be removed is different.

This means that the target state has transitions that distinguish between different annotations, i.e. they lead to different sets of annotations. We do not want to lose that distinction.



R4. A state can always be replaced with an equivalent one (i.e. a state with the same transitions leading to the same states).

The automaton should be kept minimal.



R5. A state with all transitions leading to one state can be removed with all transitions, and transitions that point to it should be replaced by transitions pointing to the target state.

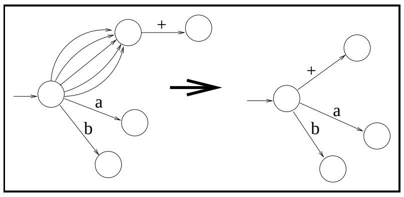
This is the rule that actually cuts transitions. Note that it also applies to states with one only transition.

The consequence of the rules R2 and R5 is that we cannot remove states (and transitions that lead to them) if they have transitions leading to different sets of annotations. This is because the rule R2 ensures that all transitions of a visited state lead to states that either cannot be removed, or that have an outgoing transition labeled with the annotation separator.

5.2.2 Default Annotations

The process described above leads to the construction of a finite state automaton that contains all required information. However, the automaton is still big, and an effort can be made to further reduce its size. Looking at its contents, we can see many states with a majority of transitions leading to one state (we will use the term *default state*), but with other transitions as well. The target state must have one outgoing transition, and it must be labeled with the annotation separator. We can treat less frequent transitions as exceptions, assuming that *all* other transitions, even those that had not appeared in our lexicon, lead to the default state. Acting under this assumption, we can replace the frequent transitions and the default state with the transition leading from the default state. A limit can be imposed on the ratio of frequent/less frequent transitions to trigger the pruning.

There is a difference between rules R1, R2, ... R5, and this rule. Rule R6 introduces a generalization. While still 100% of words present in the lexicon are annotated correctly, R6 may select one annotation among many as the correct one, and hide other possibilities. This may speed up an annotation process, but it can also introduce errors: some correct (but less probable) possibilities may not be shown, as the lexicon may not contain data that associates their endings with their correct annotations.



R6. If for a given state the number of transitions leading to one state (the default state) is greater or equal to the number of all other transitions multiplied by a small integer, and the default state has only one outgoing transition and it is labeled with annotation separator, then the default state can be removed, and all transitions that lead to it should be replaced by the transition leaving the default state.

5.2.3 More Generalization

Sometimes, it is impossible to devise a rule that associates an ending with the correct annotation, because the choice is lexicalized, i.e. it depends on a particular word, and it seems arbitrary from the morphological point of view. For example, in Polish, there is a rule that transforms adjectival endings *-sny* in lexemes into *-śniejszy* in comparatives and superlatives. There is, however, another rule that transforms endings *-śny* into *-śniejszy* in comparatives and superlatives. So there is no other way of knowing what the lexeme might be from a comparative or superlative ending other than a dictionary lookup. R6 introduces artificial divisions, e.g.:

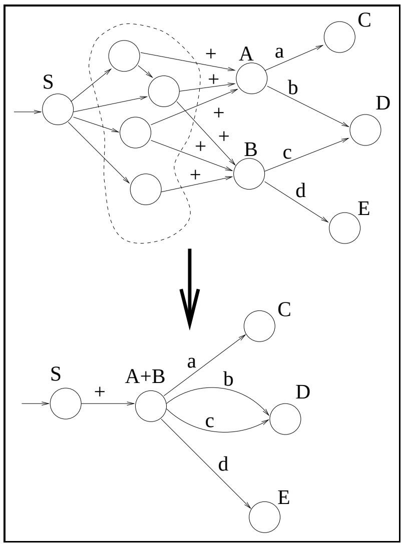
-raśniejszy → -raśny
-iaśniejszy → -iasny
-maśniejszy → -maśny
-waśniejszy → -waśny
jaśniejszy → jasny
-ośniejszy → -ośny
-dośniejszy → -dośny
-ześniejszy → -zesny
-oleśniejszy → -olesny
-bleśniejszy → -bleśny
-uśniejszy → -uśny

while the right answer is that both annotations must be considered:

-śniejszy → -śny

-śniejszy → -sny

To cope with that situation, we introduce a new rule that strives to accommodate such cases. We will use the term *first annotated state* to name a state that is a target of a transition labeled with the annotation separator (a state that begins an annotation or a set of annotations).

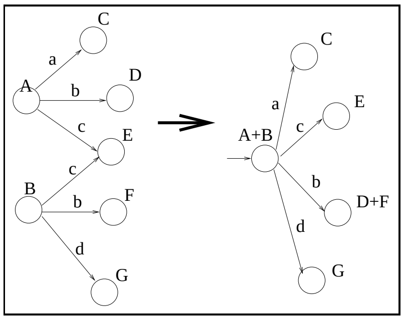


R7. If for a given state the number of first annotated states that are reachable from the given state does not exceed a given limit, then:

- replace the first annotated states by their union;
- replace all the states and transitions between the chosen state and the union of the first annotated states by a single transition labeled with the annotation separator.

Note that it is possible to introduce a lower limit on the number of states to be removed in order to insure that we are dealing with a case such that the one described above (*sny* and *śny*). The rule can then work in parallel with R6.

To make things clear, we need to describe what we mean by a union of states. We make it by constructing a new state that has all transitions from contributing states. For pairs of transitions that go to different states, we construct transitions going to unions of those states:



A union of states A and B is a state having all transitions from A and B labeled with characters present once in all transitions from A and B, all transitions form A and B that have the same labels and go to the same states, and for all transitions from A and B that have the same labels, but go to different states, transitions of the same labels going to states being a union of target states.

It is worth noting that while the rule R6 introduces very detailed distinctions, R7 discards details. For the guesser, the result of applying R7 is that one gets more choices than without having applied R6 or R7. As to the lexicon size, R7 removes small differences between similar word forms, making it possible to infer more general and compact relations between endings and annotations.

Please note that although no annotation possibility is lost, and the automaton is much smaller, the answers for known words are no longer 100% accurate. The correct answer appears always, but it may be accompanied by other, incorrect possibilities. In many cases exceptions are merged with regular rules. A lower limit imposed on the number of states to be removed by this rule can solve the problem.

5.3 Experimental Results

The algorithms were tested on French and Polish data. For Polish, prefixes that had impact on categories were removed from the automata without lexemes to make possible the comparison with French. For the rule R6, there had to be twice as much frequent transitions than the others to trigger the rule. For the rule R7, the limit on the number of first states was 2. There was no lower limit on the number of states to be removed by R7. Table 5.1 shows the automata for French, and table 5.2 – for Polish.

States with one outgoing transition constitute about half of all states, and those with one incoming transition – more than 80%. States that have only one incoming, and one outgoing transition form single lines.

The number given under the heading "total" are absolute numbers. Those under headings listing rules used are given as percents of numbers for an automaton for which no rules were applied. The standard procedure, i.e. the rules R1 ... R5 reduces the automaton to 36 to 21 percent of its original size. The effectiveness of those rules is the best for French without lexemes, and the worst for Polish with lexemes. This is because the French data is much more regular. The French dictionary was built from a two-level morphology, the Polish one – from a source listing irregular roots for all entries, and thus more prone to errors. Polish flexion is much more irregular (or rich) than the French one. And last, but not least, the Polish dictionary contained 1,757,557 inflected forms, and the French one – only 235,566. Lexemes introduce new irregularities, e.g. from irregular words.

Rules R1 ... R5 operate most effectively on states with few outgoing transitions. Adding R6 preserves that pattern, though two peaks emerge: for states with 2, and with 5 transitions. They are the result of the threshold used in the rule. There must be at least two frequent transitions for one infrequent to trigger the rule, so the states that satisfied the condition and had 3, 4, or more transitions originally, now have only 2. The second peak comes from a situation when there are are two unfrequent transitions. It is worth noticing that R6 does not introduce extra reduction of states (the reduction that does occur is the result of making some states equivalent); only transitions are cut off.

The last rule – R7 – is the most effective in terms of lexicon size reduction. While still doing better with states with fewer outgoing transitions, it cuts the number of other states much better than other methods. Both the number of states, and the number of transitions in the automaton are reduced significantly.

The size of the automaton depends on the number of different features in the tagset. The richer the tagset, the bigger the automaton. The tagset used in our experiments was verbose.

For algorithms using the guesser constructed according to the rules described in this chapter, and an evaluation of the results, see section 6.3.3.

5.4 Related Work

In the past, various handcrafted heuristics have been used for the purpose of morphological analysis of unknown words. Later, they have been supplemented by statistical techniques (e.g. [WMS+93]). However, although the probabilities of different endings leading to their corresponding categories were calculated, the endings themselves were chosen manually. A revolutionary approach was proposed by Eric Brill ([Bri93], [Bri95]). The endings, as well as prefixes, are found by the program. Unknown words are first tagged by a naive initial-state annotator that assignes the tags proper noun or common noun on the bases of their capitalization. Then five types of transformations are applied:

Change the tag of an unknown word (from X) to Y if:

1. Deleting the prefix (suffix) x, $|x| \leq 4$, results in a word (x is any string of length 1 to 4).
2. The first (last) (1,2,3,4) characters of the word are are x.
3. Adding the character string x as a prefix (suffix) results in a word ($|x| \leq 4$).
4. Word w ever appears immediately to the left (right) of the word.
5. character z appears in the word.

The result is compared with a tagged corpus. The best-scoring rule is chosen, and applied to the corpus so that it becomes new input data. The learning stops when no rule can increase the score. The transformation type 4 takes into account the context of the unknown word. When the morphological analysis is separated from a contextual tagger, as it is the case with our approach, the tagger must find those rules itself. The transformation types 1 and 3 checks whether adding or deleting characters from/to a word results in another word. In our algorithm those transformations are not present, as well as transformation type 5, which can be treated, if necessary, as a supplementary heuristics. The rules schemata presented above do not account for infixes, which can be treated with our method.

Andrei Mikheev ([Mik97]) applied Brill's transformations to the data for a (pre-existing) morphological lexicon. He uses the following template:

$$G =_{x:\{b,e\}} [-S \ + M \ ?I\text{-class} \rightarrow R\text{-class}],$$

where:

- x indicates whether the rule is applied to the beginning or end of a word and has two possible values: b – beginning, e – end;
- S is the affix to be segmented; it is deleted (-) from the beginning or end of an unknown word according to the value of x;
- M is the mutative segment (possibly empty), which should be added (+) to the result string after segmentation;

- i-class is the required POS-class (set of one or more POS-tags) of the stem; the result string after the -S and $+M$ operations should be checked (?) in the lexicon for having this particular POS-class; if I-class is set to be "void" no checking is required;
- R-class is the POS-class to assign ($\rightarrow$) to the unknown word if all the above operations (-S $+R$?I) have been successful

Compared to Brill's algorithm, Mikheev checks also (optionally) the morphological class (a set of categories) of the resulting word in transformation templates 1 and 3. Also, his algorithm returns all categories of an unknown word, not only the most probable one.

