Natural Language Engineering **1** (1): 1–18. Printed in the United Kingdom

Efficiently generating correction suggestions for garbled tokens of historical language

Ulrich Reffle
Centrum für Informations- und Sprachverarbeitung
University of Munich

(*Received 2 June 2010*)

Abstract

Text correction systems rely on a core mechanism where suitable correction suggestions for illformed input tokens are generated. Current systems, which are designed for documents including modern language, use some form of approximate search in a given background lexicon. Due to the large amount of spelling variation found in historical documents, special lexica for historical language can only offer restricted coverage. Hence historical language is often described in terms of a matching procedure to be applied to modern words. Given such a procedure and a base lexicon of modern words, the question arises of how to generate correction suggestions for garbled historical variants. In this paper we suggest an efficient algorithm that solves this problem. The algorithm is used for postcorrection of OCR results on historical document collections.

1 Introduction

Text correction systems typically rely on a core step where good correction suggestions are generated for illformed input words. The basis is a background lexicon which is assumed to contain - by and large - all correct words or phrases. Approximate search in the lexicon is then used for finding the correct words which are most similar to the illformed input words. In most applications the background lexicon is very large, which means that efficiency of the proposed methods is a central issue. Today, most competitive methods take advantage of finite state methods to represent the lexicon ((Bunke 1993), (Oflazer 1996)). Recent approaches also use automata to efficiently determine word similarity (Mihov and Schulz 2004).

The background for the work presented in this paper are text correction methods that try to improve optical character recognition (OCR) on historical document collections, preserving the original spelling found in the document. The design of such methods is motivated by recent programs in the field of digital libraries that include historical document collections in their focus such as Google Books (books.google.com), IMPACT (www.impact-project.eu) and Europeana (www.europeana.eu). In this new context it can no longer be assumed that a complete lexicon of all correct words is available. Due to historical language change

and missing normalisation, the vocabulary found in old printed documents is full of distinct spelling variants. This effect becomes more and more central and eye-catching when moving to documents of older periods. Special lexica for historical language can only offer restricted coverage since language variation differs from century to century, from region to region, and even from one document to another.

As an alternative, special matching approaches have been suggested where possible historical spelling variants are derived from modern words in a rule-based way. These approaches make use of rewrite rules or patterns that locally explain the difference between modern and old spelling of a word. Patterns are applied to modern words to produce distinct historical spellings. The complete historical vocabulary is then described in terms of a modern lexicon of the given language and a characteristic set of patterns to be applied.

In this paper we suggest an algorithm for generating correction suggestions for misrecognized words of historical documents based on such a rule-based model for historical language. The lexicon of modern words and the set of patterns for a historical language are used as background resources. Given an illformed token as input and a small bound k, the algorithm efficiently generates all historical spelling variants of modern words where the Levenshtein distance between the input token and the variant does not exceed k. The algorithm is used as a core component of a system for postcorrection of OCR results on historical document collections to be built by our group in the EU project IMPACT.

The paper is structured as follows. After some formal preliminaries in Section 2 we introduce the exact algorithmic problem considered in the paper in Section 3. We show how our background scenario can be described in terms of a model with two channels. The first channel accounts for the distinction between modern and historical spellings of words. The second channel introduces errors via optical character recognition. Receiving the result of the second channel as input, we want to predict the underlying output of the first channel.

Section 4 shows how to efficiently enumerate all spelling variants of modern words that can be derived using patterns. The procedure, which is based on finite-state techniques, uses the lexicon of modern words and the set of patterns as input. It is flexible in the sense that distinct lexica and sets of patterns can be used at low computational costs. This is interesting in application scenarios where documents from distinct historical periods and/or regions are processed, each coming with specific characteristic sets of patterns.

In Section 5 we show how universal Levenshtein automata (Mihov and Schulz 2004) can be used to control, given an illformed token w as input, the generation of spelling variants in a way that only good correction suggestions for w are produced. At the same time, this form of control drastically improves efficiency since the generation of useless variants is avoided.

The evaluation results presented in Section 6 show that our algorithm is highly efficient. We used a lexicon for modern German language with 2,336,165 entries and a pattern set for German historical language including 140 patterns. The search time depends on the bound for the maximal Levenhtein distance k between the input

word and the correction suggestions. For $k = 0, 1, 2$ the generation of all suggestions approximately takes 0.2 ms, 0.9 ms, and 17 ms respectively.

We finish with a description of related work in Section 7 and a brief conclusion.

2 Formal preliminaries and terminology

We consider words composed over a finite alphabet Σ called the *text alphabet*. The length of a word $w \in \Sigma^*$ is written $|w|$. Concatenation of words $v, w \in \Sigma^*$ is written $v \circ w$ or vw. For sets of words $V, W \subseteq \Sigma^*$ let $V \circ W = \{vw \mid v \in V, w \in W\}$. Finite state automata (FSA) and deterministic FSA over the text alphabet Σ are defined as usual (see, e.g., (Roche and Schabes 1997)). Deterministic FSA are formally presented as quintuples $A = (\Sigma, Q, \delta, s, F)$ where Q (F) denotes the set of (final) states, s is the initial state and $\delta : Q \times \Sigma \to Q$ is the transition function. By $\delta^* : Q \times \Sigma^* \to Q$ we denote the generalized transition function. For convenience we repeat the notion of a path, which is needed later.

