Fast Approximate Search in Large Dictionaries

Stoyan Mihov*
Bulgarian Academy of Sciences

Klaus U. Schulz†
University of Munich

The need to correct garbled strings arises in many areas of natural language processing. If a dictionary is available that covers all possible input tokens, a natural set of candidates for correcting an erroneous input P *is the set of all words in the dictionary for which the Levenshtein distance to* P *does not exceed a given (small) bound* k. *In this article we describe methods for efficiently selecting such candidate sets. After introducing as a starting point a basic correction method based on the concept of a "universal Levenshtein automaton," we show how two filtering methods known from the field of approximate text search can be used to improve the basic procedure in a significant way. The first method, which uses standard dictionaries plus dictionaries with reversed words, leads to very short correction times for most classes of input strings. Our evaluation results demonstrate that correction times for fixed-distance bounds depend on the expected number of correction candidates, which decreases for longer input words. Similarly the choice of an optimal filtering method depends on the length of the input words.*

1. Introduction

In this article, we face a situation in which we receive some input in the form of strings that may be garbled. A dictionary that is assumed to contain all possible correct input strings is at our disposal. The dictionary is used to check whether a given input is correct. If it is not, we would like to select the most plausible correction candidates from the dictionary. We are primarily interested in applications in the area of natural language processing in which the background dictionary is very large and fast selection of an appropriate set of correction candidates is important. By a "dictionary," we mean any regular (finite or infinite) set of strings. Some possible concrete application scenarios are the following:

- The dictionary describes the set of words of a highly inflectional or agglutinating language (e.g., Russian, German, Turkish, Finnish, Hungarian) or a language with compound nouns (German). The dictionary is used by an automated or interactive spelling checker.
- The dictionary is multilingual and describes the set of all words of a family of languages. It is used in a system for postcorrection of results of OCR in which scanned texts have a multilingual vocabulary.

* Linguistic Modelling Department, Institute for Parallel Processing, Bulgarian Academy of Sciences, 25A, Akad. G. Bonchev Str., 1113 Sofia, Bulgaria. E-mail: stoyan@lml.bas.bg
† Centrum für Informations-und Sprachverarbeitung, Ludwig-Maximilians-Universität-München, Oettingenstr. 67, 80538 Munchen, Germany. E-mail: schulz@cis.uni-muenchen.de

Submission received: 12 July 2003; Revised submission received: 28 February 2004; Accepted for publication: 25 March 2004

- The dictionary describes the set of all indexed words and phrases of an Internet search engine. It is used to determine the plausibility that a new query is correct and to suggest "repaired" queries when the answer set returned is empty.
- The input is a query to some bibliographic search engine. The dictionary contains titles of articles, books, etc.

The selection of an appropriate set of correction candidates for a garbled input P is often based on two steps. First, all entries W of the dictionary are selected for which the distance between P and W does not exceed a given bound k. Popular distance measures are the Levenshtein distance (Levenshtein 1966; Wagner and Fischer 1974; Owolabi and McGregor 1988; Weigel, Baumann, and Rohrschneider 1995; Seni, Kripasundar, and Srihari 1996; Oommen and Loke 1997) or n-gram distances (Angell, Freund, and Willett 1983; Owolabi and McGregor 1988; Ukkonen 1992; Kim and Shawe-Taylor 1992, 1994) Second, statistical data, such as frequency information, may be used to compute a ranking of the correction candidates. In this article, we ignore the ranking problem and concentrate on the first step. For selection of correction candidates we use the standard Levenshtein distance (Levenshtein 1966). In most of the above-mentioned applications, the number of correction candidates becomes huge for large values of k. Hence small bounds are more realistic.

In light of this background, the algorithmic problem discussed in the article can be described as follows:

> Given a pattern P, a dictionary D, and a small bound k, efficiently compute the set of all entries W in D such that the Levenshtein distance between P and W does not exceed k.

We describe a basic method and two refinements for solving this problem. The basic method depends on the new concept of a universal deterministic Levenshtein automaton of fixed degree k. The automaton of degree k may be used to decide, for arbitrary words U and V, whether the Levenshtein distance between U and V does not exceed k. The automaton is "universal" in the sense that it does not depend on U and V. The input of the automaton is a sequence of bitvectors computed from U and V. Though universal Levenshtein automata have not been discussed previously in the literature, determining Levenshtein neighborhood using universal Levenshtein automata is closely related to a more complex table-based method described by the authors Schulz and Mihov (2002). Hence the main advantage of the new notion is its conceptual simplicity. In order to use the automaton for solving the above problem, we assume that the dictionary is given as a determininistic finite-state automaton. The basic method may then be described as a parallel backtracking traversal of the universal Levenshtein automaton and the dictionary automaton. Backtracking procedures of this form are well-known and have been used previously: for example, by Oflazer (1996) and the authors Schulz and Mihov (2002).

For the first refinement of the basic method, a filtering method used in the field of approximate text search is adapted to the problem of approximate search in a dictionary. In this approach, an additional "backwards" dictionary D^{-R} (representing the set of all reverses of the words of a given dictionary D) is used to reduce approximate search in D with a given bound $k \geq 1$ to related search problems for smaller bounds $k' < k$ in D and D^{-R}. As for the basic method, universal Levenshtein automata are used to control the search. Ignoring very short input words and correction bound $k = 1$,

this approach leads to a drastic increase in speed. Hence the "backwards dictionary method" can be considered the central contribution of this article.

The second refinement, which is only interesting for bound $k = 1$ and short input words, also uses a filtering method from the field of approximate text search (Muth and Manber 1996; Mor and Fraenkel 1981). In this approach, "dictionaries with single deletions" are used to reduce approximate search in a dictionary D with bound $k = 1$ to a conventional lookup technique for finite-state transducers. Dictionaries with single deletions are constructed by deleting the symbol at a fixed position n in all words of a given dictionary.

For the basic method and the two refinements, detailed evaluation results are given for three dictionaries that differ in terms of the number and average length of entries: a dictionary of the Bulgarian language with 965,339 entries (average length 10.23 symbols), a dictionary of German with 3,871,605 entries (dominated by compound nouns, average length 18.74 symbols), and a dictionary representing a collection of 1,200,073 book titles (average length 47.64 symbols). Tests were restricted to distance bounds $k = 1, 2, 3$. For the approach based on backwards dictionaries, the average correction time for a given input word—including the displaying of all correction suggestions—is between a few microseconds and a few milliseconds, depending on the dictionary, the length of the input word, and the bound k. Correction times over one millisecond occur only in a few cases for bound $k = 3$ and short input words. For bound $k = 1$, which is important for practical applications, average correction times did not exceed 40 microseconds.

As a matter of fact, correction times are a joint result of hardware improvements and algorithmic solutions. In order to judge the quality of the correction procedure in absolute terms, we introduce an "idealized" correction algorithm in which any kind of blind search and superfluous backtracking is eliminated. Based on an analysis of this algorithm, we believe that using purely algorithmic improvements, our correction times can be improved only by a factor of 50–250, depending on the kind of dictionary used. This factor represents a theoretical limit in the sense that the idealized algorithm probably cannot be realized in practice.

This article is structured as follows. In Section 2, we collect some formal preliminaries. In Section 3, we briefly summarize some known techniques from approximate string search in a text. In Section 4, we introduce universal deterministic Levenshtein automata of degree k and describe how the problem of deciding whether the Levenshtein distance between two strings P and W does not exceed k can be efficiently solved using this automaton. Since the method is closely related to a table-based approach introduced by the authors (Schulz and Mihov 2002), most of the formal details have been omitted. Sections 5, 6, and 7 describe, respectively, the basic method, the refined approach based on backwards dictionaries, and the approach based on dictionaries with single deletions. Evaluation results are given for the three dictionaries mentioned above. In Section 8 we briefly comment on the difficulties that we encountered when trying to combine dictionary automata and similarity keys (Davidson 1962; Angell, Freund, and Willett 1983; Owolabi and McGregor 1988; Sinha 1990; Kukich 1992; Anigbogu and Belaid 1995; Zobel and Dart 1995; de Bertrand de Beuvron and Trigano 1995). Theoretical bounds for correction times are discussed in Section 9.

The problem considered in this article is well-studied. Since the number of contributions is enormous, a complete review of related work cannot be given here. Relevant references with an emphasis on spell-checking and OCR correction are Blair (1960), Riseman and Ehrich (1971), Ullman (1977), Angell, Freund, and Willett (1983), Srihari, Hull, and Choudhari (1983), Srihari (1985), Takahashi et al. (1990), Kukich (1992), Zobel

and Dart (1995), and Dengel et al. (1997). Exact or approximate search in a dictionary is discussed, for example, in Wells et al. (1990), Sinha (1990), Bunke (1993), Oflazer (1996), and Baeza-Yates and Navarro (1998). Some relevant work from approximate search in texts is described in Section 3.

2. Formal Preliminaries

We assume that the reader is familiar with the basic notions of formal language theory as described, for example, by Hopcroft and Ullman (1979) or Kozen (1997). As usual, **finite-state automata** (FSA) are treated as tuples of the form $A = \langle \Sigma, Q, q_0, F, \Delta \rangle$, where Σ is the input alphabet, Q is the set of states, $q_0 \in Q$ is the initial state, F is the set of final states, and $\Delta \subseteq Q \times \Sigma^\varepsilon \times Q$ is the transition relation. Here ε denotes the empty string and $\Sigma^\varepsilon := \Sigma \cup \{\varepsilon\}$. The generalized transition relation $\hat{\Delta}$ is defined as the smallest subset of $Q \times \Sigma^* \times Q$ with the following closure properties:

- For all $q \in Q$ we have $(q, \varepsilon, q) \in \hat{\Delta}$.
- For all $q_1, q_2, q_3 \in Q$ and $W_1, W_2 \in \Sigma^*$, if $(q_1, W_1, q_2) \in \hat{\Delta}$ and $(q_2, W_2, q_3) \in \Delta$, then also $(q_1, W_1 W_2, q_3) \in \hat{\Delta}$.

We write $\mathcal{L}(A)$ for the language accepted by A. We have $\mathcal{L}(A) = \{W \in \Sigma^* \mid \exists q \in F : (q_0, W, q) \in \hat{\Delta}\}$. Given A as above, the set of **active states** for input $W \in \Sigma^*$ is $\{q \in Q \mid (q_0, W, q) \in \hat{\Delta}\}$.

A finite-state automaton A is **deterministic** if the transition relation is a function $\delta : Q \times \Sigma \to Q$. Let $A = \langle \Sigma, Q, q_0, F, \delta \rangle$ be a deterministic FSA, and let $\delta^* : Q \times \Sigma^* \to Q$ denote the generalized transition function, which is defined in the usual way. For $q \in Q$, we write $\mathcal{L}_A(q) := \{U \in \Sigma^* \mid \delta^*(q, U) \in F\}$ for the language of all words that lead from q to a final state.