Although Mikheev is not aware of that fact[2], the rules learnt by Brill's algorithm can be transformed into a finite-state automaton. However, that process is complex and time-consuming. The learning process takes time as well. Our algorithm produces the FSA directly from data exploring the links that occur naturally in the format of data we use. In Brill's approach, copied by Mikheev, the length of suffixes and prefixes is a constant. Increasing it means much more computation. In our method, the suffixes are discovered naturally, so there is no need to limit their lengths.

5.5 Concluding Remarks

The method presented here leads to a device that can perform a morphological analysis of both known and unknown words as accurately as possible (see section 6.3.3 for an evaluation), while preserving the advantages of finite state automata (great speed and small size). We eliminate the need for two separate devices, and use less space.

We plan to experiment with other values for thresholds in rules R6 and R7. Another issue that needs farther investigation is the impact of the use of closed categories among the data for our guesser – how much they inflate the lexicon and affect the recognition process. Maybe exclusion of those categories from the lexicon would reduce its size. However, excluding those categories requires additional linguistic knowledge, so it contradicts our goals. On the other hand, the current treatment of prefixes requires that knowledge. However, we believe that existing algorithms (e.g. [DC92], [TC97]) make it possible to to process all data automatically.

We believe that our method can be used for lexical acquisition. Annotations may contain morphological descriptions (i.e. data for a morphology program). A

[2] [RS95] does not appear among references in [Mik97]

small core morphology can be used to construct the guesser, and then to process new words from corpora to acquire new lexemes. It should be noted that on average the current method gives only not more than 1.6 lexemes per analyzed word.

The programs used in our experiments are available free of charge for research purposes from `http://www.pg.gda.pl/~jandac/fsa.html`. The French data comes from ISSCO, Université de Genève and is available for research purposes from `ftp://issco-ftp.unige.ch/MULTEXT/` (file french.gz). The file *french.gz* is input data for *mmorph*, ISSCO's morphology program available from the same address. The Polish data comes from Zygmunt Vetulani from the Adam Mickiewicz University in Poznań, Poland, and is not available for research purposes.

Contents	no lexemes				with lexemes			
Rules used	total	1-5	1-5,6	1-5,7	total	1-5	1-5,6	R1-5,7
Transitions	335884	21	16	12	366264	25	19	14
States								
total	151693	14	13	8	177781	18	17	12
transitions								
outgoing 1	75610	10	9	8	98272	15	15	14
2	36652	14	21	4	39398	15	23	5
3	17029	17	8	6	17589	19	9	6
4	8802	21	13	10	8900	24	15	11
5	4759	26	17	13	4770	30	19	14
6	2833	29	12	16	2833	33	13	17
7	1673	33	16	18	1673	40	20	21
8	1184	35	18	21	1184	41	22	25
9	644	40	19	27	655	45	24	29
>9	2507	50	28	37	2507	61	33	42
incoming 1	129085	16	14	9	150587	19	18	12
2	9421	9	10	4	11629	12	14	8
3	3701	4	4	3	4705	8	12	7
4	2247	2	3	2	2587	6	6	6
5	1512	3	3	2	1851	8	10	6
6	955	2	2	3	1013	10	8	9
7	775	2	3	2	834	3	7	9
8	353	5	4	2	484	11	12	8
9	234	4	3	5	366	10	15	12
10-99	3178	3	5	4	3522	8	10	9
100-999	230	27	30	22	201	36	38	26
>999	2	700	450	300	2	125	400	200
in sg lines								
total	57028	11	11	9	75011	17	17	16
length 1	11821	6	5	3	14510	8	10	10
2	4163	5	4	2	5985	19	14	16
3	1246	6	6	4	1993	25	24	21
4	530	10	10	9	927	37	33	32
5	104	16	15	20	245	31	30	21
6	56	41	41	50	122	55	55	56
7	39	90	90	97	111	53	53	53
8	10	90	90	140	65	58	60	48
9	37	97	97	116	34	44	41	38
>9	157	100	100	114	180	81	81	83

Table 5.1: Impact of rules on statistics for a French Dictionary

Contents	no lexemes				with lexemes			
Rules used	total	1-5	1-5,6	1-5,7	total	1-5	1-5,6	1-5,7
Transitions	1622425	36	27	11	1839467	44	35	24
States								
total	744209	25	23	6	924579	33	30	17
transitions								
outgoing 1	406408	17	17	5	559926	25	25	16
2	159405	25	37	4	179395	33	46	10
3	69126	36	18	7	74217	47	25	18
4	36823	40	23	10	38074	50	32	23
5	21477	44	27	13	21660	38	29	26
6	14634	49	23	14	14772	62	31	33
7	10930	47	22	15	11028	65	34	36
8	8091	51	25	17	8171	69	42	41
9	5488	52	26	20	5500	67	37	47
>9	11827	52	26	20	11836	80	57	65
incoming 1	638844	27	24	7	795083	35	32	18
2	48760	22	21	2	58844	22	23	9
3	18472	13	13	2	22667	18	22	11
4	9008	11	11	2	11611	21	20	11
5	5449	15	14	2	6958	19	19	14
6	3846	16	12	2	4723	20	20	12
7	2898	15	16	3	3372	21	20	11
8	2037	11	12	2	2604	15	17	14
9	1483	12	10	3	2143	18	24	7
10-99	12133	8	9	3	15356	15	17	14
100-999	1231	8	23	14	1180	41	54	40
>999	48	8	23	14	38	345	245	63
in sg lines								
total	319113	19	18	6	448697	28	27	18
length 1	66454	17	15	1	79037	23	24	12
2	25984	15	14	1	36134	31	27	22
3	9476	14	13	2	14823	33	30	22
4	2993	10	8	2	6391	34	32	26
5	1112	11	11	7	2831	35	32	29
6	574	10	10	11	1696	26	26	27
7	240	48	49	60	722	31	30	21
8	148	108	112	135	590	31	32	17
9	193	140	168	48	416	48	40	32
>9	712	79	77	64	1510	55	57	49

Table 5.2: Impact of rules on statistics for a Polish dictionary

Chapter 6

Usage

The algorithms presented here are given for the deterministic acyclic finite-state automata (i.e. the classical automata). This makes their comparison with other algorithms easier, as the classical automata are the standard reference. However, versions for deterministic acyclic finite-state automata with final transitions (DAFSA-FT) were implemented. Actually, most implementations have implicit states, so all features of the states are moved to the transitions. This means that they in fact use DAFSA-FTs, but they are not minimal!

6.1 Spelling Correction

The purpose of spelling correction is to verify whether a given text is correct linguistically, i.e. whether it constitutes a sequence of statements that are correct statements of the language the text is written in, and if not, propose means to correct it. The text is correct if all words of the text are:

- correct words in the language the document is written in,
- and correct in their context.

Possibly incorrect words can be detected by dictionary lookup. A word that is not present in the dictionary may still be a valid word – language is something that changes all the time. However, errors are less likely to be in a dictionary. In the past, statistical techniques were used to detect erroneous words. A program called TYPO ([Pet80]) used frequencies of digrams and trigrams to find possibly incorrect words. Words that contained digrams or trigrams of low frequency were judged to be (possibly) incorrect.

For all incorrect words and expressions, a spelling checker should produce a list of their possible replacements. The replacements are words present in the dictionary that are "similar" to the given word in a certain aspect. This similarity can be measured in a number of ways. The most important involve:

- *edit distance*
- similarity of roots and morphological categories
- pronunciation

Those measures reflect the origins of errors. Errors can be the result of:

- modifications to the translation of the word's image in the brain to the sequence of nerve impulses that drive the muscles that make fingers hit keys on a keyboard; these erroneous words lie within a short edit distance from their correct forms;
- lack of knowledge of the morphology of certain words, the use of improper flectional or derivational paradigm (e.g. in Polish, the nominalisation of *otwarty* gives *otwartość*, but the nominalisation of *uparty* does not give *upartość*, but *upór*); the correct forms can be found with the morphological analysis;
- lack of knowledge of the proper spelling of words with the knowledge of their (at least approximate) pronunciation; the correct forms can be found by looking at words that are pronounced in a similar way.

Those measures rely solely on the properties of individual words. The lists of replacements can further be reduced by checking the context of words. Certain words cannot appear in certain contexts. The disambiguation can be done using syntactic and semantic analysis, statistical methods, or machine learning. Although finite-state tools are used with success on the syntactic level (see e.g. [RS95], we have not developed our own tools for that purpose, so these methods are not described here. The reason those methods were not implemented is that it would involve the syntactic level, while this work stays on the morphology and lexicon level (see fig. 1.1). For a description of methods that do not use finite-state tools, see [CDK$^+$91], [CDGK92], [CGM93], [GC92], [HJM$^+$82], [JHMR83], [KS81], and [Atw87]. The syntactic level (and higher levels) can also be used to detect words that are correct words of the language and are present in the lexicon, but that are used in wrong context. These are most probably incorrect forms of other words.

The *edit distance* ([Dam64]) between two strings specifies how many basic editing operations are needed to convert one string into the other. The basic editing operations are:

- insertion of one symbol

- deletion of one symbol
- change of one symbol
- transposition of two adjacent symbols

Note that if one of the symbols involved in the basic operations is a space, these operations may break words in two, or join them.

The edit distance between two strings (or words) X and Y of lengths m and n respectively, can be defined as follows ([DC92], [OG94], [Ofl96]):

$$
\begin{array}{rcll}
ed(X_{1\ldots i+1}, Y_{1\ldots j+1}) & = & ed(X_{1\ldots i}, Y_{1\ldots j}) & \text{if } x_{i+1} = y_{j+1} \\
 & & & \text{(last characters} \\
 & & & \text{are the same)} \\
 & = & 1 + min\{ed(X_{1\ldots i-1}, Y_{1\ldots j-1}, & \text{if both } x_i = y_{j+1} \\
 & & \qquad ed(X_{1\ldots i+1}, Y_{1\ldots j}), & \text{and } x_{i+1} = y_j \\
 & & \qquad ed(X_{1\ldots i}, Y_{1\ldots j+1})\} & \\
 & & & \text{(last two characters} \\
 & & & \text{are transposed)} \\
 & = & 1 + min\{ed[X_{1\ldots i}, Y_{1\ldots j}), & \text{otherwise} \\
 & & \qquad ed(X_{1\ldots i+1}, Y_{1\ldots j}), & \\
 & & \qquad ed(X_{1\ldots i}, Y_{1\ldots j+1})\} & \\
ed(X_{1\ldots 0}, Y_{1\ldots j}) & = & j & 0 \leq j \leq n \\
ed(X_{1\ldots i}, Y_{ldots0}) & = & i & 0 \leq i \leq m \\
ed(X_{1\ldots -1}, Y_{1\ldots j}) & = & ed(X_{1\ldots i}, Y_{1\ldots -1}) = max(m, n) & \text{(Boundary definitions)}
\end{array}
$$

The recursive part of the definition can further be divided into three sections. The first one is obvious: if the last characters are the same, then the edit distance is the same as if we remove them from the ends of the words. For example, $ed(aba, bba) = ed(ab, bb) = ed(a, b)$.