Definition 2.1
Let $A = (\Sigma, Q, \delta, s, F)$ be a deterministic FSA. A *path* of A of length l from $q_0 \in Q$ to $q_l \in Q$ is a sequence

$$\pi = (q_0, \sigma_0, q_1, \sigma_1, \ldots, q_{l-1}, \sigma_{l-1}, q_l)$$

where $q_i \in Q$ $(0 \leq i \leq l)$, $\sigma_i \in \Sigma$ and $\delta(q_i, \sigma_i) = q_{i+1}$ $(0 \leq i \leq l-1)$. The path π has the *label* $\text{label}(\pi) := \sigma_0 \cdots \sigma_{l-1}$. Two paths π_1 and π_2 of A can be concatenated (in the obvious way) if the target state of π_1 is equal to the initial state of π_2. We write $\pi_1 \circ \pi_2$. Obviously $\text{label}(\pi_1 \circ \pi_2) = \text{label}(\pi_1) \circ \text{label}(\pi_2)$.

In what follows we assume that a finite set $D \subseteq \Sigma^*$ is given. In our application scenarios, D represents the set of words in a lexicon for a given modern language.

Definition 2.2
A *pattern* is a simple rewrite rule composed of two non-empty strings α, β over Σ written in the form $\alpha \mapsto \beta$.

Using a pattern $\alpha \mapsto \beta$ a string w of the form $x\alpha y$ can be rewritten to $v = x\beta y$.

Definition 2.3
Let P denote a set of patterns, let $w \in D$, let $v \in \Sigma^*$. If for some $n \geq 0$ the word w can be written in the form $w_0\alpha_1 w_1 \alpha_2 ... \alpha_n w_n$ $(w_i \in \Sigma^*)$ for patterns $(\alpha_i \mapsto \beta_i) \in P$, then $v = w_0\beta_1 w_1 \beta_2 ... \beta_n w_n$ is called a *spelling variant* of w. The *hypothetical lexicon* determined by D and P is the set of all spelling variants of words in D.

Note that for the derivation of spelling variants patterns are applied in parallel and left-hand sides of patterns in words w must not overlap.

We are often interested in keeping track of the positions where patterns are applied when rewriting a modern word $w = w_0\alpha_1 w_1\alpha_2 ...\alpha_n w_n$ to a spelling variant v.

Definition 2.4
Let w, v as above, let $j_i = |w_0\alpha_1..w_{i-1}|$. Assuming that j_i is the position where pattern $\alpha_i \mapsto \beta_i$ is applied, we call $\tau_P = \langle \alpha_1 \mapsto \beta_1, j_1\rangle, .., \langle \alpha_n \mapsto \beta_n, j_n\rangle$ the *pattern trace* of the derivation of v from w.

We write $w \xrightarrow{\tau_P} v$ to indicate that the spelling variant v is derived from w using pattern trace τ_P.

The *Levenshtein distance* between two words $v, w \in \Sigma^*$ is the minimal number of single-letter deletions, insertions or substitution that are needed to transform v into w. Single-letter deletions, insertions and substitution are called *edit operations*. We write $d_L(v, w)$ for the Levenshtein distance of the strings $v, w \in \Sigma^*$. By a *Levenshtein trace* we mean a description τ_L which states which kind of edit operations are applied at which positions of v to produce w. We only consider Levenshtein traces that use a minimal number of edit operations.

3 Background scenario and algorithmic problem

Let D denote a lexicon of modern words, let P denote a set of patterns. We consider a scenario where words $u \in D$ are garbled following a two-channel model. The first channel ("spelling variation") transforms u to a spelling variant v of u using patterns in P. The exact transformation is decribed in terms of a pattern trace τ_P. The second channel ("error channel") introduces errors by applying edit operations, which results in a possibly erroneous version w of v. The latter process is described in terms of a Levenshtein trace τ_L. We write

$$u \xrightarrow{\tau_P} v \xrightarrow{\tau_L} w$$

to capture the complete process. An important practical instance of this scenario is the situation where we analyze the result of applying OCR to historical documents. The first channel describes the distinction between modern and historical spelling. The second channel takes recognition errors into account.

In what follows, a quintuple of the form

$$u \xrightarrow{\tau_P} v \xrightarrow{\tau_L} w$$

is called an *interpretation* of w. In practice we are just given w as input. In this paper we study the following

Algorithmic Problem: *Assume we are given a lexicon of modern words, D, a set of patterns P and a small bound k for the Levenshtein distance. For an input word $w \in \Sigma^*$, compute all interpretations $u \xrightarrow{\tau_P} v \xrightarrow{\tau_L} w$ where $u \in D$ and $d_L(v, w) \leq k$. The trace τ_L must be minimal in terms of the number of edit operations used.*

As a simplification we consider the situation where we are not interested in u, τ_P and τ_L and just want to compute, given $w \in \Sigma^*$, the set of spelling variants v where $d_L(v, w) \leq k$ (**Simplified Algorithmic Problem**).

A solution for the simplified algorithmic problem helps to compute a set of correction suggestions for garbled tokens w. A solution for the full algorithmic problem in addition helps to rank correction suggestions (e.g. when using weighted patterns and/or edit operations) and to collect statistical evidence for properties of the two channels.

4 Representing and traversing hypothetical lexica

Assuming that the lexicon of modern words D and the set of patterns P are given we first look at the simplified algorithmic problem. This section presents a preparatory step, the focus is on the first channel (spelling variation). We introduce a method for enumerating all entries of a hypothetical lexicon. The main advantage of the method is its flexibility. The lexicon of modern words D and the set of patterns P are inputs to the method and can be changed in a simple manner at low computational costs. We avoid an explicit representation of the full hypothetical lexicon, which might lead to storage problems, see Remark 4.5 below. The three subsections introduce a product construction which leads to a "virtual representation" of the hypothetical lexicon that can be used for enumerating all spelling variants. In the following section (Section 5) we will then combine the virtual representation of the hypothetical lexicon with techniques from (Mihov and Schulz 2004) to solve the simplified algorithmic problem defined in Section 3. We will then see that only a small part of the hypothetical lexicon is visited in the complete procedure.