The length of a word W is denoted by $|W|$. Regular languages over Σ are defined in the usual way. With $\mathcal{L}_1 \cdot \mathcal{L}_2$ we denote the concatenation of the languages $\mathcal{L}_1$ and $\mathcal{L}_2$. It is well-known that for any regular language $\mathcal{L}$, there exists a deterministic FSA $A_\mathcal{L}$ such that $\mathcal{L}(A) = \mathcal{L}$ and $A_\mathcal{L}$ is minimal (with respect to number of states) among all deterministic FSA accepting $\mathcal{L}$. $A_\mathcal{L}$ is unique up to renaming of states.

A **p-subsequential transducer** is a tuple $T = \langle \Sigma, \Pi, Q, q_0, F, \delta, \lambda, \Psi \rangle$, where

- $\langle \Sigma, Q, q_0, F, \delta \rangle$ is a deterministic finite-state automaton;
- Π is a finite output alphabet;
- $\lambda : Q \times \Sigma \to \Pi^*$ is a function called the **transition output function**;
- the **final function** $\Psi : F \to 2^{\Pi^*}$ assigns to each $f \in F$ a set of strings over Π, where $|\Psi(f)| \leq p$.

The function λ is extended to the domain $Q \times \Sigma^*$ by the following definition of λ^*:

$$\begin{aligned} \forall q \in Q \qquad & (\lambda^*(q, \varepsilon) = \varepsilon) \\ \forall q \in Q\, \forall U \in \Sigma^*\, \forall a \in \Sigma \qquad & (\lambda^*(q, Ua) = \lambda^*(q, U)\lambda(\delta^*(q, U), a)) \end{aligned}$$

The **input language** of the transducer is $\mathcal{L}(T) := \{U \in \Sigma^* \mid \delta^*(q_0, U) \in F\}$. The subsequential transducer maps each word from the input language to a set of at most

p output words. The **output function** $O_T : \mathcal{L}(T) \rightarrow 2^{\Pi^*}$ of the transducer is defined as follows:

$$\forall U \in \mathcal{L}(T)\ (O_T(U) = \lambda^*(q_0, U) \cdot \Psi(\delta^*(q_0, U)))$$

By a **dictionary**, we mean a regular (finite or infinite) set of strings over a given alphabet Σ. Using the algorithm described by Daciuk et al. (2000), the minimal deterministic FSA A_D accepting a finite dictionary D can be effectively computed.

By a **dictionary with output sets**, we mean a regular (finite or infinite) set of input strings over a given alphabet together with a function that maps each of the input strings to a finite set of output strings. Given a finite dictionary with output sets, we can effectively compute, using the algorithm described by Mihov and Maurel (2001), the minimal subsequential transducer that maps each input string to its set of output strings.

3. Background

In this section, we describe some established work that is of help in understanding the remainder of the article from a nontechnical, conceptual point of view. After introducing the Levenshtein distance, we describe methods for computing the distance, for checking whether the distance between two words exceeds a given bound, and for approximate search for a pattern in a text. The similarities and differences described below between approximate search in a text, on the one hand, and approximate search in a dictionary, on the other hand, should help the reader understand the contents of the following sections from a broader perspective.

3.1 Computation of Levenshtein Distance

The most prominent metric for comparing strings is the Levenshtein distance, which is based on the notion of a primitive edit operation. In this article, we consider the standard Levenshtein distance. Here the primitive operations are the **substitution** of one symbol for another symbol, the **deletion** of a symbol, and the **insertion** of a symbol. Obviously, given two words W and V in the alphabet Σ, it is always possible to rewrite W into V using primitive edit operations.

Definition 1
Let P, W be words in the alphabet Σ. The (standard) **Levenshtein distance** between P and W, denoted $d_L(P, W)$, is the minimal number of primitive edit operations (substitutions, deletions, insertions) that are needed to transform P into W.

The Levenshtein distance between two words P and W can be computed using the following simple dynamic programming scheme, described, for example, by Wagner and Fischer (1974):

$$\begin{aligned} d_L(\varepsilon, W) &= |W| \\ d_L(P, \varepsilon) &= |P| \\ d_L(Pa, Wb) &= \begin{cases} d_L(P, W) & \text{if } a = b \\ 1 + \min(d_L(P, W), d_L(Pa, W), d_L(P, Wb)) & \text{if } a \neq b \end{cases} \end{aligned}$$

for $P, W \in \Sigma^*$ and $a, b \in \Sigma$. Given $P = p_1 \ldots p_m$ and $W = w_1 \ldots w_n$ $(m, n \geq 0)$, a standard way to apply the scheme is as follows: Proceeding top-down and from left to right, the cells of an $(m+1) \times (n+1)$ table $T_L(P, W)$ are filled, where entry (i, j) of $T(P, W)$ is $d_L(p_1 \ldots p_i, w_1 \ldots w_j)$ $(0 \leq i \leq m,\ 0 \leq j \leq n)$ (Wagner and Fischer 1974). The first two

		h	c	h	o	l	d
	0	1	2	3	4	5	6
c	1	1	1	2	3	4	5
h	2	1	2	1	2	3	4
o	3	2	2	2	1	2	3
l	4	3	3	3	2	1	2
d	5	4	4	4	3	2	1

Figure 1
Computation of the Levenshtein distance using dynamic programming and filling table $T_L(\text{chold}, \text{hchold})$. Shaded regions represent diagonals in Ukkonen's approach (cf. Section 3.2).

clauses above are used for initialization and yield, respectively, the first column and the first row. The third clause is used to compute the remaining entries. The table for the strings *chold* and *hchold* is shown in Figure 1.

3.2 Testing Levenshtein Neighborhood

The algorithm of Wagner and Fischer, which has time complexity $O(m \cdot n)$, has been improved and generalized in many aspects. (See, for example, Stephen [1994] for a survey). We briefly sketch a more efficient variant that can be used for the restricted problem of deciding whether the Levenshtein distance between two words P and W exceeds a fixed bound, k. Ukkonen (1985a) shows that in this case only the values of $2k + 1$ "diagonals" of $T_L(P, W)$ are essential for a test to make such a determination. Figure 1 illustrates the situation in which $k = 2$. Ukkonen obtained an algorithm with time complexity $O(k \cdot \min(m, n))$. He used the test for determining whether the Levenshtein distance between two words exceeds a given bound to derive an algorithm for computing the edit distance with complexity $O(\min(m, n) \cdot d_L(P, W))$.

3.3 Approximate Search for a Pattern in a Text

A problem closely related to approximate search in a dictionary is approximate search for a pattern in a text (AST): Given two strings P and T (called, respectively, the **pattern** and the **text**), find all occurrences T' of substrings of T that are within a given distance of P. Each occurrence T' is called a hit. In the following discussion, we consider the case in which a fixed bound k for the Levenshtein distance between P and potential hits is specified.

3.3.1 Adapting the Dynamic Programming Scheme. A simple adaptation of the Wagner-Fischer algorithm may be used for approximate search for a pattern $P = p_1 \cdots p_m$ in a text $T = t_1 \cdots t_n$. As before, we compute an $(m+1) \times (n+1)$ table $T_{AST}(P, T)$. Entry (i, j) of $T_{AST}(P, T)$ has value h if h is the minimal Levenshtein distance between $p_1 \cdots p_i$ and a substring of T with last symbol (position) t_j (j). From the definition, we see that all cells in line 0 have to be initialized with 0. The remaining computation proceeds as above. For a given bound k, the output consists of all positions j such that entry (m, j) does not exceed k. Note that the positions j in the output are the end points in T of approximate matches of P. Figure 2 illustrates a search employing dynamic programming.

3.3.2 Automaton Approach. Several more-efficient methods for approximate search of a pattern P in a text T take as their starting point a simple nondeterministic finite-state automaton, $A_{AST}(P, k)$, which accepts the language of all words with Levenshtein

		t	h	i	s	_	c	h	i	l	d
	0	0	0	0	0	0	0	0	0	0	0
c	1	1	1	1	1	1	0	1	1	1	1
h	2	2	1	2	2	2	1	0	1	2	2
o	3	3	2	2	3	3	2	1	1	2	3
l	4	4	3	3	3	4	3	2	2	1	2
d	5	5	4	4	4	4	4	3	3	2	1

Figure 2
Approximate search of pattern *chold* in a text using dynamic programming.

distance $\leq k$ to some word in $\Sigma^* \cdot P$ (Ukkonen 1985b; Wu and Manber 1992; Baeza-Yates and Navarro 1999). The automaton for pattern *chold* and distance bound $k = 2$ is shown in Figure 3. States are numbered in the form b^e. The "base number" b determines the position of the state in the pattern. The "exponent" e indicates the error level, that is, the number of edit errors that have been observed. Horizontal transitions encode "normal" transitions in which the text symbol matches the expected next symbol of the pattern. Vertical transitions represent insertions, nonempty (respectively, empty) diagonal transitions represent substitutions (respectively, deletions). In the example shown in Figure 3, final states are 5^0, 5^1, and 5^2. It is obvious that when using a given text T as input, we reach a final state of $A_{AST}(P, 2)$ exactly at those positions where a substring T' of T ends such that $d_L(P, T') \leq 2$. For other bounds k, we just have to vary the number of levels. Note that a string can be accepted in $A_{AST}(P, k)$ at several final states. In order to determine the optimal distance between P and a substring T' of T ending at a certain position, it is necessary to determine the final state that can be reached that has the smallest exponent.

In the remainder of the article, the set of all states with base number i is called the ith **column** of $A_{AST}(P, k)$.

Remark 1
There is a direct relationship between the entries in column j of the dynamic programming table $T_{AST}(P, T)$ and the set of active states of $A_{AST}(P, k)$ that are reached with input $t_1 \cdots t_j$. Entry (i, j) of $T_{AST}(P, T)$ has the value $h \leq k$ iff h is the exponent of the bottom-most active state in the ith column of $A_{AST}(P, k)$. For example, in Figure 3, the set of active states of $A_{AST}(\text{chold}, k)$ reached after reading the two symbols t and h is highlighted. The bottom-most elements 0^0, 1^1, 2^1, and 3^2 correspond to the entries $0, 1, 1,$ and 2 shaded in the upper part of the third column of Figure 2.

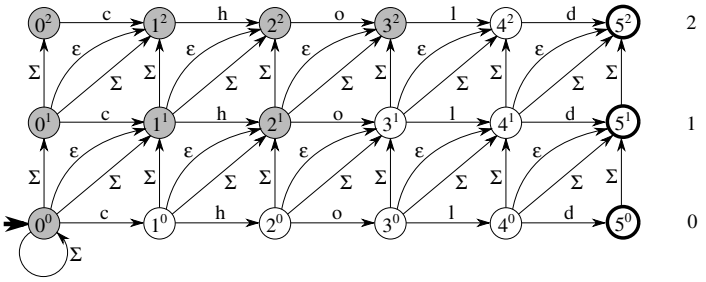

Figure 3
Nondeterministic automaton $A_{AST}(\text{chold}, 2)$ for approximate search with pattern *chold* and distance bound $k = 2$. Active states after symbols t and h have been read are highlighted.