The next section provides a formula that should be applied in case the last letter of the first word is the penultimate letter of the second word, and *vice versa*. In such situation a transposition of the last two letters of one the words could be one of the transformations that need to be done to obtain the other word, e.g. in case of $(ababab, ababba)$. However, there are other cases that give similar results: deletion (e.g. $(aba, abab)$), and insertion (e.g. $(abab, aba)$).

The last section deals with other cases, i.e. when the last letters of the words are neither equal to one another, nor equal to the penultimate letters of the other word. This situation can be the result of one of three operations: replacement of the last letter (e.g. $(ababa, ababb)$), deletion (e.g. (a, ab)), or insertion (e.g. (ab, a)).

Using the edit distance, we can define the set of plausible corrections of the word $w \in \Sigma^*$ as:

$$C = \{w_i \in L |\ ed(w, w_i) \leq t\}$$

where t is the maximum edit distance between the word w and its correction.

It is possible to attribute different costs to different editing operations. In the following formula, IC is the insertion cost, DC is the deletion cost, CC is the change cost, and TC is the transposition cost. By manipulating the constants, it is possible to favor one type of operation over another.

$$
\begin{array}{rcll}
ed(X_{1\ldots i+1}, Y_{1\ldots j+1}) & = & ed(X_{1\ldots i}, Y_{1\ldots j}) & \text{if } x_{i+1} = y_{j+1} \\
 & & & \text{(last characters} \\
 & & & \text{are the same)} \\
 & = & min\{ed(X_{1\ldots i-1}, Y_{1\ldots j-1} + TC, & \text{if both } x_i = y_{j+1} \\
 & & \qquad ed(X_{1\ldots i+1}, Y_{1\ldots j}) + IC, & \text{and } x_{i+1} = y_j \\
 & & \qquad ed(X_{1\ldots i}, Y_{1\ldots j+1}) + DC\} & \\
 & & & \text{(last two characters} \\
 & & & \text{are transposed)} \\
 & = & min\{ed[X_{1\ldots i}, Y_{1\ldots j}) + CC, & \text{otherwise} \\
 & & \qquad ed(X_{1\ldots i+1}, Y_{1\ldots j}) + IC, & \\
 & & \qquad ed(X_{1\ldots i}, Y_{1\ldots j+1}) + DC\} & \\
ed(X_{1\ldots 0}, Y_{1\ldots j}) & = & j \cdot IC & 0 \leq j \leq n \\
ed(X_{1\ldots i}, Y_{1\ldots 0}) & = & i \cdot IC & 0 \leq i \leq m \\
ed(X_{1\ldots -1}, Y_{1\ldots j}) & = & ed(X_{1\ldots i}, Y_{1\ldots -1}) = max(m,n) \cdot IC & \text{(Boundary definitions)}
\end{array}
$$

The editing operations that produce typographical errors are reversible, i.e. if a string was obtained by a basic editing operation from a correct word, then the correct word can be obtained by performing a single basic editing operation complementary to that that lead to the error. Insertion and deletion are mutually complementary, and change and transposition are self-complementary.

Because the generation of all strings that lie within the specified edit distance and testing their presence in a lexicon would be very time-inefficient, special techniques were used in the past to limit the number of candidates for corrections. Joseph Pollock and Antonio Zamora ([PZ84]) used a *skeleton key* and an *omission key* to index all words in a dictionary that could be corrections of a given incorrect string. The words with the same value of the key as the incorrect string were then tested whether they lied within the given edit distance from the incorrect string. The skeleton key contained the first letter of the incorrect string, unique consonants in order of occurrence, and the unique vowels in order of occurrence.

The omission key was used to find corrections that were not handled by the skeleton key. It was constructed by sorting the unique consonants of the incorrect string in the following order: KQXZVWYBFMGPDHCLNTSR, and then appending the unique vowels in their original order. The order of consonants was established empirically from the frequencies of omissions of those letters from words in corpora.

The pioneering work on spelling correction with finite-state automata was done by Kemal Oflazer ([OG94] and [Ofl96]. We present here his approach along with our modifications for the Polish language.

To find all corrections of the given (erroneous) word, we need to find all paths from the start state to one of the final states, so that when labels on the transitions along a path are concatenated, the resulting string is within a given edit distance threshold t of the erroneous word. As searching needs to be fast, one cannot search the whole automaton. As soon as it becomes clear that the current path does not lead to a solution, we should abandon it. To detect such cases, a notion of *cut-off edit distance* was introduced ([DC92], [OG94], [Ofl96]). The cut-off edit distance is the minimum edit distance between an initial substring of the incorrect word, and any initial substring of the candidate (correct) word. For the edit distance t, the cut-off edit distance $cuted(X, Y)$ of the string X of length m, and the partial candidate word Y of length n is defined as follows:

$$cuted(X_{1\ldots m}, Y_{1\ldots n}) = \min_{l \leq i \leq u} ed(X_{1\ldots i}, Y_{1\ldots n}), l = min(1, n-t), u = max(m, n+t)$$

The range of lengths of the substrings of X checked is from $n-t$ to $n+t$, except close to word boundaries. If $n-t$ is less than one, one cannot take a null string, and if $n+t$ is more than the length of X, one cannot append additional characters at the end of X. The depth-first search in the automaton with the use of the cut-off edit distance gives the algorithm ([Ofl96]) given in fig. 6.1.

procedure spell(state, word, candidate, max_distance)
  for all $a \in \Sigma$ such that $\exists_{t \in Q} \delta(state, a) = t$
    candidate' $\leftarrow$ candidate . a
    if cuted(word,candidate') $\leq$ max_distance
      spell(t, word, candidate', max_distance)
    if ed(word, candidate') $\leq$ max_distance $\wedge$ t $\in F$
      print candidate'

Figure 6.1: Spelling correction with finite-state automata

Note that it is possible to use automata that store words with annotations, such as those used in the morphological analysis, with the algorithms in fig. 6.1 by substituting the test for t $\in F$ for a test on the presence of a transition labeled with the annotation separator. A dynamic programming technique can be used to reuse computed values of the edit distance so that the speed of the algorithms can be improved ([DC92]). Closer examination of the formula for computing the edit distance shows that the value of $ed(X_{1...i}, Y_{1...j})$ depends on $ed(X_{1...i-1}, Y_{1...j})$, $ed(X_{1...i-1}, Y_{1...j-1})$, $ed(X_{1...i}, Y_{1...j-1})$, and $ed(X_{1...i-2}, Y_{1...j-2})$. The values of ed can be put into a matrix H, so that $ed(X_{1...i}, Y_{1...j})$ is in $H(i, j)$. Looking at the algorithm from fig. 6.1, we can see that before we move to the row j, the values from the row $j-1$ are already computed. They stay the same for all transitions tried in the loop. So we do not need to recompute $H(i-2, j-2)$, $H(i-1, j-1)$, $H(i, j-1)$. We do need to recompute $H(i-1, j)$. As the algorithm for computing the cut-off edit distance proceeds from the rows with smaller indices to those with bigger ones, the value of $H(i-1, j)$ is always computed before it is needed in the calculation of $H(i, j)$.

In many languages, one can find words that have different spelling, but that are pronounced either identically, or in a similar way. In Polish, for example, the pairs *ó* – *u* and *ż* – *rz* have identical pronunciation, and although there are native Polish speakers that distinguish between *h* (voiced), and *ch* (not voiced), most Polish speakers pronounce both as *ch*. As the number of speakersreading books that cannot write properly grows, new types of errors emerge, e.g. *ą* – *on*, *ą* – *om*, *ę* – *en*, and *ę* – *em*. The pairs *ó* – *u* and *h* – *ch* can be treated as typographical errors. However, the pair *ż* – *rz*, and the new pairs involving nasal vowels cannot. Finding the words that are pronounced in a similar manner to a given word is usually done with transducers. One transducer level contains the inflected forms, the other one – symbols that represent their pronunciation. The search for candidates for correct forms is done on the pronunciation level, possibly using the edit distance. For languages with highly regular pronunciation (such as Polish), one can do away with automata-acceptors. A solution that takes into account the possibility of replacing a pair of characters with a single character and *vice versa* is given in fig. 6.2. Note that from the four transformations that produce errors, only one – change is affected. This is because it highly unlikely that someone would accidentally drop or insert two consecutive letters, or would move one letters by two positions. This algorithm is sufficient for Polish. A more elegant and general solution could involve weighted transducers and the local extension algorithm ([RS95], [Moh97]).

Application of morphological analysis for correction of lexical errors was proposed in a number of papers, e.g. [CGM93], [CDGK92]. The method requires the following steps:

1. checking if a word is in the lexicon;

2. morphological analysis of the incorrect word yielding the lexeme (or the root when using derivational morphology) and the corresponding morphological

procedure spell(state, word, candidate, i, j, max_distance)
  for all $a \in \Sigma$ such that $\exists_{t \in Q} \delta(state, a) = t$
    candidate' ← candidate . a
    if $match(word_{i...i}, candidate'_{j...j+1})$
      spell(t, word, candidate', i, j+1, max_distance)
      l1 ⟵ length(word)-1
      l2 ⟵ length(candidate')-2
      if $t \in F \wedge ed(word_{1...l1}, candidate_{1...l2}, i-1, j-2) < max\_distance$
        print candidate'
    if cuted(word, candidate', i, j) ≤ max_distance
      spell(t, word, candidate', i+1, j+1, max_distance)
      if $match(word_{i...i+1}, candidate'_{j+1...j+1})$
        spell(t, word, candidate', i+2, j+1, max_distance)
        l1 ⟵ length(word)-2
        l1 ⟵ length(candidate')-1
        if t $\in F \wedge$
          $ed(word_{1...l1}, candidate'_{1...l2}, i-2, j-1) \leq$ max_distance
          print candidate'
    if t $\in F \wedge$ ed(word, candidate', i, j) ≤ max_distance
      print candidate'

Figure 6.2: Spelling correction with finite-state automata with modifications

categories;

3. morphological generation of the inflected form on the basis of the lexeme (or the root when using derivational morphology) and the categories obtained form the morphological analysis.

We have implemented all those steps, but no attempt was made yet to integrate them. In the second step, a morphological analysis of a word not present in the lexicon should be performed. Such analysis is described in section 6.3.3, page 72, the construction of a finite-state automaton used in it is discussed in chapter 5, page 49, and the data for it is discussed in section 3.3, page 24. Our analysis is flectional, so it will not correct errors such as e.g. *upartość* that could be an incorrect nominalization of *uparty*, while the proper nominalization is *upór*. It will, however, detect errors such as e.g. *piesa*, recognizing that (among possibly other candidates) the lexeme could be *pies*, and the categories should be *noun, animal,* *single, genitive*. The morphological generation with input data being the lexeme and the categories obtained in the previous step would yield *psa*. The morphological generation (analogous to morphological analysis) is described in section 6.3.1, page 71. Note that if the lexeme (or root) is not present in the lexicon, or if no inflected forms of the given categories can be found for it, no candidates will be generated.

6.2 Restoration of Diacritics

Restoration of diacritics can be seen as a special case of spelling correction. In particular, the same division on word-based methods and methods using the context is present. The methods that use the context are similar to those used in spelling correction. For a description of those methods see [Yar94a] and [Yar94b].