4.1 Virtual paths of the lexicon automaton

In the following we assume that the base lexicon D is given in the form of a deterministic FSA $A_D = (\Sigma, Q_D, \delta_D, s_D, F_D)$. Figure 1 illustrates how A_D can be extended to a non-deterministic FSA A_H that represents the hypothetical lexicon. The set of states of A_H is equal to the set of states of A_D. All transitions of A_D are also transitions of A_H. The needed additional transitions are captured by the relation

$$\Delta_{\mathcal{H}} = \{(q, \beta, q') \mid \delta_D^*(q, \alpha) = q', (\alpha \mapsto \beta) \in P\}.$$

It is trivial to see that for each word w in D with one or several occurrences of left-hand sides of patterns in P the new automaton A_H recognizes the variant obtained from a parallel application of the rewrite patterns. Conversely, each word v accepted by A_H can be rewritten into a word accepted by A_D by a reverse parallel application of patterns, hence v is in fact a spelling variant.

As we mentioned before, we want to avoid the explicit computation of A_H. In our approach the transitions in $\Delta_{\mathcal{H}}$, which are called *virtual transitions*, are not explicitly added to A_D. In a way to be explained below, virtual transitions are simulated by a special mechanism that is used for traversing A_D.

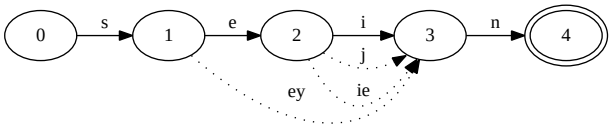

Fig. 1. Additional transitions to represent hypothetical lexica. Here three patterns are considered: $ei \mapsto ey$, $i \mapsto j$, $i \mapsto ie$

A *virtual path* is obtained from a path π in A_D by replacing (zero or some) subpaths $\pi' = (p, \ldots, q)$ of π where the label of π' represents the left-hand side α of a pattern $\alpha \to \beta \in P$ by a a single virtual transition (p, β, q). Replaced subpaths must not overlap. The new path $\hat{\pi}$ derived from π is called a *variant* of path π. The label of $\hat{\pi}$ is called a *variant* of the label of π.

It is important to draw the analogy between these definitions and those in Section 2. When looking at the corresponding labels, the substitution of subpaths of a path π of A_D by virtual transitions is equivalent to applying patterns to the label of π. Hence the labels of all variants of a path π exactly describe the set of all spelling variants of the label of π.

4.2 Generating all spelling variants

In the following we present an algorithm that enumerates all spelling variants, traversing the lexicon automaton A_D in a depth first manner. Let us briefly recall a standard procedure for completely traversing an acyclic deterministic FSA A with initial state s and transition function δ. The procedure is based on a single stack Γ. Entries of Γ are pairs of the form (q, u) where q is a state of A and $u \in \Sigma^*$ is the label of a path from s to q. At the beginning, (s, ϵ) is pushed on the stack[1] (Initialisation). As long as Γ is not empty, we treat the top element (q, u) of Γ by applying three steps:

1. If q is final, then we output u.
2. For all letters $\sigma \in \Sigma$ where $\delta(q, \sigma)$ is defined we recursively call the procedure after pushing $(\delta(q, \sigma), u\sigma)$ on top of Γ.
3. We pop (q, u) from Γ.

It is simple to see that the output is the set of words accepted by the automaton.

In our context we modify the procedure. Entries of the stack Γ now have the form (q, u, V) where q, u are as before and $V \subseteq \Sigma^*$ is the set of all variants of u. The computation of the sets V is based on an inductive definition which is found in the following description.

[1] Here ϵ denotes the empty word.

Basic Procedure: At the beginning, $(s_D, \epsilon, \{\epsilon\})$ is pushed on the stack Γ (Initialisation). As long as Γ is not empty, we treat the top element (q, u, V) of Γ applying three steps:

1. If q is final, then we output all elements of V.
2. For all letters $\sigma \in \Sigma$ where $\delta_D(q, \sigma)$ is defined we recursively call the procedure after pushing $(\delta_D(q, \sigma), u\sigma, V')$ on top of Γ. Here V' contains $V \circ \{\sigma\}$ and in addition all strings of the form $v_1 \circ \beta$ where for some pattern $(\alpha \to \beta) \in P$ the string $u\sigma$ can be represented in the form $u_1 \circ \alpha$ such that v_1 is a variant of u_1.
3. We pop (q, u, V) from Γ.

In Step 2, the additional strings take into account the situation where $u\sigma$ has a suffix α representing the left-hand side of a pattern $(\alpha \to \beta) \in P$. It is important to note that there can be more than one pattern with left hand side α. Also, $u\sigma$ can have suffixes of different lengths matching a left hand side of a pattern.

For computing V', the central problem is to efficiently recognize the suffixes α of $u\sigma = u_1 \circ \alpha$ that represent left hand sides of patterns in P. In fact, once α and thus u_1 and β are known, we go back to the unique element (p, u_1, V_1) of the stack determined by u_1. Here V_1 lists all variants v_1 of u_1. We use V_1 to compute the additional variants of the form $v_1\beta$ of V'. The efficient recognition of all suffixes α is described in the next subsection. Some further remarks are in order.

Remark 4.1
In the Basic Procedure, the cardinality of the set V can grow exponentially in the length of the string u. In the next section, where we present the full solution to our algorithmic problem, we shall see that this growth is avoided in practice due to an additional control mechanism.

Remark 4.2
The Basic Procedure is used for solving the simplified algorithmic problem, in a way to be explained in the next section. A straightforward modification can be used for solving the full algorithmic problem. We not only store the set of variants V of the label u of the current path but store for each $v \in V$ also the pattern trace which encodes the transformation of u into v. Each spelling variant in the output set then comes with a pattern trace.