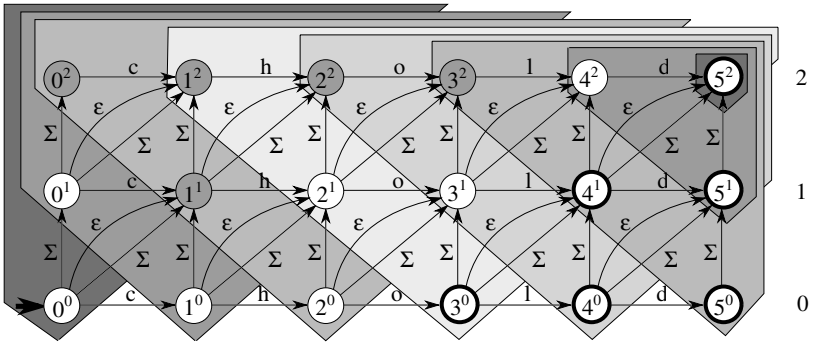

Figure 4
Nondeterministic automaton $A(\text{chold}, 2)$ for testing Levenshtein distance with bound $k = 2$ for pattern *chold*. Triangular areas are highlighted. Dark states are active after symbols *h* and *c* have been read.

The direct use of the nondeterministic automaton $A_{AST}(P, k)$ for conducting approximate searches is inefficient. Furthermore, depending on the length m of the pattern and the error bound k, the explicit construction and storage of a deterministic version of $A_{AST}(P, k)$ might be difficult or impossible. In practice, simulation of determinism via bit-parallel computation of sets of active states gives rise to efficient and flexible algorithms. See Navarro (2001) and Navarro and Raffinot (2002) for surveys of algorithms along this line.

4. Testing Levenshtein Neighborhood with Universal Deterministic Levenshtein Automata

In our approach, approximate search of a pattern P in a dictionary D is traced back to the problem of deciding whether the Levenshtein distance between P and an entry W of D exceeds a given bound k. A well-known method for solving this problem is based on a nondeterministic automaton $A(P, k)$ similar to $A_{AST}(P, k)$. A string W is accepted by $A(P, k)$ iff $d_L(P, W) \leq k$. The automaton $A(P, k)$ does not have the initial Σ loop that is needed in $A_{AST}(P, k)$ to traverse the text. The automaton for pattern *chold* and distance bound $k = 2$ is shown in Figure 4. Columns of $A(P, k)$ with numbers $0, \ldots, m = |P|$ are defined as for $A_{AST}(P, k)$. In $A(P, k)$, we use as final states all states q from which we can reach one of the states in column m using a (possibly empty) sequence of ϵ-transitions. The reason for this modification—which obviously does not change the set of accepted words—will become apparent later.

We now show that for fixed small error bounds k, the explicit computation of $A(P, k)$, in a deterministic or nondeterministic variant, can be completely avoided. In our approach, pattern P and entry $W = w_1 \cdots w_n$ are compared to produce a sequence of n bitvectors $\vec{\chi}_1, \ldots, \vec{\chi}_n$. This sequence is used as input for a fixed automaton $A^{\forall}(k)$. The automaton $A^{\forall}(k)$ is deterministic and "universal" in the sense that it does not depend on a given pattern P. For each bound k, there is just one fixed automaton $A^{\forall}(k)$, which is precomputed once and used for arbitrary patterns P and words W. $A^{\forall}(k)$ accepts input $\vec{\chi}_1, \ldots, \vec{\chi}_n$ iff $d_L(P, W) \leq k$. The efficiency of this method relies on the fact that given $A^{\forall}(k)$, we need only one operation for each transition of the recognition phase after the initial computation of the bitvectors $\vec{\chi}_1, \ldots, \vec{\chi}_n$.

It is worth mentioning that the possibility of using a fixed universal automaton $A^{\forall}(k)$ instead of a specific automaton $A(P, k)$ for each pattern P is based on special features of the automata $A(P, k)$ (cf. Remark 2); a similar technique for $A_{AST}(P, k)$

appears to be impossible. For defining states, input vectors and transitions of $A^{\forall}(k)$, the following two definitions are essential:

Definition 2
The **characteristic vector** $\vec{\chi}(w, V)$ of a symbol $w \in \Sigma$ in a word $V = v_1 \cdots v_n \in \Sigma^*$ is the bitvector of length n where the ith bit is set to 1 iff $w = v_i$.

Definition 3
Let P denote a pattern of length m. The **triangular area of a state** p of $A(P, k)$ consists of all states q of $A(P, k)$ that can be reached from p using a (potentially empty) sequence of u upward transitions and, in addition, $h \leq u$ horizontal or reverse (i.e., leftward) horizontal transitions. Let $0 \leq i \leq m$. By **triangular area** i, we mean the triangular area of state i^0. For $j = 1, \ldots, k$, by **triangular area** m+j, we mean the triangular area of the state m^j.

For example, in Figure 4, triangular areas $0, \ldots, 7$ of $A(\text{chold}, 2)$ are shown.

In Remark 1, we pointed to the relationship between the entries in column i of table $T_{AST}(P, T)$ and the set of active states of $A_{AST}(P, k)$ that are reached with input $w_1 \cdots w_i$. A similar relationship holds between the entries in column i of table $T_L(P, T)$ and the set of active states of the automaton $A(P, k)$ that are reached with input $w_1 \cdots w_i$. Triangular area i corresponds to the ith column of the subregion of $T_L(P, T)$ given by the $2k + 1$ diagonals used in Ukkonen's (1985a) approach. The left-to-right orientation in $A(P, k)$ corresponds to a top-down orientation in $T_L(P, T)$. As an illustration, the active states of $A(\text{chold}, 2)$ after symbols h and c have been consumed are marked in Figure 4. The exponents 2, 1, 2, and 2 of the bottom-most active states in columns 1, 2, 3, and 4 are found in the shaded region of the third column of Figure 1.

Remark 2
It is simple to see that for any input string $W = w_1 \ldots w_n$, the set of active states of $A(P, k)$ reached after reading the ith symbol w_i of W is a subset of triangular area i ($0 \leq i \leq \min\{n, m + k\}$). For $i > m + k$, the set is empty. Furthermore, the set of active states that is reached after reading symbol w_i depends only

1. on the previous set of active states (the set reached after reading $w_1 \ldots w_{i-1}$, a subset of the triangular area $i - 1$);
2. on the characteristic vector $\vec{\chi}(w_i, p_l \cdots p_i \cdots p_r)$ where $l = \max\{1, i - k\}$ and $r = \min\{m, i + k\}$.

The following description of $A^{\forall}(k)$ proceeds in three steps that introduce, in order, input vectors, states, and the transition function. States and transition function are described informally.

1. Input vectors. Basically we want to use the vectors $\chi(w_i, p_l \cdots p_i \cdots p_r)$, which are of length $\leq 2k + 1$, as input for $A^{\forall}(k)$. For technical reasons, we introduce two modifications. First, in order to standardize the length of the characteristic vectors that are obtained for the initial symbols $w_1, w_2, \ldots$, we define $p_0 = p_{-1} = \ldots = p_{-k+1} := \$$. In other words, we attach to P a new prefix with k symbols \$. Here \$ is a new symbol that does not occur in W. Second imagine that we get to triangular area i after reading the ith letter w_i (cf. Remark 2). As long as $i \leq m - k - 1$, we know that we cannot reach a triangular area containing final states after reading w_i. In order to encode

this information in the input vectors, we enlarge the relevant subword of P for input w_i and consider one additional position $i+k+1$ on the right-hand side (whenever $i+k+1 \leq m$). This means that we use the vectors $\vec{\chi}_i := \vec{\chi}(w_i, p_{i-k} \cdots p_i \cdots p_r)$, where $r = \min\{m, i+k+1\}$, as input for $A^{\forall}(k)$, for $1 \leq i \leq \min\{n, m+k\}$. Consequently, for $0 \leq i \leq m-k-1$, the length of $\vec{\chi}_i$ is $2k+2$; for $i = m-k$ (respectively, $m-k+1, \ldots, m, \ldots, m+k$), the length of $\vec{\chi}_i$ is $2k+1$ (respectively, $2k, \ldots, k+1, \ldots, 1$).

Example 1
Consider Figure 4 where P is *chold* and $k = 2$. Input *hchold* is translated into the vectors

$$\begin{aligned}
\vec{\chi}_1 &= \vec{\chi}(h, \$\$chol) = 000100 \\
\vec{\chi}_2 &= \vec{\chi}(c, \$chold) = 010000 \\
\vec{\chi}_3 &= \vec{\chi}(h, chold) = 01000 \\
\vec{\chi}_4 &= \vec{\chi}(o, hold) = 0100 \\
\vec{\chi}_5 &= \vec{\chi}(l, old) = 010 \\
\vec{\chi}_6 &= \vec{\chi}(d, ld) = 01
\end{aligned}$$

The computation of the vectors $\vec{\chi}_i$ for input $W = w_1 \ldots w_n$ is based on a preliminary step in which we compute for each $\sigma \in \Sigma$ the vector $\vec{\kappa}(\sigma) := \vec{\chi}(\sigma, \$ \ldots \$ p_1 \ldots p_m)$ (using k copies of \$). The latter vectors are initialized in the form $\vec{\kappa}(w) := 0^{k+m}$. We then compute for $i = 1, \ldots, n$ the value $\vec{\kappa}(w_i) := \vec{\kappa}(w_i) \mid 0^{k+i-1}10^{m-i}$. Here the symbol $\mid$ denotes bitwise OR. Once we have obtained the values $\vec{\kappa}(w_i)$, which are represented as arrays, the vectors $\vec{\chi}_i := \vec{\chi}(w_i, p_{i-k} \ldots p_i \ldots p_r)$ can be accessed in constant time.

2. States. Henceforth, states of automata $A(P,k)$ will be called **positions**. Recall that a position is given by a base number and an exponent $e, 0 \leq e \leq k$ representing the error count. By a **symbolic triangular area**, we mean a triangular area in which "explicit" base numbers (like $1, 2, \ldots$) in positions are replaced by "symbolic" base numbers of a form described below. Two kinds of symbolic triangular areas are used. A unique "I-area" represents all triangular areas of automata $A(P,k)$ that do not contain final positions. The "integer variable" I is used to abstract from possible base numbers i, $0 \leq i \leq m-k-1$. Furthermore, $k+1$ "M-areas" are used to represent triangular areas of automata $A(P,k)$ that contain final positions. Variable M is meant to abstract from concrete values of m, which differ for distinct P. **Symbolic base numbers** are expressions of the form $I, I+1, I-1, I+2, I-2 \ldots$ (I-areas) or $M, M-1, M-2, \ldots$ (M-areas). The elements of the symbolic areas, which are called **symbolic positions**, are symbolic base numbers together with exponents indicating an error count. Details should become clear in Example 2. The use of expressions such as $(I+2)^2$ simply enables a convenient labeling of states of $A^{\forall}(k)$ (cf. Figure 6). Using this kind of labeling, it is easy to formulate a correspondence between derivations in automata $A(P,k)$ and in $A^{\forall}(k)$ (cf. properties C1 and C2 discussed below).