The word-based methods for the restoration of diacritics are also the basis for the context-based methods by providing them with choices. The techniques used in them are partially different from those used in spelling correction. In particular, if the word from a text is present in the lexicon, it does not mean that it is correct; all words that when deprived of diacritics give the word should be given.

We define a function *strip* that converts the letter with diacritics in the word being the argument of the function to the letters that have the same shape, but do not have diacritics. That function defines a relation:

$$R_s \equiv \{(w_1, w_2) \in (\Sigma^* \times \Sigma^*) |\ strip(w_1) = strip(w2)\}$$

The purpose of the word-based method for the restoration of diacritics is to find all words that are in R_s with the word in question. This can be achieved by the use of statistical techniques measuring the frequencies of sequences of letters in words (see [Dac97]), but the standard technique is to use a lexicon. In particular, the lexicon in form of a finite-state automaton offers the usual advantages: compact representation and great speed of processing. Figure 6.3 gives the algorithm for finding all words in the lexicon that are in R_s with the given word.

The dot operator in fig. 6.3 represents concatenation. Note that it is also possible to use the same algorithm with the morphological dictionaries, implemented both with transducers and automata-acceptors. The corrections necessary for their adaptation are trivial.

6.3 Morphology

The purpose of morphological processing is either to obtain morphological information about an inflected word form, or to generate the form given its corresponding

procedure restore_diacritics(state, output)
    if state $\in F$
        print(output)
    for each $a \in \Sigma$ such that $\exists_{t \in Q} \delta(state, a) = t$
        restore_diacritics(t, output . a)

Figure 6.3: Restoration of diacritics with finite-state automata

lexeme and morphological categories.

6.3.1 Using Transducers

As transducers translate a string into another string[1], the morphological analysis and generation of inflected forms is straightforward – it consists of following labels from one level in a dictionary-transducer. As the generation is an inversion of the analysis, i.e. we follow the output labels instead of input labels, we will include only the analysis.

procedure analyze(state, word, index, output)
    if state $\in F$
        print(output)
    for each $a \in \Sigma \cup \{\epsilon\}$ such that $\exists_{t \in Q} \delta(state, \epsilon, a) = t$
        analyze(t, word, index + 1, output . a)
    for each $a \in \Sigma \cup \{\epsilon\}$ such that $\exists_{t \in Q} \delta(state, word[index], a) = t$
        analyze(t, word, index + 1, output . a)

Figure 6.4: Morphological analysis with transducers

The dot operator in fig. 6.4 represents concatenation. The result of concatenating a string with an empty string is the same string, i.e. $word.\epsilon = word$. The use of the empty string in transition labels is necessary as the lengths of the strings may

[1] These are string-to-string transducers. There are are also string-to-weight transducers ([Moh97]) that we do not describe here.

not match. It is also useful for the alignment of segments of strings that represent the same features. Such alignment may reduce the size of the transducer[2].

6.3.2 Using Automata - Acceptors

Simple automata - acceptors can be used for morphological analysis. They can provide faster analysis, but the range of languages that can be described with such machines is more limited than in the case of transducers. Also, special encoding techniques are necessary for the contents of dictionaries (see section 3.2.2, page 22, for the instructions on how to prepare data for a dictionary in form of an automaton-acceptor).

Note that the strings in a dictionary have an internal structure. They have two parts: the first is the inflected form of a word, the second – its annotations, describing e.g. the corresponding lexeme or morphological categories. Those parts are separated with an annotation separator. The analysis has two phases corresponding to those parts. The first phase is the recognition of the inflected forms. The second one treats annotations.

Annotations may contain lexemes. The lexemes are coded (see section 3.2.2, page 22) to reduce the size of the dictionary. They must be decoded during the analysis. The decoding consists of copying the inflected form without a few letters from the end, as indicated by the code. The procedures can be found in figure6.5.

Note that it may be useful to introduce prefixes in the dictionary in the same way as in the analysis of unknown words. An additional code would say how many characters should be rejected from the beginning of the word to get a lexeme. This variation has not yet been implemented.

6.3.3 Unknown Words

During the process of tag assignment, the word to be tagged is inverted, a word beginning marker appended at the end of the string, and subsequent transitions in the automaton starting from the start state are traversed if their labels match subsequent characters in the string. A state is reached where there are no matching transitions. If there are transitions in the state that have labels belonging to tags, labels from all paths in the automaton starting from those transitions and ending in final states are printed as tags. Otherwise, the same algorithm applies recursively to all states reached directly from that state.

The process of recognizing words and finding corresponding annotations can be decomposed into a few steps (see fig. 6.6).

[2] Personal communication with Martin Kay

procedure analyze(state, word, index)
  if $\exists_{t \in Q} \delta(state, word[index]) = t$
    analyze(t, word, index + 1)
  if $index > length(word) \wedge \exists_{t \in Q} \delta(state, annotation\_separator) = t$
    lexeme(t, word)

procedure lexeme(state, word)
  for each a such that $\exists_{t \in Q} \delta(state, a) = t$
    annotations(t,$word_{1 \ldots (length(word)-(ord(a)-ord('A')))}$)

procedure annotations(state, output)
  if state $\in F$
    print(output)
  for each a such that $\exists_{t \in Q} \delta(state, a) = t$
    annotations(t, output . a)

Figure 6.5: Morphological analysis with automata-acceptors

The steps described above involve recognition of the prefixes, and decoding the lexemes. If the purpose is to obtain only the lexemes, or only the categories, or if the language in question does not have prefixes (e.g. French), the algorithms above are simplified (appropriate procedures are scrapped. See fig. 6.7 for an example of guessing the categories of inflected forms.

To evaluate how the rules described in sections 5.2.1, 5.2.2, and 5.2.3 (page 50) influence analyses of inflected forms, unknown (i.e. not present in the lexicon) correct Polish words were selected from a corpus. 405 words were chosen – all words that started with “b” and were judged to be correct. Foreign words, abbreviations, and misspellings were rejected. Also, a small percent of words that fell into classes not yet in the lexicon (which is under construction) were rejected as well. Table 6.1 shows the results.

It seems that the rule R6 is best suited for tagging or as a preprocessor for a parser, while R7 is better for a lexicographer, as it can find some additional analyses.

The standard measures of the quality of guessing are:

recall – the percentage of POS tags correctly assigned by the guesser over the

procedure recognize(state, word)
  recognize_letter(invert(word) . end_of_word_marker, 0)

procedure recognize_letter(state, word, letter_number)
  if $\exists_{t \in Q} \delta(state, word[letter\_number]) = t$
    recognize_letter(t, word, letter_number + 1)
  else if $\exists_{t \in Q} \delta(state, annotation\_separator) = t$
    recognize_prefix(state, word, 1)
  else
    for each $a \in \Sigma$ such that $\exists_{t \in Q} \delta(state, a) = t$
      recognize_letter(t, “”, 0)

procedure recognize_prefix(state, word, index)
  if $\exists_{t \in Q} \delta(state, word[index]) = t$
    recognize_prefix(t, word, index + 1)
  if $\exists_{t \in Q} \delta(state, annotation\_separator) = t$
    decode_lexeme(t, $word_{index \ldots length(word)}$)

procedure decode_lexeme(state, word)
  for each a such that $\exists_{t \in Q} \delta(state, a) = t$
    get_letter(t, $word_{1 \ldots (length(word) - (ord(a) - ord(``A")))}$)

procedure get_letter(state, annotation)
  if state $\in F$
    print annotation
  if $\exists_{target\_state} \delta(state, word[letter\_number]) = target\_state$
    for each {target_state: $\exists_{a \in \Sigma} \delta(state, a) = target\_state$}
      get_letter(target_state, annotation . a)

Figure 6.6: Morphological analysis of unknown words

total number of correct POS tags for the word,

precision – the percentage of POS tags the guesser assigned correctly over the total number of POS tags it assigned to the word,

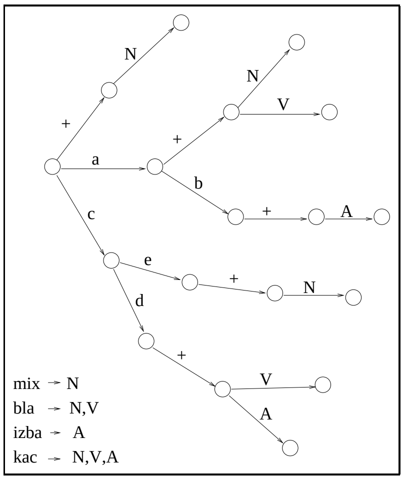


Figure 6.7: Guessing the categories of words. Prefixes and lexemes not present

	with lexemes			no lexemes		
Rules	1-5	1-6	1-5,7	1-5	1-6	1-5,7
Analyses per word	3.05	2.87	3.91	3.03	2.87	3.89
% of correct analyses	73.16	76.46	57.99	73.65	77.09	58.46
% of words without incorrect analyses	77.04	80.74	41.73	78.27	80.99	41.73
% of words without correct analyses	6.17	7.40	5.43	6.17	7.16	5.68
Lexemes per word	1.24	1.17	1.59	n/a	n/a	n/a

Table 6.1: Impact of rules on analyses

coverage – the proportion of words guesser was able to classify, but not necessarily correctly.

We set apart each tenth of morphological data, and we constructed a guessing automaton out of the remaining nine tenths of data. Then we used the words from the separated part to measure the quality of guessing. Table 6.2 shows the results. Mikheev reported recall 95%, precision 85%, and recall 92%.

When comparing our method with that used by Mikheev we should take several circumstances into account:

Rules	1-6	1-5,7
Recall	84	96
Precision	85	75
Recall	99	99

Table 6.2: Average quality of guessing for Polish morphological data

- We used a different language than Mikheev, and indeed the language we used was never used in any experiment of that sort. The impact of that difference may be manifold. As Polish is a typically flectional language with large paradigms, with many defective forms, and with many syncretisms, the precision for Polish may be lowered than for English. E.g. there are frequent *singularia tantum* and *pluralia tantum* (i.e. nouns without plural or singular forms), adjectives and adverbs without comparative and superlative degrees, and verbs without participles.

- Mikheev used corpus-based smoothing. We did no such thing. The smoothing should eliminate some of unproductive rules introduced by very frequent but irregular words (see [BS96] for a discussion of *hapax words*).

- Our method is under constant development, and we see potential for improvements. Introducing an limit on the number of transitions triggering the application of the rule R7 and combining that rule with the rule R6 should give a method with recall close to that displayed in table 6.2 under heading "1-5,7", and precision – under heading "1-6".

- Our method has much better coverage and this feature is language-independent.

- Our method is much, much faster. What we do is to construct an automaton, and then prune it. Mikheev uses Brill's method to obtain rules. It is very time-consuming, as only one rule should be chosen for one run over the data, processing all words. Mikheev obtains only a set of rules, not an automaton. The application of such rules is very slow compared to statistical and finite-state methods. Roches and Schabes ([RS95]) give a method for transforming those rules into a transducer, which gives much higher recognition speed. However, such transformation is itself time-consuming, as it involves local extention and determinization.