Remark 4.3
The Basic Procedure may generate duplicates in the output. When we solve the general algorithmic problem and take pattern traces into account, then there are no duplicates in the output.

4.3 A finite-state data structure to represent the pattern set

In the above description of the Basic Procedure, the efficient recognition of situations where a suffix of the current path can be replaced by a virtual transition was left as an open issue. The solution presented in this section treats this problem

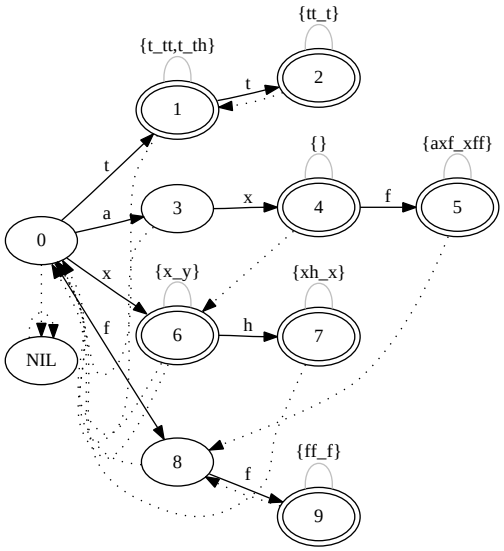

Fig. 2. Aho-Corasick trie for
$P = \{\mathrm{t} \mapsto \mathrm{tt}, \mathrm{t} \mapsto \mathrm{th}, \mathrm{tt} \mapsto \mathrm{t}, \mathrm{ff} \mapsto \mathrm{f}, \mathrm{axf} \mapsto \mathrm{xff}, \mathrm{x} \mapsto \mathrm{y}, \mathrm{xh} \mapsto \mathrm{x}\}$

as a special variant of multiple string search. In our case, the set of search strings is the set of all left pattern sides $L(P) = \{\alpha \mid (\alpha \mapsto \beta) \in P\}$, and we want to find these search strings on the fly as suffixes of u, the label of the current path reached in the lexicon automaton A_D. This means that, other than in a traditional string search application, the search string u is not a static sequence of symbols that can be processed from left to right, but a string dynamically changing with the backtracking procedure.

One classical method for multiple string search that suits to this special application in a natural way is the Aho-Corasick Algorithm (Aho and Corasick 1975). In this algorithm, the set S of search strings is compiled into a trie T_S which is enriched in a specific way. When searching strings of S in an input text, the current status of the search is encoded in a single *active state* of T_S. The following description of the search method, where we use the set $L(P)$ as our set of search strings and replace the input text by the lexicon automaton A_D, will make clear how this active state can be stored and updated in parallel with each single step of the traversal of A_D.

Data structure. In a pre-processing step a structure

$$AC = (\Sigma, Q_{AC}, \delta_{AC}, s_{AC}, F_{AC}, f_{AC}, o_{AC})$$

is computed. $\Sigma, Q_{AC}, \delta_{AC}$ and s_{AC} represent the usual components of a trie automaton which recognizes exactly the strings in $L(P)$ (see Figure 2 for an illustration). The set of final states F_{AC} is defined below.

Apart from the classic components of a deterministic FSA, the so-called *failure*

function f_{AC} and the *output function* o_{AC} are defined. The failure function is used as an ingredient to define the set of final states.

Following (Aho and Corasick 1975) we say that a string u *represents* state q ($u \overset{R}{\sim} q$) if the path from s_{AC} to q has label u. Recall that the correlation between an arbitrary state q and the path leading from the start state to q is always non-ambiguous in a trie structure.

The *failure function* $f_{AC} : Q_{AC} \mapsto Q_{AC}$ maps each state $q \in Q_{AC}$ ($s \overset{R}{\sim} q$) to the unique state q' ($s' \overset{R}{\sim} q'$) where s' is the longest proper suffix of s that is also a prefix of some search string in $L(P)$.

The set of *final states* F_{AC} contains all states represented by a search string in $L(P)$ and is closed under iterative applications of the following rule: a state q is final if $f_{AC}(q)$ is final.

The *output function* $o_{AC} : Q_{AC} \mapsto 2^P$ maps each final state $q \in F_{AC}$ to the subset of patterns ($\alpha \mapsto \beta$) where $\alpha \overset{R}{\sim} q$. Here α is unique, but P can contain several patterns with left-hand side α.

Using AC in the Basic Procedure. Consider a traversal of A_D, let (q, u, V) denote the top entry of the stack Γ, let s denote the longest suffix of u that is also a prefix of some search string in $L(P)$. The suffix s can be seen as the most promising beginning of a match. With the element (q, u, V) on the stack Γ we associate the state $q^u_{AC} \in Q_{AC}$ represented by s as the current active state of AC.

The actual computation of the active state proceeds as follows. For initialisation we define $q^{\epsilon}_{AC} := s_{AC}$. Assume we have found q^u_{AC} which is represented by s. If the traversal procedure then pushes $(\delta_D(q, \sigma), u\sigma, V')$ as new top element of the stack, then $q^{u\sigma}_{AC}$ is computed in the following way:

- If $\delta_{AC}(q^u_{AC}, \sigma)$ is defined, then we have $q^{u\sigma}_{AC} = \delta_{AC}(q^u_{AC}, \sigma)$. In this situation $s\sigma$ is a prefix of a search string in $L(P)$.
- If, on the other hand, $\delta_{AC}(q^u_{AC}, \sigma)$ is undefined (the current longest prefix s can not be extended with σ), then we apply f_{AC} starting from q^u_{AC} in an iterated manner until we
  1. either reach a state p with an outgoing σ-transition (possibly $p = s_{AC}$), or
  2. arrive at the initial state s_{AC}, which in this subcase does not have a σ-transition.