Example 2
The symbolic triangular areas for bound $k = 1$ are

$$\begin{aligned}
(I\text{-area}) &\quad \{I^0, (I-1)^1, I^1, (I+1)^1\} \\
(\text{First } M\text{-area}) &\quad \{M^0, (M-1)^1, M^1\} \\
(\text{Second } M\text{-area}) &\quad \{(M-1)^0, (M-2)^1, (M-1)^1, M^1\}
\end{aligned}$$

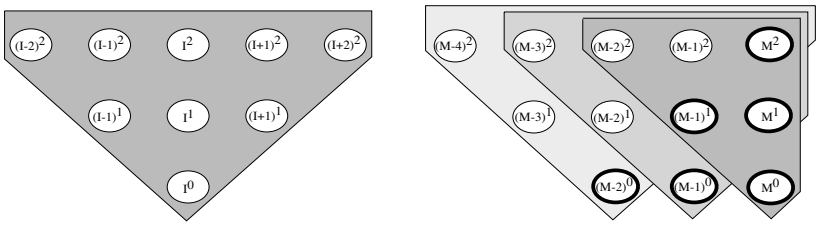

Figure 5
Symbolic triangular areas and symbolic final positions for bound $k = 2$ (cf. Example 2).

Symbolic final positions for $k = 1$ are M^1, M^0, and $(M-1)^0$. The symbolic triangular areas for $k = 2$ are indicated in Figure 5, in which the ellipses around symbolic final positions are rendered in boldface.

States of $A^\forall(k)$ are merely subsets of symbolic triangular areas. Subsets containing symbolic final positions are final states of $A^\forall(k)$, and $\{I^0\}$ is the start state. A special technique is used to reduce the number of states. Returning to automata of the form $A(P,k)$, it is simple to see that triangular areas often contain positions $p = g^e$ and $q = h^f$ where p "subsumes" q in the following sense: If, for some fixed input rest U, it is possible to reach a final position of $A(P,k)$ starting from q and consuming U, then we may also reach a final position starting from p using U. A corresponding notion of subsumption can be defined for symbolic positions. States of $A^\forall(k)$ are then defined as subsets of symbolic triangular areas that are free of subsumption in the sense that a symbolic position of a state is never subsumed by another position of the same state.

Example 3
The states of automaton $A^\forall(1)$ are shown in Figure 6. As a result of the above reduction technique, the only state containing the symbolic position I^0 is $\{I^0\}$, the start state. Each of the symbolic positions $(I-1)^1, I^1, (I+1)^1$ is subsumed by I^0.

3. Transition function. It remains to define the transition function $\delta_\forall$ of $A^\forall(k)$. We describe only the basic idea. Imagine an automaton $A(P,k)$, where the pattern P has length m. Let $W = w_1 \dots w_n$ denote an input word. Let S_i^P denote the set of active positions of $A(P,k)$ that are reached after reading the ith symbol w_i $(1 \le i \le n)$. For simplicity, we assume that in each set, all subsumed positions are erased. In $A^\forall(k)$ we have a parallel acceptance procedure in which we reach, say, state $S_i^\forall$ after reading $\vec{\chi}_i := \vec{\chi}(w_i, p_{i-k} \cdots p_i \cdots p_r)$, where $r = \min\{m, i+k+1\}$, as above, for $1 \le i \le n$. Transitions are defined in such a way that C1 and C2 hold:

C1. For all parallel sets S_i^P and $S_i^\forall$ of the two sequences

$$\begin{array}{ccccccc} S_0^P & S_1^P & \dots & S_i^P & \dots & S_n^P \\ S_0^\forall & S_1^\forall & \dots & S_i^\forall & \dots & S_n^\forall \end{array}$$

the set S_i^P is obtained from $S_i^\forall$ by instantiating the letter I by i whenever $S_i^\forall$ uses variable I and instantiating M by m in the other cases.

C2. Whenever S_i^P contains a final position, then $S_i^\forall$ is final.

Given properties C1 and C2, it follows immediately that $A(P,k)$ accepts $w_1 \dots w_n$ iff $A^\forall(k)$ accepts $\vec{\chi}_1, \dots, \vec{\chi}_n$.

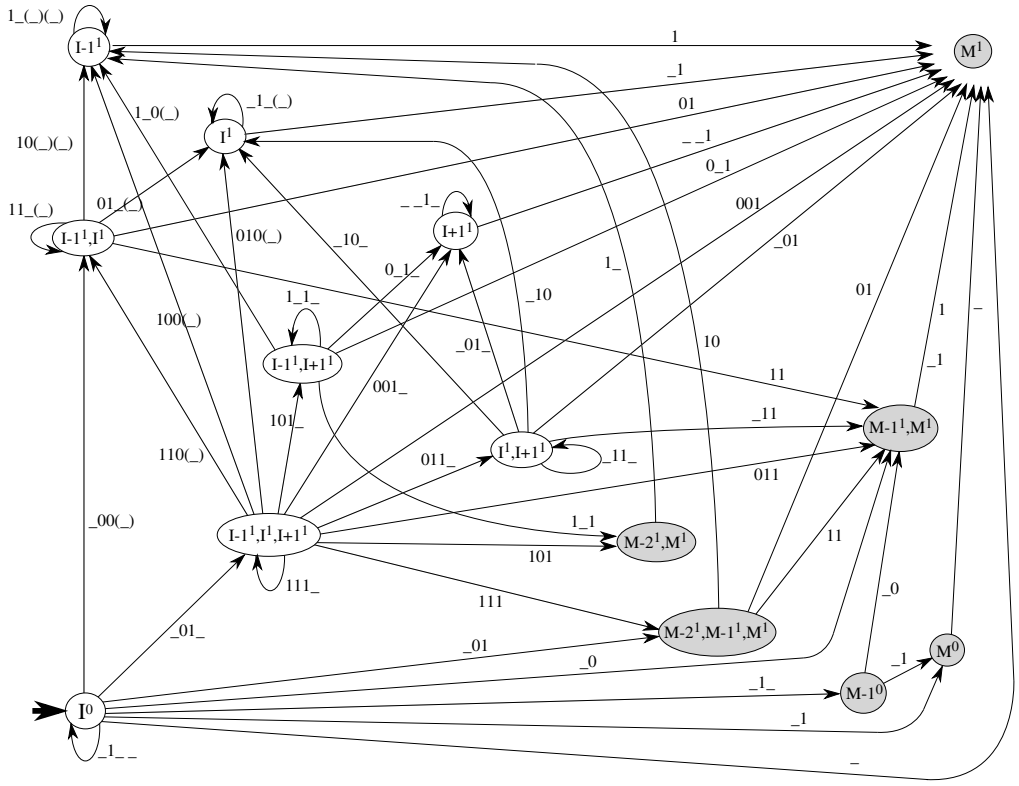

Figure 6
The universal deterministic Levenshtein automaton $A^{\forall}(1)$. See Example 4 for notation.

Example 4
The universal deterministic automaton $A^{\forall}(1)$ is shown in Figure 6. (Some redundant transitions departing from nonfinal states $S \neq \{I^0\}$ and using vectors of length ≤ 3 have been omitted.) The symbol _ stands for either 1 or 0. Moreover, $\vec{\chi}(\_)$ is shorthand for $\vec{\chi}_{\_}$ or $\vec{\chi}$. In order to illustrate the use of $A^{\forall}(1)$, consider the pattern P of the form *chold*. Input *child* is translated into the sequence

$$\begin{aligned}
\vec{\chi}_1 &= \vec{\chi}(c, \$cho) = 0100 \\
\vec{\chi}_2 &= \vec{\chi}(h, chol) = 0100 \\
\vec{\chi}_3 &= \vec{\chi}(i, hold) = 0000 \\
\vec{\chi}_4 &= \vec{\chi}(l, old) = 010 \\
\vec{\chi}_5 &= \vec{\chi}(d, ld) = 01
\end{aligned}$$

Starting from state $\{I^0\}$, we successively reach $\{I^0\}$, $\{I^0\}$, $\{(I-1)^1, I^1\}$, $\{I^1\}$, $\{M^1\}$. Hence *child* is accepted. In a similar way the input word *cold* is translated into the sequence 0100–0010–0010–001. Starting from $\{I^0\}$, we successively reach $\{I^0\}$, $\{(I-1)^1, I^1, (I+1)^1\}$, $\{(I+1)^1\}$, $\{M^1\}$. Hence *cold* is also accepted. Third, input *hchold* is translated into the sequence 0010–1000–1000–100–10–1. We reach successively $\{(I-1)^1, I^1, (I+1)^1\}$, $\{(I-1)^1\}$, $\{(I-1)^1\}$, $\{(I-1)^1\}$, $\{(I-1)^1\}$, $\{M^1\}$. Hence *hchold* is accepted as well.

For larger values of k, the number of states of $A^{\forall}(k)$ grows rapidly. $A^{\forall}(2)$ has 50 nonfinal states and 40 final states. The automaton $A^{\forall}(3)$ has 563 states. When we tried

to minimize the automata $A^\forall(1)$, $A^\forall(2)$, and $A^\forall(3)$, we found that these three automata are already minimal. However, we do not have a general proof that our construction always leads to minimal automata.

5. Approximate Search in Dictionaries Using Universal Levenshtein Automata

We now describe how to use the universal deterministic Levenshtein automaton $A^\forall(k)$ for approximate search for a pattern in a dictionary.

5.1 Basic Correction Algorithm

Let D denote the background dictionary, and let $P = p_1 \dots p_m$ denote a given pattern. Recall that we want to compute for some fixed bound k the set of all entries $W \in D$ such that $d_L(P, W) \leq k$. We assume that D is implemented in the form of a deterministic finite-state automaton $A_D = \langle \Sigma, Q^D, q_0^D, F^D, \delta^D \rangle$, the **dictionary automaton**. Hence $\mathcal{L}(A_D)$ represents the set of all correct words.

Let $A^\forall(k) = \langle \Gamma, Q^\forall, q_0^\forall, F^\forall, \delta_\forall \rangle$ denote the universal deterministic Levenshtein automaton for bound k. We assume that we can access, for each symbol $\sigma \in \Sigma$ and each index $1 \leq i \leq m + k$, the characteristic vector $\vec{\chi}(\sigma, p_{i-k} \cdots p_i \cdots p_r)$, where $r = \min\{m, i + k + 1\}$, in constant time (cf. Section 4). We traverse the two automata $A^\forall(k)$ and A_D in parallel, using a standard backtracking procedure. At each step, a symbol σ read in A_D representing the ith symbol of the current dictionary path is translated into the bitvector $\vec{\chi}(\sigma, p_{i-k} \cdots p_i \cdots p_r)$, $r = \min\{m, i+k+1\}$, which is used as input for $A^\forall(k)$.

```
push (<0,ε,q_0^D,q_0^∀>);
while not empty(stack) do begin
   pop (<i,W,q^D,q^∀>);
   for σ in Σ do begin
      χ⃗ := χ⃗(σ,p_{i-k}⋯p_i⋯p_r);
      q_1^D := δ^D(q^D,σ);
      q_1^∀ := δ_∀(q^∀,χ⃗);
      if (q_1^D <> NIL) and (q_1^∀ <> NIL) then begin
         W_1 := concat(W,σ);
         push(<i+1,W_1,q_1^D,q_1^∀>);
         if (q_1^D ∈ F^D) and (q_1^∀ ∈ F^∀) then output(W_1);
      end;
   end;
end;
```

Starting with the pair of initial states $\langle q_0^D, q_0^\forall \rangle$, position $i = 0$, and the empty word ε, each step of the traversal adds a new symbol $\sigma \in \Sigma$ to the actual word W and leads from a pair of states $\langle q^D, q^\forall \rangle \in Q^D \times Q^\forall$ to $\langle \delta^D(q^D, \sigma), \delta_\forall(q^\forall, \vec{\chi}) \rangle$. We proceed as long as both components are distinct from the empty failure state,[1] NIL. Whenever a final state is reached in both automata, the actual word W is added to the output. It is trivial to show that the list of all output words represents exactly the set of all dictionary entries W such that $d_L(W, P) \leq k$.