- Mikheev and others recognized only categories, whereas we recognize the canonical forms as well.

6.4 Perfect Hashing

Hashing maps keywords from a given set of n keywords into numbers (addresses). If the mapping is a bijection into a set $1 \ldots n$ (or $0 \ldots n-1$), the hashing is said to be "perfect".

Perfect hashing can be implemented in form of an automaton ([Roc95]). We treat the keywords as words of the language accepted by an automaton. Therefore, we can build an automaton that accepts that language, and consequently, stores all those words. Because the language contains a finite number of words, the automaton must be acyclic. To be efficient, it must be deterministic.

We need to establish a mapping between words and their cardinal numbers. Note that if all transitions in an automaton are sorted by their labels, such mapping implicitly exists in the automaton. Words are ordered by the order of their appearance in the automaton when traversed using the postorder method. So the mapping can be provided by counting all words that precede the given word in the automaton.

Counting words can mean traversing the whole automaton, so it is computationally impractical. However, it is possible to precompute some values so that the search be effective. Let us look again at another sample automaton (deterministic version) at fig.6.8.

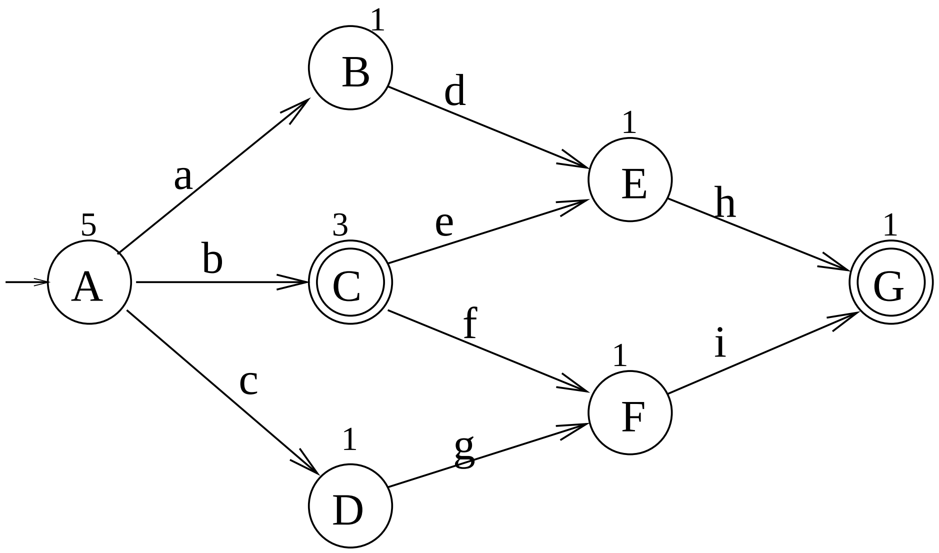


Figure 6.8: Sample deterministic finite-state automaton implementing perfect hashing

Small numbers near state circles tell how many words (suffixes of the words accepted by the automaton) are accepted by the part of the automaton that is reachable from that state, that state including. Formally, that number can be computed as:

$$C(q) = |\{w \in \Sigma^* : \delta(q, w) \in F\}|$$

That number can be calculated recursively using dynamic programming:

$$C(q) = |q \cap F| + \bigcup_{q_i \in Q : (\exists_{w \in \Sigma} : \delta(q,w) = q_i)} C(q_i)$$

These numbers can be precomputed and stored at their respective states, so that they be available during dictionary lookup.

To find the number associated with a given word, we find that word in the automaton, doing some calculations at states that we pass by. What we are looking for is the number of words that precede the given word in the automaton. E.g. the word *bfi* is the fourth word recognized by the automaton from fig. 6.8. To find that it is indeed the fourth word, we first look at the start state – A. As we must go through that state, its count does not count (it is the total number of words of the language accepted by the automaton). We see that b is the label of the second transition. The first transition leads to a state that has its count value set to 1. this means that there is one word in the automaton that precedes all words that start with b. We add it to the total, which is initially zero. The transition labeled b leads us to the state labeled with C. This state is final, so because shorter words come before longer ones, we must add 1 to the total when we go through. We add another 1 as well, because f is the second transition in that state, and the first transition leads to a state that has count value set to 1. Now the total is 3. With f, we go to F, which is not final, so there is nothing to add to the total, and then we go to G. G is a final state, so *bfi* is accepted. The total is 3, meaning there are 3 words before *bfi* in the automaton, so *bfi* is the fourth.

Chapter 7

Summary

A new type of automata, and a number of new algorithms are presented in this dissertation. Finite-state automata with final transitions (FSA-FT) can hold exactly the same information as classical finite-state automata. However, they can be smaller than their classical counterparts, i.e. they can have less states and less transitions. Most implementations have implicit states, and the features of states are stored in transitions, so in fact they use FSA-FT. Unfortunately, those automata are simple transformations of minimal classical FSA, so they have the same (subminimal) number of states and transitions. We use the new kind of automata explicitly from the beginning, making our automata smaller.

We present two new algorithms for constructing deterministic acyclic finite-state automata. Their main characteristics is that they are *incremental*. The main implication of this fact is that they require *much* less memory than other methods that need to have all the data in memory before they start. For one of the dictionaries we used in our experiments that would mean having to store about 2 GB, which is practically impossible in the current technology. The use of DAFSA-FT makes our automata even smaller, and as the construction algorithms require memory proportional to the size of the automaton, the memory requirements of those algorithms are additionally lowered in our implementation. We are aware of another algorithm that builds deterministic acyclic FSA incrementally ([Rev91]). That algorithm (by Dominique Revuz) has similar characteristics to our algorithms. However, Dominique Revuz constructs automata in two steps: the first produces an automaton after a pseudo-minimization, the second one is the true minimization. The automaton after the pseudo-minimization phase can be 8 times larger ([Rev91]) than the minimal one, thus increasing the intermediate memory requirements. Our algorithms are also more flexible than his. The algorithm for sorted data makes it possible to perform operations on the automaton on-the-fly, because the parts once minimized are no longer changed. Our algorithm for data in arbitrary order[1] does not require data to be sorted, so it can be fed directly from other programs that produce data, minimizing the storage space for intermediate results.

Words not present in the lexicon pose many problems for the treatment of free texts. We present a method for constructing a guesser from the data for a morphological lexicon. Our guesser is in form of a finite-state automaton, with the usual advantages of these machines: great speed and compact representation. The guesses are made on the basis of word endings and beginnings. The method predicts both the corresponding lexemes, and the categories associated with words. Compared to traditional approaches, i.e. to using textbook rules, our method relies directly on data, so all exceptions that are so easily overlooked in manual processing are included. Compared to Eric Brill's solution ([Bri93] and [Bri95]), we infer the rules directly from the lexicon, not from the text, so we have more accurate information as the input. There is another lexicon-based method for guessing rule inference ([Mik97]). Andrei Mikheev uses Eric Brill's learning techniques and rule schemata, while in our approach the rules are implicit in the automaton. In fact, the guesser is an automaton that is a transformation of another automaton built from data prepared in a special way. We do not use rule schemata, so the rules we infer are more general.

The same algorithm can be used for acquisition of a morphological lexicon. A core lexicon based on two-level morphology should be built. The lexicon should contain all rules. The guesser should have morphological descriptions as annotations, i.e. the corresponding lexeme and the continuation classes. As at the moment of writing we do not have a two-level morphology for Polish, we could not test our ideas yet.

Our enhancement for Kemal Oflazer's algorithm for spelling correction makes it possible to give proper corrections for incorrect Polish words. The problem was that there are frequent errors that pass the edit distance barrier, but are specific enough to be recognized and corrected. Our solution is more of a kludge; a more elegant solution could use weighted transducers.

We put the ideas and algorithms in context, and refer the reader to other sources when appropriate. Our solutions are compared to others, and the advantages of our methods are pointed out clearly. Our algorithms are implemented in C++, and are freely available in the Internet for non-commercial purposes at the following url: `http://www.pg.gda.pl/~jandac/fsa.html`.

The theses stated in the introduction:

- **It is possible to construct minimal, deterministic, acyclic, finite-**

[1] This algorithm was also developed independently by Richard E. Watson and Bruce Watson

state automata using less memory than in all previously known algorithms while maintaining the speed of the construction process.

- **It is possible to construct automata that guess canonical forms of words and their annotations directly from data for a morphological lexicon. Little or no linguistic knowledge is required for that process. We state that this process can be faster than the previously known algorithm serving the same purpose, and that it maintains similar quality of guessing.**

have been upheld. We proposed two algorithms for construction of minimal, deterministic, acyclic, finite-state automata that have the desired features. We have also presented a method for construction of a finite-state automaton that guesses the categories of unknown words, and guesses their corresponding canonical forms. That method is indeed faster than other methods with similar quality of guessing, and requires little or no linguistic knowledge other than the knowledge embedded in the data.

Appendix A

FSA Package

A.1 fsa_build

NAME

fsa_build - build an automaton from sorted data (words)

SYNOPSIS

fsa_build [*options*] [<*infile*] [>*outfile*]

DESCRIPTION

fsa_build builds an automaton from a list of words. The list should be sorted. It is not important which LOCALE is used for sorting as long as the order is consistent.

The output is an automaton that can be used by *fsa_spell, fsa_accent, fsa_morph, fsa_guess, fsa_hash,* and *fsa_prefix.*

OPTIONS

-O make the resulting automaton smaller. The time required to build the automaton is much greater. How much greater depends on compile options used during compilation of *fsa_build.* See Makefile and INSTALL from the distribution for an explanation of various compile options. The default options compress the automaton the most. This option cannot be used with -N option.

-i *input_file* specifies input file. That file should contain a list of words, one word per line. In absence of -i option, standard input is used instead.

-o *output_file* specifies output file, i.e. where the automaton should be placed. In absence of -o option, standard output is used instead.

-A *annotation_separator* specifies a character that separates words from morphological annotations.

-X prepares an index a tergo that is used to predict word categories. This option is available only if the program was compiled with A_TERGO compile option. Specifying PRUNE_ARCS compile option helps making the resulting automaton smaller and faster. These compile options are on by default. The format of data depends on compile options used to build the *fsa_guess* program, and affects the outcome of that program.

For fsa_guess compiled without GUESS_LEXEMES, the input data should be a list of inverted words with annotations. Each line should contain an inverted word (i.e. the first character should be the last character of the word, the second one - the penultimate one, and so on. This inverted word should be followed immediately by a filler character and an annotation separator, and then by grammatical annotations. They specify some morphological properties of words, such as number, gender, etc.

Assuming that a file *file* contains data in 3 columns: inflected word, canonical form, annotations, the following incantation:

```
awk '{s=""; for(i=1;i<=length($1);i++) s = substr($1,i,1) s;
printf "%s_+%s\n",s,$3;}' file | sort -u > file.idx
```

prepares data for the a tergo index. The incantation should be all in one line. For more detail see the contents of *prep_atg.awk* file included in the distribution. The standard name extension for automata prepared in this way is *atg*.