  In Case 1 we define $q^{u\sigma}_{AC} := \delta_{AC}(p, \sigma)$, in Case 2 we define $q^{u\sigma}_{AC} := s_{AC}$.

Readers familiar with the Aho-Corasick algorithm will note that this is the classical procedure for updating the active state in AC.

Example 4.4
As an illustration consider the AC trie depicted in Figure 2. Here we have $q^{ta}_{AC} = 3$, because "a" (which represents State 3) is the longest prefix of all search strings that is also a suffix of *ta*. When adding character "x" to "ta", it is easy to extend the longest prefix and the updated position in AC, simply by following the "x"-transition to state $q^{tax}_{AC} = 4$. On the other hand, when moving to "taf" we have to

follow the failure transition from State 3. Since State 0 has the f-successor 8 we have $q_{AC}^{taf} = 8$.

Recognizing left pattern sides. If $q_{AC}^{u} \in F_{AC}$, this indicates that some search string was found as a suffix of u. Recall that not only all states q which are represented by some k are final, but also those states q where $f_{AC}(q) \in F_{AC}$. For example, State 4 in Figure 2 is final although "ax" is not a search string - but the failure function points to "x" which is the matching search string. Also, "tt" representing the final State 2 is a search string, but following the failure transition to State 1 reveals another, shorter search string "t" that matches at the same position. At any position $q = q_{AC}^{u} \in F_{AC}$, the set P_{match} of patterns $\alpha \to \beta$ where α is a suffix of u can be computed using this algorithm:

1. Add all elements of $o_{AC}(q)$ to P_{match}.
2. Apply f_{AC} to q: $q := f_{AC}(q)$.
3. Repeat Steps 1 and 2 as long as q is a final state of AC.

The details of how the trie structure and the failure and output function are computed can be read in (Aho and Corasick 1975), see also (Navarro and Raffinot 2001). The slight differences between those descriptions and the method as described here (e.g. the output function returning patterns instead of matched strings) are all obvious, so the construction algorithm is not repeated here.

Remark 4.5
At the end of this section let us add a comment. Instead of using the above virtual representation of the hypothetical lexicon in terms of the lexicon automaton and the Aho-Corasick trie the reader might suggest to compute the full hypothetical lexicon in an offline step and to compile it into a minimal deterministic FSA. However, there are two problems with this idea, memory space and flexibility. We briefly comment on these points.

(1) Memory space. In practice, hypothetical lexica grow very large, and so do their representations as deterministic FSA. Table 1 shows the sizes of hypothetical lexica as they are used in various applications for the handling of historical German language. The base lexicon of modern words contains 2,336,165 inflected forms, the rule set consists of 140 patterns. We ignored the need to store the traces with each variant and built minimized deterministic FSAs, restricting ourselves to spelling variants involving not more than 1 (2, 3, respectively) pattern applications. Even with only one pattern application, more than 114 million spelling variants had to be stored, using 124MB of disk space. For up to three pattern applications, almost 50 billion variants were produced, requiring more than 1,000 MB of memory. The storage of the respective trace with each spelling variant (in order to solve the full algorithmic problem) will be even more problematic.

(2) Flexibility. Regardless of the question if such memory requirements are acceptable, or if a non-deterministic variant of such an automaton could be represented in a more compact way, a second good argument not to preprocess hypothetical lexica is flexibility. If we assume that such data structures can usually not be created at

Table 1. *Sizes of explicitly built minimal deterministic FSAs of hypothetical lexica on the basis of a lexicon containing 2,336,165 German full forms and a rule set of 140 patterns describing historical spelling variation. The application of patterns to each word was restricted to 1,2,3 patterns respectively.*

patterns	entries	states	size
1	114,387,908	2,539,169	124 MB
2	2,915,182,510	6,529,919	440 MB
3	49,180,654,333	12,469,981	1,054 MB

runtime, then the parameters for their construction - base lexicon, set of patterns and an upper bound for pattern applications - have to be determined beforehands. Distinct automata will have to be created for each required combination of these parameters.

5 Approximate matching in the hypothetical lexicon

In this section we add to the Basic Procedure described in Section 4 the last missing component that is needed to solve the algorithmic problem introduced in Section 3. We first show how to solve the simplified algorithmic problem and then comment on the modifications that are necessary to solve the full problem.

Given the lexicon of modern words D, the set of patterns P, a bound k, and an input string w we want to find all entries of D such $d_L(v, w) \leq k$.

In (Mihov and Schulz 2004) the simpler problem was considered of how to find all entries u of the lexicon D such that $d_L(u, w) \leq k$. We briefly recall this procedure, which can be adapted for our purposes. The approach in (Mihov and Schulz 2004) is based on the concept of a *universal Levenshtein automaton*. A universal Levenshtein automaton $\forall_k$ of degree k can be used to decide for any pair of words u, w if $d_L(u, w) \leq k$. The letters σ of u are translated into bitvectors χ_σ of restricted length (depending on k) which encode the distribution of occurrences of σ in subwords of w. For the exact encoding we refer to (Mihov and Schulz 2004). In what follows we write $\chi_{\sigma_1 \ldots \sigma_n}$ for a sequence $\chi_{\sigma_1} \ldots \chi_{\sigma_n}$. The sequence of bitvectors χ_u obtained from the letters of u is used as the input of $\forall_k$. It is accepted iff $d_L(u, w) \leq k$. In our context, one important aspect is that in the negative case, when the Levenshtein distance between u and w is too large, the traversal in $\forall_k$ often fails already after consuming some (translated) letters of u. In fact, the automaton $\forall_k$ represents a device that - when reading χ_u - at the earliest possible point detects that $d_L(u, w) > k$.