The computational cost of the above algorithm is bounded by the size of the dictionary automaton A_D and depends on the bound k used. If k reaches the length of

1 A **failure state** is a state q whose language $\mathcal{L}_A(q)$ is empty.

the longest word in the dictionary, then in general (e.g., for the empty input word), the algorithm will result in a complete traversal of A_D. In practice, small bounds are used, and only a small portion of A_D will be visited. For bound 0, the algorithm validates in time $O(|P|)$ if the input pattern P is in the dictionary.

5.2 Evaluation Results for Basic Correction Algorithm

Experimental results were obtained using a Bulgarian lexicon (BL) with 965,339 word entries (average length 10.23 symbols), a German dictionary (GL) with 3,871,605 entries (dominated by compound nouns, average length 18.74 symbols), and a "lexicon" (TL) containing 1,200,073 bibliographic titles from the Bavarian National Library (average length 47.64 symbols). The German dictionary and the title dictionary are nonpublic. They were provided to us by Franz Guenthner and the Bavarian National Library, respectively, for the tests we conducted. The following table summarizes the dictionary automaton statistics for the three dictionaries:

	BL	GL	TL
Number of words	956,339	3,871,605	1,200,073
Automaton states	39,339	4,068,189	29,103,779
Automaton transitions	102,585	6,954,377	30,252,173
Size (bytes)	1,191,548	90,206,665	475,615,320

The basic correction algorithm was implemented in C and tested on a 1.6 GHz Pentium IV machine under Linux.

5.2.1 A Baseline. Before we present our evaluation results, we give a simplified baseline. Let a garbled word W be given. In order to find all words from the dictionary within Levenshtein distance k, we can use two simple methods:

1. For each dictionary word V, check whether $d_L(V, W) \leq k$.
2. For each string V such that $d_L(V, W) \leq k$, check whether V is in the dictionary.

We consider input words W of length 10. Visiting a state in an automaton takes about 0.1 μs. Using Method 1, the time needed to check whether $d_L(V, W) \leq k$ for a dictionary word V using the universal Levenshtein automaton can be estimated as 1 μs (a crude approximation). When using Method 2, we need about 1 μs for the dictionary lookup of a word with 10 symbols. Assume that the alphabet has 30 symbols. Given the input W, we have 639 strings within Levenshtein distance 1, about 400,000 strings within distance 2, and about 260,000,000 strings within distance 3. Assuming that the dictionary has 1,000,000 words, we get the following table of correction times:

	Distance 1	Distance 2	Distance 3
Method 1	1,000 ms	1,000 ms	1,000 ms
Method 2	0.639 ms	400 ms	260,000 ms

5.2.2 Correction with BL. To test the basic correction algorithm with the Bulgarian lexicon, we used a Bulgarian word list containing randomly introduced errors. In each word, we introduced between zero and four randomly selected symbol substitutions,

insertions, or deletions. The number of test words created for each length is shown in the following table:

Length	3	4	5	6	7	8
♯ words	3,563	11,066	27,196	53,763	90,202	128,620
Length	9	10	11	12	13	14
♯ words	155,888	163,318	148,879	117,783	81,481	50,291
Length	15	16	17	18	19	20
♯ words	28,481	15,079	8,048	4,350	2,526	1,422

Table 1 lists the results of the basic correction algorithm using BL and standard Levenshtein distance with bounds $k = 1, 2, 3$. Column 1 shows the length of the input words. Column 2 (CT1) describes the average time needed for the parallel traversal of the dictionary automaton and the universal Levenshtein automaton using Levenshtein distance 1. The time needed to output the correction candidates is always included; hence the column represents the total correction time. Column 3 (NC1) shows the average number of correction candidates (dictionary words within the given distance bound) per input word. (For $k = 1$, there are cases in which this number is below 1. This shows that for some of the test words, no candidates were returned: These words were too seriously corrupted for correction suggestions to be found within the given distance bound.) Similarly Columns 4 (CT2) and 6 (CT3) yield, respectively, the total correction times per word (averages) for distance bounds 2 and 3, and Columns 5 (NC2) and 7 (NC3) yield, respectively, the average number of correction candidates per word for distance bounds 2 and 3. Again, the time needed to output all corrections is included.

5.2.3 Correction with GL. To test the correction times when using the German lexicon, we again created a word list with randomly introduced errors. The number of test words of each particular length is shown in the following table:

Length	1–14	15–24	25–34	35–44	45–54	55–64
♯ words	100,000	100,000	100,000	9,776	995	514

The average correction times and number of correction candidates for GL are summarized in Table 2, which has the same arrangement of columns (with corresponding interpretations) as Table 1.

5.2.4 Correction with TL. To test the correction times when using the title "lexicon," we again created a word list with randomly introduced errors. The number of test words of each length is presented in the following table:

Length	1–14	15–24	25–34	35–44	45–54	55–64
♯ words	91,767	244,449	215,094	163,425	121,665	80,765

Table 3 lists the results for correction with TL and standard Levenshtein distance with bounds $k = 1, 2, 3$. The arrangement of columns is the same as for Table 1, with corresponding interpretations.

5.2.5 Summary. For each of the three dictionaries, evaluation times strongly depend on the tolerated number of edit operations. When fixing a distance bound, the length of the input word does not have a significant influence. In many cases, correction works faster for long input words, because the number of correction candidates decreases. The large number of entries in GL leads to increased correction times.

Table 1
Evaluation results for the basic correction algorithm, Bulgarian dictionary, standard Levenshtein distance, and distance bounds $k = 1, 2, 3$. Times in milliseconds.

Length	(CT1)	(NC1)	(CT2)	(NC2)	(CT3)	(NC3)
3	0.107	12.03	0.974	285.4	4.589	2983.2
4	0.098	8.326	1.048	192.1	5.087	2426.6
5	0.085	5.187	1.086	105.0	5.424	1466.5
6	0.079	4.087	0.964	63.29	5.454	822.77
7	0.079	3.408	0.853	40.95	5.426	466.86
8	0.081	3.099	0.809	30.35	5.101	294.84
9	0.083	2.707	0.824	22.36	4.631	187.12
10	0.088	2.330	0.794	16.83	4.410	121.73
11	0.088	1.981	0.821	12.74	4.311	81.090
12	0.088	1.633	0.831	9.252	4.277	51.591
13	0.089	1.337	0.824	6.593	4.262	31.405
14	0.089	1.129	0.844	4.824	4.251	19.187
15	0.089	0.970	0.816	3.748	4.205	12.337
16	0.087	0.848	0.829	3.094	4.191	9.1752
17	0.086	0.880	0.805	2.970	4.138	8.1250
18	0.087	0.809	0.786	2.717	4.117	7.1701
19	0.087	0.810	0.792	2.646	4.078	6.6544
20	0.091	0.765	0.795	2.364	4.107	5.7686

Table 2
Evaluation results for the basic correction algorithm, German dictionary, standard Levenshtein distance, and distance bounds $k = 1, 2, 3$. Times in milliseconds.

Length	(CT1)	(NC1)	(CT2)	(NC2)	(CT3)	(NC3)
1–14	0.225	0.201	4.140	0.686	23.59	2.345
15–24	0.170	0.605	3.210	1.407	19.66	3.824
25–34	0.249	0.492	4.334	0.938	24.58	1.558
35–44	0.264	0.449	4.316	0.781	24.06	1.187
45–54	0.241	0.518	3.577	0.969	20.18	1.563
55–64	0.233	0.444	3.463	0.644	19.03	0.737

Table 3
Evaluation results for the basic correction algorithm, title "lexicon," standard Levenshtein distance, and distance bounds $k = 1, 2, 3$. Times in milliseconds.

Length	(CT1)	(NC1)	(CT2)	(NC2)	(CT3)	(NC3)
1–14	0.294	0.537	3.885	2.731	19.31	24.67
15–24	0.308	0.451	4.024	0.872	19.50	1.703
25–34	0.321	0.416	4.160	0.644	19.98	0.884
35–44	0.330	0.412	4.225	0.628	20.20	0.844
45–54	0.338	0.414	4.300	0.636	20.44	0.857
55–64	0.344	0.347	4.340	0.433	20.61	0.449

6. Using Backwards Dictionaries for Filtering

In the related area of pattern matching in strings, various filtering methods have been introduced that help to find portions of a given text in which an approximate match of a given pattern P is *not* possible. (See Navarro [2001] and Navarro and Raffinot [2002] for surveys). In this section, we show how one general method of this form (Wu and Manber 1992; Myers 1994; Baeza-Yates and Navarro 1999; Navarro and Baeza-Yates 1999) can be adapted to approximate search in a dictionary, improving the basic correction algorithm.