For fsa_guess compiled with GUESS_LEXEMES, but without GUESS_PREFIX, one data line should contain the same information as above, but an additional annotation separator, a code, and the ending of the corresponding lexeme must be inserted in front of the first annotation separator. The code specifies how many characters from the end of the inflected word must be rejected before appending the ending of the lexeme. The code is a letter. 'A' means there are no characters to reject, 'B' - there is one, 'C' - 2, and so on. For more detail see *prep_atl.awk* file included in the distribution. The standard name extension for automata prepared in this way is *atl*.

For fsa_guess compiled with both GUESS_LEXEMES, and GUESS_PREFIX, data lines are similar to those specified above. For inflected forms that do not contain flectional prefixes, an additional annotation separator is added after the first one (see *prep_atp.awk* file included in the distribution). For inflected forms that do contain flectional prefixes, the prefix is removed from the inverted word leaving the filler character, and it is placed between two annotation separators in simple, noninverted form. The *prep_atp.awk* file does not contain code for recognizing prefixes; it should be modified for individual languages and recognize specific morphological categories. Only prefixes that differentiate between forms that have the same suffix should be recognized. The standard name extension for automata prepared in this way is *atp.*

-N number entries. All entries are numbered according to their position (line number) in the input stream. This is so called perfect hashing. This option works only if the program was compiled with NUMBERS compile option. This option (i.e. -N) cannot be used with -O option.

-v print version details with compile options used.

EXIT STATUS

0 OK

1 Invalid options, or lack of a required option.

4 Not enough memory.

SEE ALSO

fsa_ubuild, fsa_spell, fsa_accent, fsa_prefix, fsa_guess, fsa_hash, fsa_morph

BUGS

Send bug reports to the author: Jan Daciuk, jandac@pg.gda.pl.

A.2 fsa_accent

NAME

fsa_accent - find words that deprived of diacritics match words from input

SYNOPSIS

fsa_accent [*options*] [<*infile*] [>*outfile*]

DESCRIPTION

fsa_accent reads lines from the input. Each line contains one word. Every word is looked up in specified dictionaries. All words that when deprived of diacritics match the input word are printed.

OPTIONS

-d *dictionary*
use that dictionary. Several dictionaries may be given. At least one dictionary must be specified. Dictionaries are automata built using *fsa_build* or *fsa_ubuild.*

-a ***accent_file*** use *accent_file* to establish relationships between characters with diacritics, and those without them. The first character of the *accent_file* is a comment character. All lines that begin with it are comments, and are ignored. All other lines specify one character without diacritics for one or more characters with diacritics. The first character in a line is a character without diacritics. It is followed by one or more spaces and HT characters, and then by a string of characters without diacritics, followed immediately by new line. There should be no spaces between characters with diacritics. For each word containing a character found in the first column in the *accent_file* words containing that character as well as characters listed to the right of it in the *accent_file* will be looked up.

-i ***input_file*** specifies an input file, i.e. a file with words that lack accents. More input files can be specified in that way. If the option is not present, standard input is used.

-l ***language_file*** specifies a file that hold language specific information, i.e. (for now) characters that form words, and pairs of (lowercase, uppercase) characters for case conversion. If the option is not specified, latin letters with standard case conversions wil be used.

Note: whether this information will be used or not depends on the module used for I/O handling (one_word_io or text_io).

-v prints version details.

EXIT STATUS

0 OK

1 Invalid option used, or lack of a required option.

2 Dictionary file could not be opened.

4 Not enough memory.

5 Accent file could not be opened or used.

SEE ALSO

fsa_ubuild, fsa_build, fsa_spell, fsa_prefix

BUGS

Send bug reports to the author: Jan Daciuk, jandac@pg.gda.pl.

A.3 fsa_guess

NAME

fsa_guess - guess lexeme and categories of a word

SYNOPSIS

fsa_guess [*options*] [<*infile*] [>*outfile*]

DESCRIPTION

fsa_guess reads lines from the input. Each line contains one word. For each word, its probable categories are printed, based on the contents of a dictionary of (category, word ending) pairs. If the program has been compiled with GUESS_LEXEMES option, and the dictionary has been prepared accordingly, not only categories, but lexemes as well are printed.

OPTIONS

-d *dictionary*

use that dictionary. Several dictionaries may be given. At least one dictionary must be specified. Dictionaries are automata built using *fsa_ubuild* or *fsa_build* with -X option. Data for the automata must be prepared in a special way.

If the automata are to be used to predict only the categories, each line of the input to should contain inverted word with the beginning (the end when inverted) of the word marked with the filler character, followed by an annotation separator, and followed by tags. See *prep_atg.awk* script available in the package.To treat such dictionaries *fsa_guess* should *not* be compiled with GUESS_LEXEMES compile option. The standard name extension for dictionaries prepared in this way is *atg*.

If *fsa_guess* is to guess also lexemes, it must be compiled with GUESS_LEXEMES compile option, but without GUESS_PREFIX, and the input to *fsa_build* must contain in each line: the inflected form, annotation separator, a code, lexeme ending, annotation separator, and tags (annotations). The code specifies how many characters from the end of the inflected form must be deleted before appending there the lexeme ending to get the lexeme. It is one character. To calculate the number, take the character code and substract 65 (character code for 'A') from it. See *prep_atl.awk* script available in the package. The standard name extension for automata prepared in this way is *atl*.

To make *fsa_guess* take into account information included in prefixes, it must be compiled with GUESS_PREFIX. In data lines for *fsa_build,* the first annotation separator is replaced by two annotation separators for entries that do not contain prefixes, otherwise the prefix is deleted from the inverted inflected form leaving the filler character, and placed between the two annotation marks. The standard name extension for automata prepared in this way is *atp*.

-g makes fsa_guess work as if it were compiled without GUESS_LEXEMES. This option is available only if the program was compiled with GUESS_LEXEMES.

-p makes fsa_guess work as if it were compiled without GUESS_PREFIX. This option is avalaible only if the program is compiled with GUESS_PREFIX.

-i ***input_file*** specifies an input file - file that contains words which categories should be guessed. More than one file can be specified (i.e. the option can be used more than once). In absence of -i option, standard input is used.

-l ***language_file*** specifies a file that holds language specific information, i.e. (for now) characters that form words, and pairs of (lowercase, uppercase) characters for case conversion. If the option is not specified, latin letters with standard case conversions will be used.

-v print version details.

EXIT STATUS

0 OK

1 Invalid options, or lack of a required option.

2 Dictionary file could not be opened.

4 Not enough memory.

SEE ALSO

fsa_ubuild, fsa_build, fsa_spell, fsa_prefix, fsa_accent, fsa_hash, fsa_morph

BUGS

Send bug reports to the author: Jan Daciuk, jandac@pg.gda.pl.

A.4 fsa_hash

NAME

fsa_hash - convert words to numbers, and numbers to words

SYNOPSIS

fsa_hash -N [*options*] [<*infile*] [>*outfile*]
fsa_hash -W [*options*] [<*infile*] [>*outfile*]

DESCRIPTION

fsa_hash reads lines from the input. Each line contains either one word, or one number. Every word is looked up in specified dictionaries. If the word is found, its number is printed, if not, -1 is printed. Every number is treated as a word number. The word corresponding to that number is printed, or an error message if the word is not present in the dictionary. Note that *fsa_hash* is only a demo of what the underlying funtions can do.

OPTIONS

-d *dictionary*
use that dictionary. Several dictionaries may be given. At least one dictionary must be specified. Only the first dictionary is used. Dictionaries are automata built using *fsa_build* or *fsa_ubuild.*

-N convert words into numbers.

-W convert numbers into words.

-i input_file specifies a file with words to be corrected. More than one file can be specified in this way (i.e. the option can be used more than once). In absence of this option, standard input is used.

-l ***language_file*** specifies a file that hold language specific information, i.e. (for now) characters that form words, and pairs of (lowercase, uppercase) characters for case conversion. If the option is not specified, latin letters with standard case conversions wil be used.

Note: whether this information will be used or not depends on the module used for I/O handling (one_word_io or text_io).

-v prints version details.

EXIT STATUS

0 OK

1 Invalid option, or lack of a required option.

2 Dictionary file could not be opened.

4 Not enough memory.

SEE ALSO

fsa_ubuild, fsa_build, fsa_accent, fsa_prefix, fsa_spell, fsa_morph, fsa_guess

BUGS

Send bug reports to the author: Jan Daciuk, jandac@pg.gda.pl.

A.5 fsa_morph

NAME

fsa_morph - perform morphological analysis of a word

SYNOPSIS

fsa_morph [*options*] [<*infile*] [>*outfile*]

DESCRIPTION

fsa_morph reads lines from the input. Each line contains one word. For each (inflected) word, its morphological analysis is printed. It consists of one or more pairs (lexeme, tag).

OPTIONS

-d *dictionary*
use that dictionary. Several dictionaries may be given. At least one dictionary must be specified. Dictionaries are automata built using *fsa_ubuild* or *fsa_build*. The data for them must be prepared in a special way. Each line of data consists of the inflected form, followed by a separator, followed by a code, followed by the ending of the lexeme, followed by the separator, followed by tags. The code specifies how many characters from the end of the inflected form are different from last characters of the lexeme (i.e. how many characters constitute the inflected word ending). 'A' means that no characters are to be rejected, 'B' - 1, 'C' - 2, and so on. Though the data is different from the normal word list format, the automaton is the same (its content is different), so the magic number is the same as with the normal data.

-i *input_file* specifies an input file - file that contains words that should be analysed. More than one file can be specified (i.e. the option can be used more than once). In absence of -i option, standard input is used.

-l *language_file* specifies a file that hold language specific information, i.e. (for now) characters that form words, and pairs of (lowercase, uppercase) characters for case conversion. If the option is not specified, latin letters with standard case conversions wil be used.

Note: whether this information will be used or not depends on the module used for I/O handling (one_word_io or text_io).

-v print version details.

EXIT STATUS

0 OK

1 Invalid options, or lack of a required option.

2 Dictionary file could not be opened.

4 Not enough memory.

SEE ALSO

fsa_ubuild, fsa_build, fsa_spell, fsa_prefix, fsa_accent, fsa_guess, fsa_hash

BUGS

Send bug reports to the author: Jan Daciuk, jandac@pg.gda.pl.

A.6 fsa_prefix

NAME

fsa_prefix - print all words beginning with the specified prefix

SYNOPSIS

fsa_prefix [*options*] [<*infile*] [>*outfile*]

DESCRIPTION

fsa_prefix reads lines from the input. Each line contains one word or nothing. The word is treated as a prefix. All words in the specified dictionaries that have the word as their prefix are printed.

If the word is empty, the whole contents of the dictionary is printed.

OPTIONS

-d *dictionary*
use that dictionary. Several dictionaries may be given. At least one dictionary must be specified. Dictionaries are automata built using *fsa_ubuild* or *fsa_build.*

-i *input_file* specifies a file with prefixes to be completed. More than one file can be specified in that way (i.e. the option can used more than once). In absence of this option, standard input is used.

-l *language_file* specifies a file that hold language specific information, i.e. (for now) characters that form words, and pairs of (lowercase, uppercase) characters for case conversion. If the option is not specified, latin letters with standard case conversions wil be used.