In (Mihov and Schulz 2004) the lexicon D is encoded as a deterministic FSA A_D. To find all entries u of D such that $d_L(u, w) \leq k$ a complete traversal of A_D is started. Each transition label σ represents a letter of a candidate word u to be found in D. Transition labels σ are translated into bitvectors representing occurrences of

σ in subwords of w as above. In this way the traveral of A_D is controlled using a parallel traversal of $\forall_k$ with translated labels as input. When we reach a final state both in A_D and in $\forall_k$ on a path with label u we know that $d_L(u, w) \leq k$. The property mentioned above guarantees that in most paths we soon reach a point where the traversal in $\forall_k$ can not be continued. Hence only a small part of A_D is visited in practice.

This idea can be generalized to control the generation of spelling variants in the Basic Procedure. Recall that in this procedure triples (q, u, V) store with each prefix u of a word in A_D the set V of variants of u. For each variant $v \in V$ we now maintain a parallel run of $\forall_k$ using χ_v as input. At each step of the traversal we store for each $v \in V$ the state of $\forall_k$ reached with input χ_v. If for a variant v the parallel traversal of $\forall_k$ leads to failure we delete it. In what follows, all other variants are called "promising". In the new procedure, instead of storing *all* variants of u the triples (q, u, V) now only store "promising" variants. In practice, small bounds k are used and the set of promising variants remains small (cf. Remark 4.1). When we reach a final state in A_D we check for each $v \in V$ if the the parallel run in $\forall_k$ has also lead to a final state. In the positive case we output v.

It should be noted that we *cannot* immediately start backtracking as soon as the set of promising variants V of the top triple of the stack is empty. Recall that in the basic procedure, when we add $(\delta_D(q, \sigma), u\sigma, V')$ on top of (q, u, V), the new set of variants V' contains not only $V \circ \{\sigma\}$ but in addition all new variants of the form $v_1\beta$ where $u\sigma$ can be represented in the form $u_1\alpha$ for $\alpha \to \beta \in P$ where v_1 is a variant of u_1. This step may introduce new promising variants. However, backtracking may start as soon as the number of triples at the top of stack Γ with empty set of promising variants exceeds the maximal length l_{max} of a left-hand side of a pattern in P. In this situation, new variants $v_1\beta$ can not be promising since v_1 is not promising. We may now describe our procedure for solving the simplified algorithmic problem.

Main Procedure: At the beginning, $(s_D, \epsilon, \{\epsilon\})$ is pushed on the stack Γ (Initialisation). As long as Γ is not empty, we treat the top element (q, u, V) of Γ. If the number of top triples in the stack with empty third component is larger than l_{max} we immediately pop (q, u, V) from stack Γ. Otherwise we apply three steps:

1. If q is final, then we output all elements v of V where we have reached a final state in the parallel traversal of $\forall_k$.

2. For all letters $\sigma \in \Sigma$ where $\delta_D(q, \sigma)$ is defined we recursively call the procedure after pushing $(\delta_D(q, \sigma), u\sigma, V')$ on top of Γ. Here V' contains all promising strings of the form $v\sigma$ for $v \in V$ and in addition all promising strings of the form $v_1 \circ \beta$ where for some pattern $(\alpha \to \beta) \in P$ the string $u\sigma$ can be represented in the form $u_1 \circ \alpha$ such that v_1 is a promising variant of u_1.

3. We pop (q, u, V) from Γ.

In Clause 2 we assume that the states reached in $\forall_k$ with input χ_{v_1} are stored[2], we only have to continue the run of $\forall_k$ with new input χ_β.

Remark 5.1
It is straightforward but technically tedious to extend the Main Procedure in a way that it solves the full algorithmic problem, that is, to provide with each matching variant v also the respective modern word u as well as the pattern trace τ_P and Levenshtein trace τ_L. The modern word u is provided by the Main Procedure directly. As to τ_P, we mentioned in Remark 4.2 that it is simple to store with each generated variant v its pattern trace. For all matches v of the input token w, the Levenshtein trace τ_L can be computed using the well-known algorithm based on dynamic programming. An extended output procedure then yields the desired result.

In practical applications it is often useful to specify an upper bound of spelling variant patterns applied to one word. If the respective pattern traces are stored with each $v \in V$, it is again straightforward to limit the set of promising variants in that sense.

6 Evaluation

Experiments were made using a realistic setup for the processing of historical German language. We solved the full algorithmic problem. The lexicon D_{GER} contains $2,336,165$ modern German full forms. P_{GER} is a set of 140 patterns modelling orthographical spelling variation in historical German texts. The rules were selected manually by linguists.

From a large corpus of German texts from the seventeenth to the nineteenth century we extracted the list of all words (more than 200,000). From this list we randomly chose words of distinct lengths l ($6 \leq l \leq 14$) and introduced, again randomly, k ($0 \leq k \leq 2$) symbol substitutions, deletions or insertions. The result are sets $T_{k,l}$ of test queries, each containing 1,000 tokens with the specified length and number of errors.

Table 2 shows the average time needed to compute all correction suggestions (in milliseconds) and the average number of correction suggestions obtained for all test sets $T_{k,l}$. The bound for the maximal Levenshtein distance between input token and correction suggestion was set to k for each test set.

In a second series of tests we restricted the number of patterns that are applied when deriving a historical spelling variant v from a modern word u. The maximal number of applications, denoted maxPat, was set to $0, 1, 2, 3$, and ∞. We prepared new test sets $T_{k,\text{-}}$ ($0 \leq k \leq 2$), each containing 1,000 tokens of varying length. Again we garbled the tokens of the sets, randomly introducing k errors ($k = 0, 1, 2$).