For approximate text search, the crucial observation is the following: If the Levenshtein distance between a pattern P and a portion of text T' does not exceed a given bound k, and if we cut P into $k+1$ disjoint pieces $P_1, \ldots, P_{k+1}$, then T' must contain at least one piece. Hence the search in text T can be started with an exact multipattern search for $\{P_1, \ldots, P_{k+1}\}$, which is much faster than approximate search for P. When finding one of the pieces P_i in the text, the full pattern P is searched for (returning now to approximate search) within a small neighborhood around the occurrence. Generalizations of this idea rely on the following lemma (Myers 1994; Baeza-Yates and Navarro 1999; Navarro and Raffinot 2002):

Lemma 1
Let T' match P with $\leq k$ errors. Let P be represented as the concatenation of j words $P_1, \ldots, P_j$. Let $a_1, \ldots, a_j$ denote arbitrary integers, and define $A = \sum_{i=1}^{j} a_i$. Then, for some $i \in \{1, \ldots, j\}$, P_i matches a substring of T' with $\leq \lfloor a_i k / A \rfloor$ errors.[2]

In our experiments, which were limited to distance bounds $k = 1, 2, 3$, we used the following three instances of the general idea. Let P denote an input pattern, and let W denote an entry of the dictionary D. Assume we cut P into two pieces, representing it in the form $P = P_1 P_2$:

1. If $d_L(P, W) \leq 3$, then W can be represented in the form $W = W_1 W_2$, where we have the following mutually exclusive cases:
   (a) $d_L(P_1, W_1) = 0$ and $d_L(P_2, W_2) \leq 3$
   (b) $1 \leq d_L(P_1, W_1) \leq 3$ and $d_L(P_2, W_2) = 0$
   (c) $d_L(P_1, W_1) = 1$ and $1 \leq d_L(P_2, W_2) \leq 2$
   (d) $d_L(P_1, W_1) = 2$ and $d_L(P_2, W_2) = 1$
2. If $d_L(P, W) \leq 2$, then W can be represented in the form $W = W_1 W_2$, where we have the following mutually exclusive cases:
   (a) $d_L(P_1, W_1) = 0$ and $d_L(P_2, W_2) \leq 2$
   (b) $d_L(P_2, W_2) = 0$ and $1 \leq d_L(P_1, W_1) \leq 2$
   (c) $d_L(P_1, W_1) = 1 = d_L(P_2, W_2)$
3. If $d_L(P, W) \leq 1$, then W can be represented in the form $W = W_1 W_2$, where we have the following mutually exclusive cases:
   (a) $d_L(P_1, W_1) = 0$ and $d_L(P_2, W_2) \leq 1$
   (b) $d_L(P_1, W_1) = 1$ and $d_L(P_2, W_2) = 0$

2 As usual, $\lfloor r \rfloor$ denotes the largest integer $\leq r$.

In order to make use of these observations, we compute, given dictionary D, the backwards dictionary $D^{-R} := \{W^{-R} \mid W \in D\}$.[3] Dictionary D and backwards dictionary D^{-R} are compiled into deterministic finite-state automata A_D and $A_{D^{-R}}$, respectively. If the dictionary is infinite and directly given as a finite-state automaton A_D, the automaton $A_{D^{-R}}$ may be computed using standard techniques from formal language theory. Further steps depend on the bound k. We will describe only approximate search with bound $k = 3$; the methods for bounds $k = 1, 2$ are similar.

Let P denote the given pattern. P is cut into two pieces P_1, P_2 of approximately the same length. We compute P_2^{-R} and P_1^{-R}. We then start four subsearches, corresponding to Cases (1a)–(1d) specified above.

For subsearch (1a), we first traverse A_D using input P_1. Let q denote the state that is reached. Starting from q and the initial state $\{I^0\}$ of $A^\forall(3)$, we continue with a parallel traversal of A_D and $A^\forall(3)$. Transition symbols in A_D are translated into input bitvectors for $A^\forall(3)$ by matching them against appropriate subwords of $\$\$\$P_2$, as described in Section 5. The sequence of all transition labels of the actual paths in A_D is stored as usual. Whenever we reach a pair of final states, the current sequence—which includes the prefix P_1—is passed to the output. Clearly, each output sequence has the form P_1P_2', where $d_L(P_2, P_2') \leq 3$. Conversely, any dictionary word of this form is found using subsearch (1a).

For subsearch (1b), we first traverse $A_{D^{-R}}$ using P_2^{-R}. Let q denote the state that is reached. Starting from q and the initial state $\{I^0\}$ of $A^\forall(3)$, we continue with a parallel traversal of $A_{D^{-R}}$ and $A^\forall(3)$. Transition symbols in $A_{D^{-R}}$ are translated into input bitvectors for $A^\forall(3)$ by matching them against appropriate subwords of $\$\$\$P_1^{-R}$. Whenever we reach a pair of final states, the inversed sequence is passed to the output. Clearly, each output sequence has the form $P_1'P_2$, where $d_L(P_1, P_1') \leq 3$. Conversely, any dictionary word of this form is found using a search of this form. For a given output, a closer look at the final state S that is reached in $A^\forall(3)$ may be used to exclude cases in which $P_1 = P_1'$. (Simple details are omitted).

For subsearch (1c), we start with a parallel traversal of A_D and $A^\forall(1)$. Transition symbols in A_D are translated into input bitvectors for $A^\forall(1)$ by matching them against appropriate subwords of $\$P_1$. For each pair of states (q, S) that are reached, where S represents a final state of $A^\forall(1)$, we start a parallel traversal of A_D and $A^\forall(2)$, departing from q and the initial state $\{I^0\}$ of $A^\forall(2)$. Transition symbols in A_D are translated into input bitvectors for $A^\forall(2)$ by matching them against appropriate subwords of $\$\$P_2$. Whenever we reach a pair of final states, the current sequence is passed to the output. Clearly, each output sequence has the form $P_1'P_2'$, where $d_L(P_1, P_1') \leq 1$ and $d_L(P_2, P_2') \leq 2$. Conversely, any dictionary word of this form is found using a search of this form. A closer look at the final states that are respectively reached in $A^\forall(1)$ and $A^\forall(2)$ may be used to exclude cases in which $P_1 = P_1'$ or $P_2 = P_2'$. (Again, simple details are omitted).

For subsearch (1d), we start with a parallel traversal of $A_{D^{-R}}$ and $A^\forall(1)$. Transition symbols in $A_{D^{-R}}$ are translated into input bitvectors for $A^\forall(1)$ by matching them against appropriate subwords of $\$P_2^{-R}$. For each pair of states (q, S) that are reached, where S represents a final state of $A^\forall(1)$, we start a parallel traversal of $A_{D^{-R}}$ and $A^\forall(2)$, departing from q and the initial state $\{I^0\}$ of $A^\forall(2)$. Transition symbols in $A_{D^{-R}}$ are translated into input bitvectors for $A^\forall(2)$ by matching them against appropriate subwords of $\$\$P_1^{-R}$. Whenever we reach a pair of final states, the inversed sequence is passed to the output. Clearly, each output sequence has the form $P_1'P_2'$,

3 W^{-R} denotes the reverse of W.

where $d_L(P_1, P_1') \leq 2$ and $d_L(P_2, P_2') \leq 1$. Conversely, any word in the dictionary of this form is found using a search of this form. A closer look at the final states that are reached in $A^{\forall}(1)$ and $A^{\forall}(2)$ may be used to exclude cases where $P_2 = P_2'$ or $d_L(P_1, P_1') \leq 1$, respectively. (Again, simple details are omitted).

It should be noted that the output sets obtained from the four subsearches (1a)–(1d) are not necessarily disjoint, because a dictionary entry W may have more than one partition $W = W_1W_2$ of the form described in cases (1a)–(1d).

6.1 Evaluation Results

The following table summarizes the statistics of the automata for the three backwards dictionaries:

	BL	GL	TL
Number of words	956,339	3,871,605	1,200,073
Automaton states	54,125	4,006,357	29,121,084
Automaton transitions	183,956	7,351,973	30,287,053
Size (bytes)	2,073,739	92,922,493	475,831,001

Note that the size of the backwards-dictionary automata is approximately the same as the size of the dictionary automata.

Tables 4, 5, and 6 present the evaluation results for the backwards dictionary filtering method using dictionaries BL, GL, and TL, respectively. We have constructed additional automata for the backwards dictionaries.

For the tests, we used the same lists of input words as in Section 5.2 in order to allow a direct comparison to the basic correction method. Dashes indicate that the correction times were too small to be measured with sufficient confidence in their level of precision. In columns 3, 5, and 7, we quantify the speedup factor, that is, the ratio of the time taken by the basic algorithm to that taken by the backwards-dictionary filtering method.

6.2 Backwards-Dictionary Method for Levenshtein Distance with Transpositions

Universal Levenshtein automata can also be constructed for the modified Levenshtein distance, in which character transpositions count as a primitive edit operation, along with insertions, deletions, and substitutions. This kind of distance is preferable when correcting typing errors. A generalization of the techniques presented by the authors (Schulz and Mihov 2002) for modified Levenshtein distances—using either transpositions or merges and splits as additional edit operations—has been described in Schulz and Mihov (2001). It is assumed that all edit operations are applied in parallel, which implies, for example, that insertions between transposed letters are not possible.

If we want to apply the filtering method using backwards dictionaries for the modified Levenshtein distance $d_L'(P, W)$ with transpositions, we are faced with the following problem: Assume that the pattern $P = a_1a_2 \ldots a_ma_{m+1} \ldots a_n$ is split into $P_1 = a_1a_2 \ldots a_m$ and $P_2 = a_{m+1}a_{m+2} \ldots a_n$. When we apply the above procedure, the case in which a_m and a_{m+1} are transposed is not covered. In order to overcome this problem, we can draw on the following observation:

If $d_L'(P, W) \leq 3$, then W can be represented in the form $W = W_1W_2$, where there are seven alternatives, inlcuding the following four:

1. $d_L'(P_1, W_1) = 0$ and $d_L'(P_2, W_2) \leq 3$
2. $1 \leq d_L'(P_1, W_1) \leq 3$ and $d_L'(P_2, W_2) = 0$

Table 4
Evaluation results using the backwards-dictionary filtering method, Bulgarian dictionary, and distance bounds $k = 1, 2, 3$. Times in milliseconds and speedup factors (ratio of times) with respect to basic algorithm.

Length	(CT1)	Speedup 1	(CT2)	Speedup 2	(CT3)	Speedup 3
3	0.031	3.45	0.876	1.11	6.466	0.71
4	0.027	3.63	0.477	2.20	4.398	1.16
5	0.018	4.72	0.450	2.41	2.629	2.06
6	0.016	4.94	0.269	3.58	2.058	2.65
7	0.011	7.18	0.251	3.40	1.327	4.09
8	0.012	6.75	0.196	4.13	1.239	4.12
9	0.009	9.22	0.177	4.66	0.828	5.59
10	0.010	8.80	0.159	4.99	0.827	5.33
11	0.008	11.0	0.147	5.59	0.603	7.15
12	0.008	11.0	0.142	5.85	0.658	6.50
13	0.006	14.8	0.128	6.44	0.457	9.33
14	0.006	14.8	0.123	6.86	0.458	9.28
15	0.005	17.8	0.112	7.29	0.321	13.1
16	0.005	17.4	0.111	7.47	0.320	13.1
17	0.005	17.2	0.108	7.45	0.283	14.6
18	0.005	17.4	0.108	7.28	0.280	14.7
19	0.004	21.8	0.103	7.69	0.269	15.2
20	—	—	0.105	7.57	0.274	15.0

Table 5
Evaluation results using the backwards-dictionary filtering method, German dictionary, and distance bounds $k = 1, 2, 3$. Times in milliseconds and speedup factors (ratio of times) with respect to basic algorithm.