Note: whether this information will be used or not depends on the module used for I/O handling (one_word_io or text_io).

-v print version details.

EXIT STATUS

0 OK

1 Invalid options, or lack of a required option.

2 Dictionary file could not be opened.

4 Not enough memory.

SEE ALSO

fsa_ubuild, fsa_build, fsa_spell, fsa_accent

BUGS

Send bug reports to the author: Jan Daciuk, jandac@pg.gda.pl.

A.7 fsa_spell

NAME

fsa_spell - find words in dictionary or propose replacements

SYNOPSIS

fsa_spell [*options*] [<*infile*] [>*outfile*]

DESCRIPTION

fsa_spell reads lines from the input. Each line contains one word. Every word is looked up in specified dictionaries. If the word is not found, similar words are listed. The degree of similarity is controlled by the edit distance option.

OPTIONS

-d *dictionary*
use that dictionary. Several dictionaries may be given. At least one dictionary must be specified. Dictionaries are automata built using *fsa_build* or *fsa_ubuild.*

-e ***edit_distance*** if the word is not in the dictionaries, look for words that lie within the specified distance. The edit distance is a number of basic editing operations needed to transform one string into another. Those operations are: inserting one character, deleting one character, changing one character, and transposing two adjacent characters.

-i ***input_file*** specifies a file with words to be corrected. More than one file can be specified in this way (i.e. the option can be used more than once). In absence of this option, standard input is used.

-l ***language_file*** specifies a file that holds language specific information, i.e. (for now) characters that form words, and pairs of (lowercase, uppercase) characters for case conversion. If the option is not specified, latin letters with standard case conversions wil be used.

-r ***character_class_file*** specifies a file that information about relation between single characters and two-character sequences. If a two-letter sounds the same as one (different) letter, that relation can be established in this file, and the replacement of the first with the second will be treated as being one edit distance unit apart. The first character of this file is a comment character - all lines beginning with it will be ignored. Data lines consist of two columns. Both columns specify either a single character or a two-character sequence, but the sum of the number of characters should be three. The first column specifies something that may (incorrectly) appear in text, the second one - the correct form.

-v prints version details.

EXIT STATUS

0 OK

1 Invalid option, or lack of a required option.

2 Dictionary file could not be opened.

4 Not enough memory.

SEE ALSO

fsa_ubuild, fsa_build, fsa_accent, fsa_prefix, fsa_morph, fsa_guess, fsa_hash.

BUGS

Send bug reports to the author: Jan Daciuk, jandac@pg.gda.pl.

Appendix B

UTR Package

B.1 tr_ubuild

NAME

tr_ubuild - build a transducer from unsorted data (string pairs)

SYNOPSIS

tr_ubuild [*options*] [<*infile*] [>*outfile*]

DESCRIPTION

tr_ubuild builds a transducer (two-level finite state automaton) from a list of word pairs, where the second word might have annotations. The format of input is:

surface_word lexical_word [annotations]

where *surface_word, lexical_word* and *annotations* are arbitrary strings that do not contain *filler* or *annotation separator.*

The output is a transducer that can be used by *tr_spell, tr_accent, tr_morph,* and *tr_prefix.*

OPTIONS

-O make the resulting transducer smaller. The time required to build the transducer is much greater. How much greater depends on compile options used during compilation of *tr_ubuild.*

-I ***input_ separator*** indicate a character used in input data to separate a surface string from a lexical string, and the lexical string from annotations.

-F ***filler*** indicate a character used to align surface and lexical strings (with annotations). This character may appear in input data (put there by some program), and it is used by *tr_ ubuild* to make the two strings in the transducer have the equal length. The string that is longer is padded with trailing fillers. The default is "_".

-S ***annotation_ separator*** indicate a character to separate annotations from lexical string in *tr_ ubuild* output. The default is "+".

-i ***inputfile*** specifies a file to be used as a source of data for the transducer. If the option is not present, standard input is used instead.

-o ***output_ file*** specifies a file where the transducer should be written. If the option is not present, standard output is used instead.

-v print version details.

EXIT STATUS

0 OK

1 Invalid options, or lack of a required option.

4 Not enough memory.

5 Error in secondary index.

6 Attempt to register already registered node.

7 Attempt to register a node that has never been registered.

13 Too many arcs per node.

SEE ALSO

tr_ build, tr_ spell, tr_ accent, tr_ morph, tr_ prefix

BUGS

Send bug reports to the author: Jan Daciuk, jandac@pg.gda.pl.

For my French data, the transducer built with tr_ubuild is 1 arc bigger than that build with tr_build. There must be a bug somewhere, but it does not affect the programs that use the transducer.

B.2 tr_accent

NAME

tr_accent - find words that deprived of diacritics match words from input

SYNOPSIS

tr_accent [*options*] [<*infile*] [>*outfile*]

DESCRIPTION

tr_accent reads lines from standard input. Each line contains one word. Every word is looked up in specified dictionaries. All words that when deprived of diacritics match the input word are printed.

OPTIONS

-d *dictionary*

use that dictionary. Several dictionaries may be given. At least one dictionary must be specified. Dictionaries are transducers built using *tr_ubuild* or *tr_build.*

-a ***accent_file*** use *accent_file* to establish relationships between characters with diacritics, and those without them. The first character of the *accent_file* is a comment character. All lines that begin with it are comments, and are ignored. All other lines specify one character without diacritics for one or more characters with diacritics. The first character in a line is a character without diacritics. It is followed by one or more spaces and HT characters, and then by a string of characters without diacritics, followed immediately by new line. There should be no spaces between characters with diacritics. For each word containing a character found in the first column in the *accent_file* words containing that character as well as characters listed to the right of it in the *accent_file* will be looked up.

-l *language_file* specifies the name of of file containing language specific information on characters (letters). The first character of *language file* is a comment character; all lines beginning with that character are ignored. The first data line is a sequence of all characters valid in words (default: letters). The second data line contains a sequence of pairs of letters, lowercase letter first, then uppercase, and so on.

If the option is not present, standard case conversions apply.

-i *input_file* specifies input file to be used instead of the standard input.

-v prints version details.

EXIT STATUS

0 OK

1 Invalid option used, or lack of a required option.

2 Dictionary file could not be opened.

4 Not enough memory.

5 Accent file could not be opened or used.

SEE ALSO

tr_ubuild, tr_build, tr_spell, tr_morph, tr_prefix

BUGS

Send bug reports to the author: Jan Daciuk, jandac@pg.gda.pl.

B.3 tr_morph

NAME

tr_morph - print morphology of a word

SYNOPSIS

tr_morph [*options*] [<*infile*] [>*outfile*]

DESCRIPTION

tr_morph reads lines from standard input. Each line contains one word. Every word is looked up in specified dictionaries. For each entry in the dictionary that matches the word, morpological information for that entry is printed.

It is assumed that each dictionary entry can be described as a pair (surface_form, lexical_form+annotations). There can be many surface forms that match the word; in that case all possible lexical forms with annotations are printed. The names *surface form* and *lexical form* reflect their main purpose. They can be used for other purposes as well.

OPTIONS

-d *dictionary*

use that dictionary. Several dictionaries may be given. At least one dictionary must be specified. Dictionaries are transducers built using *tr_ubuild* or *tr_build.*

-l *language_file* specifies the name of of file containing language specific information on characters (letters). The first character of *language file* is a comment character; all lines beginning with that character are ignored. The first data line is a sequence of all characters valid in words (default: letters). The second data line contains a sequence of pairs of letters, lowercase letter first, then uppercase, and so on.

If the option is not present, standard case conversions apply.

-i *input_file* specifies input file to be used instead of the standard input.

-G makes tr_morph generate inflected forms from lexemes and sets of categories (aka tags, or annotations) instead of performing morphological analysis.

-A *annot_separator* specifies the character that separates lexemes from annotations (or tags, i.e. sets of categories) in the input of tr_morph when it is given the -G option. This option has no effect if specified without -G.

-v print version details.

EXIT STATUS

0 OK

1 Invalid options or lack of required options.

2 Dictionary file could not be opened.

4 Not enough memory.

SEE ALSO

tr_ubuild, tr_build, tr_spell, tr_accent, tr_prefix

BUGS

Send bug reports to the author: Jan Daciuk, jandac@pg.gda.pl.

B.4 tr_prefix

NAME

tr_prefix - print all words beginning with the specified prefix

SYNOPSIS

tr_prefix [*options*] [*<infile*] [*>outfile*]

DESCRIPTION

tr_prefix reads lines from standard input. Each line contains one word or nothing. The word is treated as a prefix. All words in the specified dictionaries that have the word as their prefix are printed. The output is in the form:

surface_form lexical_form+annotations

(see *tr_morph).* If the word is empty, the whole contents of the dictionary is printed.

OPTIONS

-d *dictionary*

use that dictionary. Several dictionaries may be given. At least one dictionary must be specified. Dictionaries are transducers built using *tr_ubuild* or *tr_build.*

-l ***language_file*** specifies the name of of file containing language specific information on characters (letters). The first character of *language file* is a comment character; all lines beginning with that character are ignored. The first data line is a sequence of all characters valid in words (default: letters). The second data line contains a sequence of pairs of letters, lowercase letter first, then uppercase, and so on.

If the option is not present, standard case conversions apply.

-i *input_file* specifies input file to be used instead of the standard input.

-v print version details.

EXIT STATUS

0 OK

1 Invalid options, or lack of a required option.

2 Dictionary file could not be opened.

4 Not enough memory.

SEE ALSO

tr_ubuild, tr_build, tr_spell, tr_morph, tr_prefix

BUGS

Send bug reports to the author: Jan Daciuk, jandac@pg.gda.pl.

B.5 tr_spell

NAME

tr_spell - find words in dictionary or propose replacements

SYNOPSIS

tr_spell [*options*] [<*infile*] [>*outfile*]

DESCRIPTION

tr_spell reads lines from standard input. Each line contains one word. Every word is looked up in specified dictionaries. If the word is not found, similar words are listed. The degree of similarity is controlled by the edit distance option.

OPTIONS

-d *dictionary*
use that dictionary. Several dictionaries may be given. At least one dictionary must be specified. Dictionaries are transducers built using *tr_ubuild* or *tr_build.*

-e ***edit_distance*** if the word is not in the dictionaries, look for words that lie within the specified distance. The edit distance is a number of basic editing operations needed to transform one string into another. Those operations are: inserting one character, deleting one character, changing one character, and transposing two adjacent characters.

-l ***language_file*** specifies the name of of file containing language specific information on characters (letters). The first character of *language file* is a comment character; all lines beginning with that character are ignored. The first data line is a sequence of all characters valid in words (default: letters). The second data line contains a sequence of pairs of letters, lowercase letter first, then uppercase, and so on.

If the option is not present, standard case conversions apply.

-i ***input_file*** specifies input file to be used instead of the standard input.

-v prints version details.

EXIT STATUS

0 OK

1 Invalid option, or lack of a required option.

2 Dictionary file could not be opened.

4 Not enough memory.

SEE ALSO

tr_ubuild, tr_build, tr_accent, tr_morph, tr_prefix

BUGS

Send bug reports to the author: Jan Daciuk, jandac@pg.gda.pl.