Table 3 describes the average number of correction suggestions and the average processing time obtained for these test sets and bounds.

[2] To this end, triples (q, u, V) are replaced by a more complex representation, the obvious details are omitted.

Table 2. *Average number of correction suggestions and time needed to compute all correction suggestions (in ms) for different bounds $k = 0, 1, 2$ of the maximal Levenshtein distance between input w and correction suggestion v. Queries have length 6-14 and contain k errors.*

length	$k = 0$		$k = 1$		$k = 2$	
	cands	time	cands	time	cands	time
6	1.519	0.202	41.628	4.058	1543.960	59.892
7	1.044	0.204	24.939	4.061	750.599	55.799
8	0.975	0.221	16.222	4.134	415.087	54.457
9	0.922	0.217	11.634	4.186	245.460	54.514
10	0.844	0.216	8.246	4.271	150.838	53.684
11	0.770	0.216	8.075	4.379	92.601	54.288
12	0.704	0.223	6.794	4.400	60.962	55.019
13	0.688	0.226	4.606	4.348	52.900	55.231
14	0.625	0.227	4.518	4.505	37.097	55.283

Table 3. *Average number of correction suggestions and time needed to compute all correction suggestions (in ms) for different bounds $k = 0, 1, 2$ of the maximal Levenshtein distance between input w and correction suggestion v. Correction suggestions are derived from modern words applying maxPat patterns at most.*

k_{err}	$maxPat = 0$		$maxPat = 1$		$maxPat = 2$		$maxPat = 3$		$maxPat = \infty$	
	cands	time	cands	time	cands	time	cands	time	cands	time
0	0.506	0.102	0.758	0.179	0.882	0.205	0.914	0.212	0.917	0.212
1	1.376	1.282	7.002	2.713	13.723	3.760	17.315	4.187	18.455	4.235
2	16.562	8.365	124.542	20.302	347.983	36.710	540.134	48.619	625.798	53.325

6.1 Using backwards dictionaries for filtering

For conventional approximate search in dictionaries, (Mihov and Schulz 2004) achieve considerable improvements by adapting a filter method known from approximate text search (WM92). The central idea of the adaptation is to avoid approximate search with high distance bounds in the front part of the lexicon automaton where it is usually densely branched. To this end, the search string is divided into two halves of approximately equal length: $w = w_1 \circ w_2$. Then, approximate search with distance bound k can be split up into several cases considering all possible distributions of up to k errors among the two halves.

For example, for $k = 2$, the following disjoint cases have to be considered:

1. No error in w_1, up to 2 errors in w_2.
2. 1 error in w_1, 0 or 1 error in w_2.
3. 2 errors in w_1, no errors in w_2.

(Mihov and Schulz 2004) explain in detail how the search first for w_1 and subsequently for w_2 can be organized using different distance bounds. For Case 1, as an

Table 4. *Repetition of experiments from Table 2 using the filtering method with backwards dictionaries. Times in milliseconds and speed-up factors (ratio of times) w.r.t Table 2.*

length	$k = 1$		$k = 2$	
	time	speed-up	time	speed-up
6	1.314	3.088	28.362	2.112
7	1.070	3.795	23.949	2.330
8	0.945	4.375	14.194	3.837
9	0.798	5.246	13.466	4.048
10	0.770	5.547	9.169	5.855
11	1.102	3.974	8.910	6.093
12	0.726	6.061	7.202	7.639
13	0.660	6.588	7.248	7.620
14	0.663	6.795	6.439	8.586

Table 5. *Repetition of experiments from Table 3 using the filtering method with backwards dictionaries. Times in milliseconds and speed-up factors (ratio of times) w.r.t Table 3.*

k_{err}	$maxPat = 0$		$maxPat = 1$		$maxPat = 2$		$maxPat = 3$		$maxPat = \infty$	
	time	speed-up	time	speed-up	time	speed-up	time	speed-up	time	speed-up
1	0.277	4.628	0.563	4.819	0.774	4.858	0.876	4.780	0.915	4.628
2	2.136	3.916	5.837	3.478	10.794	3.401	15.151	3.209	17.247	3.092

example, the traversal is started accepting only exact matches for w_1. The much more expensive error-tolerant search only starts at points where w_1 was already found and the search space is narrowed down considerably. To produce the same effect for Case 3, where both errors are to be found in the initial half of the search string, reversed strings w_1^{-1} and w_2^{-1} are used as search strings and a *backwards lexicon* A_D^{-1} containing all reversed strings of D: the search in A_D^{-1} can now begin with w_2^{-1}, not allowing any errors, and continue with distance bound 2.

This same filtering technique can be applied directly to the algorithm presented in this paper, taking some extra care where a pattern application affects both w_1 and w_2. Tables 4 and 5 show that the efficiency of our method is improved significantly using the filter: the method is about two times faster for very short words, and up two 8.5 times faster for very long words. Table 5 shows that for words of different lengths, distributed according to realistic text material, speed-up factors range from 3 to 4.8. Here, the processing time for one query with distance bound $k = 1$ does not exceed 1 millisecond.

7 Related work

Rule-based models for historical spelling variation have been proposed by several groups to enhance information retrieval on historical text collections ((Pilz et al. 2005), (Hauser et al. 2006), (Archer et al. 2006), (Ernst-Gerlach and Fuhr 2006), (Ernst-Gerlach and Fuhr 2007)). The references also describe different approaches on how to obtain good rule sets for historical spelling variation. In (Brill and Moore 2000) a method is presented to learn good rewrite rules for noisy channels from example material.