Length	(CT1)	Speedup 1	(CT2)	Speedup 2	(CT3)	Speedup 3
1–14	0.007	32.1	0.220	18.8	0.665	35.5
15–24	0.010	17.0	0.175	18.3	0.601	32.7
25–34	0.009	27.7	0.221	19.6	0.657	37.4
35–44	0.007	37.7	0.220	19.6	0.590	40.8
45–54	—	—	0.201	17.8	0.452	44.6
55–64	—	—	0.195	17.8	0.390	48.8

3. $d'_L(P_1, W_1) = 1$ and $1 \leq d'_L(P_2, W_2) \leq 2$
4. $d'_L(P_1, W_1) = 2$ and $d'_L(P_2, W_2) = 1$

In the remaining three alternatives, $W_1 = W'_1 a_{m+1}$ ends with the symbol a_{m+1}, and $W_2 = a_m W'_2$ starts with a_m. For $P'_1 := a_1 a_2 \ldots a_{m-1}$ and $P'_2 := a_{m+2} a_{m+3} \ldots a_n$, we have the following three alternatives:

5. $d'_L(P'_1, W'_1) = 0$ and $d_L(P'_2, W'_2) \leq 2$
6. $d'_L(P'_1, W'_1) = 1$ and $d'_L(P'_2, W'_2) \leq 1$
7. $d'_L(P'_1, W'_1) = 2$ and $d'_L(P'_2, W'_2) = 0$

The cases for distance bounds $k = 1, 2$ are solved using similar extensions of the original subcase analysis. In each case, it is straightforward to realize a search procedure with subsearches corresponding to the new subcase analysis, using an ordinary dictionary and a backwards dictionary, generalizing the above ideas.

Table 6
Evaluation results using the backwards-dictionary filtering method, title "lexicon," and distance bounds $k = 1, 2, 3$. Times in milliseconds and speedup factors (ratio of times) with respect to basic algorithm.

Length	(CT1)	Speedup 1	(CT2)	Speedup 2	(CT3)	Speedup 3
1–14	0.032	9.19	0.391	9.94	1.543	12.5
15–24	0.019	16.2	0.247	16.3	0.636	30.7
25–34	0.028	11.5	0.260	16.0	0.660	30.3
35–44	0.029	11.4	0.295	14.3	0.704	28.7
45–54	0.037	9.14	0.332	13.0	0.759	26.9
55–64	0.038	9.05	0.343	12.7	0.814	25.3

Table 7
Evaluation results using the backwards-dictionary filtering method for the modified Levenshtein distance d'_L with transpositions, for German dictionary and title "lexicon," distance bound $k = 3$. Times in milliseconds and speedup factors (ratio of times) with respect to basic algorithm.

Length	(CT3 GL)	Speedup GL	(CT3 TL)	Speedup TL
1–14	1.154	23.0	2.822	7.7
15–24	1.021	21.3	1.235	17.5
25–34	1.148	23.7	1.261	17.7
35–44	1.096	24.3	1.283	17.6
45–54	0.874	25.5	1.326	17.3
55–64	0.817	25.7	1.332	17.4

We have tested the new search procedure for the modified Levenshtein distance d'_L. In Table 7 we present the experimental results with the German dictionary and the title "lexicon" for distance bound $k = 3$.

6.2.1 Summary. The filtering method using backwards dictionaries drastically improves correction times. The increase in speed depends both on the length of the input word and on the error bound. The method works particularly well for long input words. For GL, a drastic improvement can be observed for all subclasses. In contrast, for very short words of BL, only a modest improvement is obtained. When using BL and the modified Levenshtein distance d'_L with transpositions, the backwards-dictionary method improved the basic search method only for words of length ≥ 9. For short words, a large number of repetitions of the same correction candidates was observed. The analysis of this problem is a point of future work.

Variants of the backwards-dictionary method also can be used for the Levenshtein distance d''_L, in which insertions, deletions, substitutions, merges, and splits are treated as primitive edit operations. Here, the idea is to split the pattern at two neighboring positions, which doubles the number of subsearches. We did not evaluate this variant.

7. Using Dictionaries with Single Deletions for Filtering

The final technique that we describe here is again an adaptation of a filtering method from pattern matching in strings (Muth and Manber 1996; Navarro and Raffinot 2002). When restricted to the error bound $k = 1$, this method is very efficient. It can be used only for finite dictionaries. Assume that the pattern $P = p_1 \dots p_m$ matches a portion of text, T', with one error. Then $m - 1$ letters of P are found in T' in the correct order.

This fact can be used to compute $m+1$ derivatives of P that are compared with similar derivatives of a window T' of length m that is slid over the text. A derivative of a word V can be V or a word that is obtained by deleting exactly one letter of V. Coincidence between derivatives of P and T' can be used to detect approximate matches of P of the above form. For details we refer to Navarro and Raffinot (2002). In what follows we describe an adaptation of the method to approximate search of a pattern P in a dictionary D.

Let i be an integer. With $V^{[i]}$ we denote the word that is obtained from a word V by deleting the ith symbol of V. For $|V| < i$, we define $V^{[i]} = V$. By a **dictionary with output sets**, we mean a list of strings W, each of which is associated with a set of output strings $O(W)$. Each string W is called a **key**. Starting from the conventional dictionary D, we compute the following dictionaries with output sets D_{all}, D_1, D_2, ..., D_{n_0}, where n_0 is the maximal length of an entry in D:

- The set of keys of D_{all} is $D \cup \{V \mid \exists i \geq 1, W \in D \text{ such that } V = W^{[i]}\}$. The output set for key V is $O_{all}(V) := \{W \in D \mid W = V \vee V = W^{[i]} \text{ for some } i \geq 1\}$.
- The set of keys for D_i is $D \cup \{V \mid \exists W \in D \text{ such that } V = W^{[i]}\}$. The output set for a key V is $O_i(V) := \{W \in D \mid W = V \vee V = W^{[i]}\}$.

Lemma 2
Let P denote a pattern, and let $W \in D$. Then $d_L(P, W) \leq 1$ iff either $W \in O_{all}(P)$ or there exists i, $1 \leq i \leq |P|$, such that $W \in O_i(P^{[i]})$.

The proof of the lemma is simple and has therefore been omitted.

In our approach, the dictionaries with output sets D_{all}, D_1, D_2, ..., D_{n_0} are compiled, respectively, into minimal subsequential transducers $A_{all}, A_1, A_2, \ldots, A_{n_0}$. Given a pattern P, we compute the union of the output sets $O_{all}(P), O_1(P^{[1]}), \ldots, O_{|P|}(P^{[|P|]})$ using these transducers. It follows from Lemma 2 that we obtain as result the set of all entries W of D such that $d_L(P, W) \leq 1$. It should be noted that the output sets are not necessarily disjoint. For example, if P itself is a dictionary entry, then $P \in O_i(P^{[i]})$ for all $1 \leq i \leq |P|$.

After we implemented the above procedure for approximate search, we found that a similar approach based on hashing had been described as early as 1981 in a technical report by Mor and Fraenkel (1981).

7.1 Evaluation Results

Table 8 presents the evaluation results for edit distance 1 using dictionaries with single deletions obtained from BL. The total size of the constructed single-deletion dictionary automata is 34.691 megabytes. The word lists used for tests are those described in Section 5.2. GL and TL are not considered here, since the complete system of subdictionaries needed turned out to be too large. For a small range of input words of length 3–6, filtering using dictionaries with single deletions behaves better than filtering using the backwards-dictionary method.

8. Similarity Keys

A well-known technique for improving lexical search not mentioned so far is the use of similarity keys. A **similarity key** is a mapping κ that assigns to each word W a simplified representation $\kappa(W)$. Similarity keys are used to group dictionaries into classes

of "similar" entries. Many concrete notions of "similarity" have been considered, depending on the application domain. Examples are phonetic similarity (e.g., SOUNDEX system; cf. Odell and Russell [1918, 1922] and Davidson [1962]), similarity in terms of word shape and geometric form (e.g., "envelope representation" [Sinha 1990; Anigbogu and Belaid 1995]) or similarity under n-gram analysis (Angell, Freund, and Willett 1983; Owolabi and McGregor 1988). In order to search for a pattern P in the dictionary, the "code" $\kappa(P)$ is computed. The dictionary is organized in such a way that we may efficiently retrieve all regions containing entries with code (similar to) $\kappa(P)$. As a result, only small parts of the dictionary must be visited, which speeds up search. Many variants of this basic idea have been discussed in the literature (Kukich 1992; Zobel and Dart 1995; de Bertrand de Beuvron and Trigano 1995).

In our own experiments we first considered the following simple idea. Given a similarity key κ, each entry W of dictionary D is equipped with an additional prefix of the form $\kappa(W)$&. Here & is a special symbol that marks the border between codes and original words. The enhanced dictionary $\hat{D}$ with all entries of the form $\kappa(W)$&W is compiled into a deterministic finite-state automaton $A_{\hat{D}}$. Approximate search for pattern P in D is then reorganized in the following way. The enhanced pattern $\kappa(P)$&P is used for search in $A_{\hat{D}}$. We distinguish two phases in the backtracking process. In Phase 1, which ends when the special symbol & is read, we compute an initial path of $A_{\hat{D}}$ in which the corresponding sequence of transition labels represents a code α such that $d_L(\kappa(P), \alpha) \leq k$. All paths of this form are visited. Each label sequence α& of the above form defines a unique state q of the automaton $A_{\hat{D}}$ such that $\mathcal{L}_{A_{\hat{D}}}(q) = \{W \in D \mid \kappa(W) = \alpha\}$. In Phase 2, starting from q, we compute all entries W with code $\kappa(W) = \alpha$ such that $d_L(W, P) \leq k$. In both phases the automaton $A^{\forall}(k)$ is used to control the search, and transition labels of $A_{\hat{D}}$ are translated into characteristic vectors. In order to guarantee completeness of the method, the distance between codes of a pair of words should not exceed the distance between the words themselves.

It is simple to see that in this method, the backtracking search is automatically restricted to the subset of all dictionary entries V such that $d_L(\kappa(V), \kappa(P)) \leq k$. Unfortunately, despite this, the approach does not lead to reduced search times. A closer look at the structure of (conventional) dictionary automata A_D for large dictionaries D shows that there exists an enormous number of distinct initial paths of A_D of length 3–5. During the controlled traversal of A_D, most of the search time is spent visiting paths of this initial "wall." Clearly, most of these paths do not lead to any correction candidate. Unfortunately, however, these "blind" paths are recognized too late. Using the basic method described in Section 5, we have to overcome one single wall in A_D for the whole dictionary. In contrast, when integrating similarity keys in the above form, we have to traverse a similar wall for the subdictionary $D_{\kappa(W)} := \{V \in D \mid \kappa(V) = \kappa(W)\}$ for each code $\kappa(W)$ found in Phase 1. Even if the sets $D_{\kappa(W)}$ are usually much smaller than D, the larger number of walls that are visited leads to increased traversal times.

As an alternative, we tested a method in which we attached to each entry W of D *all* prefixes of the form α&, where α represents a possible code such that $d_L(\kappa(W), \alpha) \leq k$. Using a procedure similar to the one described above, we have to traverse only one wall in Phase 2. With this method, we obtained a reduction in search time. However, with this approach, enhanced dictionaries $\hat{D}$ are typically much larger than original dictionaries D. Hence the method can be used only if both dictionary D and bound k are not too large and if the key is not too fine. Since the method is not more efficient than filtering using backwards dictionaries, evaluation results are not presented here.