Bibliography

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[Atw87] Eric Steven Atwell. How to detect grammatical errors in a text without parsing it. In *ACL Proceedings, Third European Conference*, pages 34–45, 1987.

[BE94] Steven Bird and T. Mark Ellison. One-level phonology: Autosegmental representations and rules as finite automata. *Computational Linguistics*, 20(1):55–90, March 1994.

[Ben85] Jon Bentley. A spelling checker. *Communications of the ACM*, 28(5), May 1985.

[Bri93] Eric Brill. *A Corpus-Based Approach to Language Learning.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, USA, 1993.

[Bri95] Eric Brill. Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics*, 21(4):543–565, December 1995.

[BS96] Harald Baayen and Richard Sproat. Estimating lexical priors for low-frequency morphologically ambiguous forms. *Computational Linguistics*, 22(2):155–166, June 1996.

[CDGK92] Jacques Courtin, Danièle Dujardin, Damian Genthial, and Irène Kowarski. Outils lexicaux de l'équipe trilan: bilans et perspectives. In *Séminaire Lexique, PRC Communication Homme-Machine, Pôle Langage Naturelle*, Toulouse, France, January 1992.

[CDK+91] Jacques Courtin, Danièle Dujardin, Irène Kowarski, Damien Gential, and Vera Lucia Strube de Lima. Towards a complete detection/correction system. In *International Conference on Current Issues in Computational Linguistics*, Penang, Malaysia, June 1991.

[CGM93] Jacques Courtin, Damien Genthial, and Jacques Menézo. Intégration de stratégies de correction dans un systemème de détection/correction d'erreurs. In *Colloque Informatique et Langue Naturelle (ILN'93)*, Nantes, France, 1993.

[CH83] Jaime G. Carbonell and Philip J. Hayes. Recovery strategies for parsing extragrammatical language. *American Journal of Computational Linguistics*, 9(3-4):123–146, July-December 1983.

[CKPS92] Doug Cutting, Julian Kupiec, Jan Pedersen, and Penelope Sibun. A practical part-of-speech tagger. In *Proceedings of the Third Conference on Applied Natural Language Processing*, Trento, Italy, 1992. ACL. Also available as Xerox PARC technical report SSL-92-01.

[CT95] Jean-Pierre Chanod and Pasi Tapanainen. Creating a tagset, lexicon and guesser for a french tagger. In *ACL SIGDAT workshop on "From Texts To Tags: Issues in Multilingual Language Analysis"*, pages 58–64, University College Dublin, Ireland, 1995. ACL.

[Dac97] Jan Daciuk. Dopisywanie polskich znaków diakrytycznych za pomocą automatów. In *Krajowe Sympozjum Telekomunikacji '97*, volume D, pages 77–82, Bydgoszcz, September 1997. Akademia Techniczno-Rolnicza w Bydgoszczy, Instytut Telekomunikacji Politechniki Warszawskiej.

[Dam64] F. J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, March 1964.

[DC92] M. W. Du and S. C. Chang. A model and a fast algorithm for multiple errors spelling correction. *Acta Informatics*, 29(3):281–302, 1992.

[DLS83] I. Durham, D. A. Lamb, and J. B. Saxe. Spelling correction in user interfaces. *Communications of the ACM*, 26(10):764–773, October 1983.

[DWW98] Jan Daciuk, Richard E. Watson, and Bruce W. Watson. Incremental construction of acyclic finite-state automata and transducers. In *Finite State Methods in Natural Language Processing*, Bilkent University, Ankara, Turkey, June – July 1998.

[GC92] Damien Genthial and Jacques Courtin. From detection/correction to computer aided writing. In *COLING'92*, Nantes, France, July 1992.

[Gen91] Damien Genthial. Souplesse et robustesse de l'analyse syntaxique: apports d'une hiérarchie de catégories. In *Colloque Informatique et Langue Naturelle (ILN'91)*, Nantes, France, 1991.

[GJ94] Dan Gildea and Dan Jurafsky. Automatic induction of finite state transducers for simple phonological rules. Technical Report TR-94-052, International Computer Science Institute, Berkeley, California, October 1994.

[GLS95] Dennis Grinberg, John Lafferty, and Daniel Sleator. A robust parsing algorithm for link grammars. Technical Report CMU-CS-95-125, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA, August 1995.

[HJM+82] G. E. Heidorn, K. Jensen, L. A. Miller, R. J. Byrd, and M. S. Chodorow. The epistle text-critiquing system. *IBM System Journal*, 21(3):305–326, 1982.

[HU79] John E. Hopcroft and Jefferey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Adison-Wesley Publishing Company, Reading, Massachusets, USA, 1979.

[JHMR83] K. Jensen, G. E. Heidorn, L. A. Miller, and Y. Ravin. Parse fitting and prose fixing: Getting a hold on ill-formedness. *American Journal of Computational Linguistics*, 9(3-4):147–160, July-December 1983.

[Kar94] Lauri Karttunen. Constructing lexical transducers. In *COLING-94*, Kyoto, Japan, 1994.

[KK94] Ronald M. Kaplan and Martin Kay. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378, September 1994.

[KLS93a] Tomasz Kowaltowski, Cláudio L. Lucchesi, and Jorge Stolfi. Application of finite automata in debugging natural language vocabularies. In *First South American String Processing Workshop*, Belo Horizonte, Brasil, 1993.

[KLS93b] Tomasz Kowaltowski, Cláudio L. Lucchesi, and Jorge Stolfi. Minimization of binary automata. In *First South American String Processing Workshop*, Belo Horizonte, Brasil, 1993.

[Kos83] Kimmo Koskenniemi. Two-level model for morphological analysis. In *IJCAI-83*, pages 683–685, Karlsruhe, Germany, 1983.

[Kos84] Kimmo Koskenniemi. A general computational model for word-form recognition and production. In *COLING-84*, pages 178–181, Stanford University, California, USA, 1984. Association for Computational Linguistics.

[KS81] Stan C. Kwasny and Norman K. Sondheimer. Relaxation techniques for parsing grammatically ill-formed input in natural language understanding systems. *American Journal of Computational Linguistics*, 7(2):99–108, April-June 1981.

[Mik96] Andrei Mikheev. Learning part-of-speech guessing rules from lexicon: Extension to non-concatenative operations. In *Coling-96, The 16th Internatinal Conference on Computational Linguistics*, volume 2, pages 770–775, Copenhagen, Denmark, August 5-9 1996.

[Mik97] Andrei Mikheev. Automatic rule induction for unknown-word guessing. *Computational Linguistics*, 23(3):405–423, September 1997.

[Moh94a] Mehryar Mohri. Compact representations by finite-state transducers. In *ACL'94*, San Francisco, California, 1994. Association for Computational Linguistics, Morgan Kaufmann.

[Moh94b] Mehryar Mohri. Syntactic analysis by local grammars automata: an efficient algorithm. In *International Conference on Computational Lexicography (COMPLEX '94)*, Budapest, Hungary, 1994.

[Moh97] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, June 1997.

[Ofl96] Kemal Oflazer. Error-tolerant finite state recognition with applications to morphological analysis and spelling correction. *Computational Linguistics*, 22(1):73–89, March 1996.

[OG94] Kemal Oflazer and Cemalettin Güzey. Spelling correction in agglutinative languages. In *4th Conference on Applied Natural Language Processing*, pages 194–195, Stuttgart, Germany, October 1994.

[Pet80] J. L. Peterson. Computer programs for detecting and correcting spelling errors. *Communications of the ACM*, 23(12):676–687, December 1980.

[PV93] Joseph Pentheroudakis and Lucy Vanderwende. Automatically identifying morphological relations in machine-readable dictionaries. In *Ninth Annual Conference of the UW Centre for the New OED and Text Research*, pages 114–131, 1993.

[PZ84] Joseph J. Pollock and Antonio Zamora. Automatic spelling correction in scientific and scholary text. *Communications of the ACM*, 27(4):358–368, April 1984.

[Rev91] Dominique Revuz. *Dictionnaires et lexiques: méthodes et algorithmes.* PhD thesis, Institut Blaise Pascal, Paris, France, 1991. LITP 91.44.

[Roc95] Emmanuel Roche. Finite-state tools for language processing. In *ACL'95.* Association for Computational Linguistics, 1995. Tutorial.

[RRBP92] Graeme D. Ritchie, Graham J. Russell, Alan W. Black, and Stephen G. Pulman. *Computational Morphology.* Bradford Books. MIT Press, Cambridge, Massachusetts, USA, 1992.

[RS95] Emmanuel Roche and Yves Schabes. Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics*, 21(2):227–253, June 1995.

[RS97] Emmanuel Roche and Yves Schabes. *Finite-State Language Processing.* Bradford Book. MIT Press, Cambridge, Massachusetts, USA, 1997.

[SHS+94] Bruno Maximilian Schulze, Ulrich Heid, Helmut Schmid, Anne Schiller, Mats Rooth, Gregory Grefenstette, Jean Gaschler, Annie Zaenen, and Simone Teufel. Decide. MLAP-Project 93-19 D-1b I, STR and RXRC, November 1994.

[Spr92] Richard Sproat. *Morphology and Computation.* Bradford Books. MIT Press, Cambridge, Massachusetts, USA, 1992.

[TC97] Pieter Theron and Ian Cloete. Automatic acquisition of two-level morphological rules. In *Fifth Conference on Applied Natural Language Processing*, pages 103–110, Washington, DC, USA, April 1997. Association for Computational Linguistics, Association for Computational Linguistics.

[TY79] Robert Endre Tarjan and Andrew Chi-Chih Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, November 1979.

[Ver88] Jean Veronis. Morphosyntactic correction in natural language interfaces. In *international Conference on Computational Linguistics*, pages 708–713. International Comittee on Computational Linguistics, 1988.

[VWOV98] Zygmunt Vetulani, Bogdan Walczak, Tomasz Obrębski, and Grażyna Vetulani. *Unambiguous coding of the inflection of Polish nouns and its application in electronic dictionaries – format POLEX*. Wydawnictwo Naukowe UAM, 1998.

[Wat93a] Bruce W. Watson. A taxonomy of finite automata construction algorithms. Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands, 1993.

[Wat93b] Bruce W. Watson. A taxonomy of finite automata minimization algorithmes. Computing Science Note 93/44, Eindhoven University of Technology, The Netherlands, 1993.

[Wat95] Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Eindhoven University of Technology, the Netherlands, 1995.

[WMS+93] Ralph Weischedel, Marie Meteer, Richard Schwartz, Lance Ramshaw, and Jeff Palmucci. Coping with ambiguity and unknown words through probabilistic models. *Computational Linguistics*, 19(2):359–382, 1993.

[Yar94a] David Yarowsky. A comparison of corpus-based techniques for restoring accents in spanish and french texts. In *2nd Annual Workshop on Very Large Text Corpora*, Kyoto, Japan, 1994.

[Yar94b] David Yarowsky. Decision lists for lexical ambiguity resolution: Application to accent restoration in spanish and french. In *Proceeding of the ACL'94*, San Francisco, California, 1994. Association for Computational Linguistics, Morgan Kaufmann.

Index