(Gotscharek et al. 2009a) and (Gotscharek et al. 2009b) illuminate which lexical resources and rule-based methods are suitable for processing historical German language of different time periods. Experiments show that information retrieval and OCR results can be improved considerably by applying suitable lexica - especially for eighteenth and nineteenth century material, hypothetical lexica help to obtain the best results.

For approximate matching in lexica based on similarity measures, several solutions have been proposed. Most methods rely on a first coarse filter step to narrow down the set of possible candidates. For each of those, a more elaborated distance to the garbled word can be computed. In earlier works the first filter step is often realised using n-gram similarity (Owolabi and McGregor 1988).

The method described in (Oflazer 1996) can easily deal with very large or even infinite lexica and is based on the Levenshtein distance instead of n-gram measures. It introduces the idea to walk through the lexicon organised as a finite state automaton, and to control the walk such that only prefixes that potentially lead to a correction candidate are generated. This idea is refined in (Schulz and Mihov 2002) and (Mihov and Schulz 2004) and lead to the concept of universal Levenshtein automata used above. In more recent work (Schulz et al. 2007), these automata are not restricted to the Levenshtein distance but can be configured to a customized rule set.

8 Conclusion

In this paper we introduced a method for computing correction suggestions for ill-formed words in historical documents, assuming that historical language variation is described in terms of a matching procedure and a lexicon for modern language. Correction candidates are historical spelling variants obtained from modern words via matching.

The method does not use a special lexicon for historical language and it avoids the explicit representation of all matching variants. Instead, only the lexicon of modern words and the set of rewrite rules for the matching procedure are used as input. The importance of this feature relies on the fact that the number of possible spellings in historical documents is astronomic, hence an explicit representation of all possible historical words seems difficult.

The method is very efficient and flexible in the sense that pattern sets and lex-

ica can be changed rapidly at low costs. This is important in applications where document collections from distinct centuries and places are processed.

In the meantime, more complex applications have been built on top of the method. The first application makes use of the fact that our method is able to produce with each correction suggestion a pattern trace and an OCR trace. Given an OCRed historical text as input, the traces found for the best correction suggestions are used to collect hypotheses on frequent OCR errors and on the kind of historical language variation - the typical patterns - occurring in the document. This information can be used for improving OCR in a feedback loop. Furthermore, the method introduced in this paper is used as the kernel of a system for postcorrecting OCR results on historical document collections.

Acknowledgements. The author was supported by the EU project IMPACT, which provides a nice natural home for research on language processing with a focus on historical documents.

References

Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

Dawn Archer, Andrea Ernst-Gerlach, Sebastian Kempen, Thomas Pilz, and Paul Rayson. The identification of spelling variants in english and german historical texts: manual or automatic. In *Proceedings of the Digital Humanities Conference*, pages 3–5, Paris, France, 2006.

Eric Brill and Robert C. Moore. An improved error model for noisy channel spelling correction. In *ACL '00: Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, pages 286–293, Morristown, NJ, USA, 2000. Association for Computational Linguistics.

Horst Bunke. A fast algorithm for finding the nearest neighbor of a word in a dictionary, 1993. Aus http://citeseer.nj.nec.com/cs ResearchIndex.

Andrea Ernst-Gerlach and Norbert Fuhr. Generating search term variants for text collections with historic spellings. In *Proceedings of the 28th European Conference on Information Retrieval Research (ECIR 2006)*. Springer, 2006.

Andrea Ernst-Gerlach and Norbert Fuhr. Retrieval in text collections with historic spelling using linguistic and spelling variants. In *JCDL '07: Proceedings of the 7th ACM/IEEE-CS joint conference on Digital libraries*, pages 333–341, New York, NY, USA, 2007. ACM.

EU project Europeana, http://www.europeana.eu/.

Google Books, http://books.google.com/.

Annette Gotscharek, Andreas Neumann, Ulrich Reffle, Christoph Ringlstetter, and Klaus U. Schulz. Enabling information retrieval on historical document collections: the role of matching procedures and special lexica. In *AND '09: Proceedings of The Third Workshop on Analytics for Noisy Unstructured Text Data*, pages 69–76, New York, NY, USA, 2009. ACM.

Annette Gotscharek, Ulrich Reffle, Christoph Ringlstetter, and Klaus U. Schulz. On lexical resources for digitization of historical documents. In *DocEng '09: Proceedings of the 9th ACM symposium on Document engineering*, pages 193–200, New York, NY, USA, 2009. ACM.

Andreas Hauser, Markus Heller, Elisabeth Leiss, Klaus U. Schulz, and Christiane Wanzeck. Information access to historical documents from the early new high german period. In *IJCAI-2007 Workshop on Analytics for Noisy Unstructured Text Data*, 2006.

EU project Improving Access to Text IMPACT, http://www.impact-project.eu/.

Stoyan Mihov and Klaus U. Schulz. Fast approximate search in large dictionaries. *Computational Linguistics*, 30(4):451–477, December 2004.

G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, 2001.

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

O. Owolabi and D.R. McGregor. Fast approximate string matching. *Software - Practice and Experience*, 18(4):387–393, 1988.

Thomas Pilz, Wolfram Luther, U. Ammon, and Norbert Fuhr. Rule-based search in text databases with nonstandard orthography. In *Proceedings ACH/ALLC*, 2005.

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

Klaus U. Schulz and Stoyan Mihov. Fast String Correction with Levenshtein-Automata. *International Journal of Document Analysis and Recognition*, 5(1):67–85, 2002.

K. Schulz, S. Mihov, and P. Mitankin. Fast selection of small and precise candidate sets from dictionaries for text correction tasks. In *ICDAR '07: Proceedings of the Ninth International Conference on Document Analysis and Recognition*, pages 471–475, Washington, DC, USA, 2007. IEEE Computer Society.

Sun Wu and Udi Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.