Table 8
Results for BL, using dictionaries with single deletions for filtering, distance bound $k = 1$. Times in milliseconds and speedup factors (ratio of times) with respect to basic algorithm.

Length	(CT1)	Speedup 1	Length	(CT1)	Speedup 1
3	0.011	9.73	12	0.026	3.38
4	0.011	8.91	13	0.028	3.18
5	0.010	8.50	14	0.033	2.70
6	0.011	7.18	15	0.035	2.54
7	0.013	6.08	16	0.039	2.23
8	0.015	5.40	17	0.044	1.95
9	0.017	4.88	18	0.052	1.67
10	0.020	4.40	19	0.055	1.58
11	0.022	4.00	20	0.063	1.44

9. Concluding Remarks

In this article, we have shown how filtering methods can be used to improve finite-state techniques for approximate search in large dictionaries. As a central contribution we introduced a new correction method, filtering based on backwards dictionaries and partitioned input patterns. Though this method generally leads to very short correction times, we believe that correction times could possibly be improved further using refinements and variants of the method or introducing other filtering methods. There are, however, reasons to assume that we are not too far from a situation in which further algorithmic improvements become impossible for fundamental reasons. The following considerations show how an "optimal" correction time can be estimated that cannot be improved upon without altering the hardware or using faster access methods for automata.

We used a simple backtracking procedure to realize a complete traversal of the dictionary automaton A_D. During a first traversal, we counted the total number of visits to any state. Since A_D is not a tree, states may be passed through several times during the complete traversal. Each such event counts as one visit to a state. The ratio of the number of visits to the total number of symbols in the list of words D gives the average number of visits per symbol, denoted v_0. In practice, the value of v_0 depends on the compression rate that is achieved when compiling D into the automaton A_D. It is smaller than 1 because of numerous prefixes of dictionary words that are shared in A_D. We then used a second traversal of A_D—not counting visits to states—to compute the total traversal time. The ratio of the total traversal time to the number of visits yields the time t_0 that is needed for a single visit. For the three dictionaries, the following values were obtained:

	BL	GL	TL
Average number v_0 of visits per symbol	0.1433	0.3618	0.7335
Average time t_0 for one visit (in μs)	0.0918	0.1078	0.0865

Given an input V, we may consider the total number n_V of symbols in the list of correction candidates. Then $n_V \cdot v_0 \cdot t_0$ can be used to estimate the optimal correction time for V. In fact, in order to achieve this correction time, we need an oracle that knows how to avoid any kind of useless backtracking. Each situation in which we proceeded on a dictionary path that does not lead to a correction candidate for V would require some extra time that is not included in the above calculation. From another point of view, the above idealized algorithm essentially just copies the correction candidates into a resulting destination. The time that is consumed is proportional to the sum of the length of the correction candidates.

For each of the three dictionaries, we estimated the optimal correction time for one class of input words. For BL we looked at input words of length 10. The average number of correction candidates for Levenshtein distance 3 is 121.73 (cf. Table 1). Assuming that the average length of correction candidates is 10, we obtain a total of 1,217.3 symbols in the complete set of all correction candidates. Hence the optimal correction time is approximately

$$1217.3 \cdot 0.0000918 \text{ ms } \cdot 0.1433 = 0.016 \text{ ms}$$

The actual correction time using filtering with the backwards-dictionary method is 0.827 ms, which is 52 times slower.

For GL, we considered input words of length 15–24 and distance bound 3. We have on average 3.824 correction candidates of length 20, that is, 76.48 symbols. Hence the optimal correction time is approximately

$$76.48 \cdot 0.0001078 \text{ ms } \cdot 0.3618 = 0.003 \text{ ms}$$

The actual correction time using filtering with the backwards-dictionary method is 0.601 ms, which is 200 times slower.

For TL, we used input sequences of length 45–54 and again distance bound 3. We have on average 0.857 correction candidates of length 50, that is, 42.85 symbols. Hence the optimal correction time is approximately

$$42.85 \cdot 0.0000865 \text{ ms } \cdot 0.7335 = 0.003 \text{ ms}$$

The actual correction time using filtering with the backwards-dictionary method is 0.759 ms, which is 253 times slower.

These numbers coincide with our basic intuition that further algorithmic improvements are simpler for dictionaries with long entries. For example, variants of the backwards-dictionary method could be considered in which a finer subcase analysis is used to improve filtering.

Acknowledgments

This work was funded by a grant from VolkswagenStiftung. The authors thank the anonymous referees for many suggestions that helped to improve the presentation.

References

Angell, Richard C., George E. Freund, and Peter Willett. 1983. Automatic spelling correction using a trigram similarity measure. *Information Processing and Management*, 19:255–261.

Anigbogu, Julain C. and Abdel Belaid. 1995. Hidden Markov models in text recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 9(6):925–958.

Baeza-Yates, Ricardo A. and Gonzalo Navarro. 1998. Fast approximative string matching in a dictionary. In R. Werner, editor, *Proceedings SPIRE'98*, pages 14–22. IEEE Computer Science.

Baeza-Yates, Ricardo A. and Gonzalo Navarro. 1999. Faster approximate string matching. *Algorithmica*, 23(2):127–158.

Blair, Charles R. 1960. A program for correcting spelling errors. *Information and Control*, 3:60–67.

Bunke, Horst. 1993. A fast algorithm for finding the nearest neighbor of a word in a dictionary. In *Proceedings of the Second International Conference on Document Analysis and Recognition (ICDAR'93)*, pages 632–637, Tsukuba, Japan.

Daciuk, Jan, Stoyan Mihov, Bruce W. Watson, and Richard E. Watson. 2000. Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1):3–16.

Davidson, Leon. 1962. Retrieval of misspelled names in an airline passenger record system. *Communications of the ACM*, 5(3):169–171.

de Bertrand de Beuvron, Francois and Philippe Trigano. 1995. Hierarchically coded lexicon with variants. *International Journal of Pattern Recognition and Artificial Intelligence*, 9(1):145–165.

Dengel, Andreas, Rainer Hoch, Frank Hönes, Thorsten Jäger, Michael Malburg, and Achim Weigel. 1997. Techniques for improving OCR results. In Horst Bunke and Patrick S. P. Wang, editors, *Handbook of Character Recognition and Document Image Analysis*, 227–258. World Scientific.

Hopcroft, John E. and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.

Kim, Jong Yong and John Shawe-Taylor. 1992. An approximate string-matching algorithm. *Theoretical Computer Science*, 92:107–117.

Kim, Jong Yong and John Shawe-Taylor. 1994. Fast string matching using an *n*-gram algorithm. *Software—Practice and Experience*, 94(1):79–88.

Kozen, Dexter C. 1997. *Automata and Computability*. Springer.

Kukich, Karen. 1992. Techniques for automatically correcting words in texts. *ACM Computing Surveys*, 24:377–439.

Levenshtein, Vladimir I. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics-Doklady*, 10:707–710.

Mihov, Stoyan and Denis Maurel. 2001. Direct construction of minimal acyclic subsequential transducers. In S. Yu and A. Pun, editors, *Implementation and Application of Automata: Fifth International Conference (CIAA'2000)* (Lecture Notes in Computer Science no. 2088), pages 217–229. Springer.

Mor, Moshe and Aviezri S. Fraenkel. 1981. A hash code method for detecting and correcting spelling errors. Technical Report CS81-03, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel.

Muth, Robert and Udi Manber. 1996. Approximate multiple string search. In *Proceedings of the Seventh Annual Symposium on Combinatorical Pattern Matching* (Lecture Notes in Computer Science, no. 1075) pages 75–86. Springer.

Myers, Eugene W. 1994. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374.

Navarro, Gonzalo. 2001. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88.

Navarro, Gonzalo and Ricardo A. Baeza-Yates. 1999. Very fast and simple approximate string matching. *Information Processing Letters*, 72:65–70.

Navarro, Gonzalo and Mathieu Raffinot. 2002. *Flexible Pattern Matching in Strings*. Cambridge University Press, Cambridge.

Odell, Margaret K. and Robert C. Russell. 1918. U.S. Patent Number 1,261,167. U.S. Patent Office, Washington, DC.

Odell, Margaret K. and Robert C. Russell. 1992. U.S. Patent Number 1,435,663. U.S. Patent Office, Washington, DC.

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

Oommen, B. John and Richard K. S. Loke. 1997. Pattern recognition of strings with substitutions, insertions, deletions, and generalized transpositions. *Pattern Recognition*, 30(5):789–800.

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

Riseman, Edward M. and Roger W. Ehrich. 1971. Contextual word recognition using binary digrams. *IEEE Transactions on Computers*, C-20(4):397–403.

Schulz, Klaus U. and Stoyan Mihov. 2001. Fast string correction with Levenshtein-automata. Technical Report 01-127, Centrum für Informations= und sprachverarbeitung, University of Munich.

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

Seni, Giovanni, V. Kripasundar, and Rohini K. Srihari. 1996. Generalizing edit distance to incorporate domain information: Handwritten text recognition as a case study. *Pattern Recognition*, 29(3):405–414.

Sinha, R. M. K. 1990. On partitioning a dictionary for visual text recognition. *Pattern Recognition*, 23(5):497–500.

Srihari, Sargur N. 1985. *Computer Text Recognition and Error Correction*. Tutorial. IEEE Computer Society Press, Silver Spring, MD.

Srihari, Sargur N., Jonathan J. Hull, and Ramesh Choudhari. 1983. Integrating diverse knowledge sources in text recognition. *ACM Transactions on Office Information Systems*, 1(1):68–87.

Stephen, Graham A. 1994. *String Searching Algorithms*. World Scientific, Singapore.

Takahashi, Hiroyasu, Nobuyasu Itoh, Tomio Amano, and Akio Yamashita. 1990. A spelling correction method and its application to an OCR system. *Pattern Recognition*, 23(3/4):363–377.

Ukkonen, Esko. 1985a. Algorithms for approximate string matching. *Information and Control*, 64:100–118.

Ukkonen, Esko. 1985b. Finding approximate patterns in strings. *Journal of Algorithms*, 6(1–3):132–137.

Ukkonen, Esko. 1992. Approximate string-matching with *q*-grams and maximal matches. *Theoretical Computer Science*, 92:191–211.

Ullman, Jeffrey R. 1977. A binary *n*-gram technique for automatic correction of substitution, deletion, insertion and reversal errors. *Computer Journal*, 20(2):141–147.

Wagner, Robert A. and Michael J. Fischer. 1974. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173.

Weigel, Achim, Stephan Baumann, and J. Rohrschneider. 1995. Lexical postprocessing by heuristic search and automatic determination of the edit costs. In *Proceedings of the Third International Conference on Document Analysis and Recognition (ICDAR 95)*, pages 857–860.

Wells, C. J., L. J. Evett, Paul E. Whitby, and R.-J. Withrow. 1990. Fast dictionary look-up for contextual word recognition. *Pattern Recognition*, 23(5):501–508.

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

Zobel, Justin and Philip Dart. 1995. Finding approximate matches in large lexicons. *Software—Practice and Experience*, 25(3):331–345.