Efficient Techniques for Approximate Record Matching

1st draft version - documentation of work done during a 1-month stay in Sofia, Bulgaria
(Project BIS-21++)

Uli Reffle

May 5, 2006

1 Introduction

In this report we face a task where we receive as input a query string (in the following often called *search pattern*) that can be tokenized into some atomic tokens. This list of tokens is to be searched in a huge collection of records (multi-token expressions) of some kind. The idea is to find matches of the search pattern even if it contains spelling errors and the word order does not correspond in the pattern and the stored record. Hereby, the structure of the records can be taken into consideration to specify which permutations of tokens shall be allowed.

Many of the fastest algorithms for approximate search use finite state technology. In [MS04] Stoyan Mihov and Klaus Schulz describe a very efficient way of computing all entries of a huge dictionary whose Levenshtein distance $d_L$to a search pattern does not exceed a fixed bound k. The general idea of this algorithm is to provide the dictionary $D \subseteq \Sigma^*$ (Σ being a finite alphabet of characters) in the form of a determistic finite state automaton A_D with $\mathcal{L}(A_D) = D$. Mihov and Schulz introduce the concept of Universal Levenshtein Automata : these finite state automata can be used to very efficiently recognize all strings s with $d_L(s, p) \leq k$ for a specific distance k and pattern p. A parallel traversal of A_D and this automaton (the intersection of both their languages) computes the desired result: All words w with $w \in D$ and $d_L(w, p) \leq k$. The automaton is universal in the sense that it can be applied to search any pattern without re-constructing or changing the automaton. Instead the pattern is used to calculate *characteristic vectors* for each $c \in \Sigma$ - those characteristic vectors serve as input to the universal automaton.

[Ref05] addresses a modification of the search problem just described. Here the dictionary D is set up of *multi-token expressions*, finite sequences of words in Σ^*. The search pattern $p = (p_1, .., p_m)$ is also a multi-token expression. The task is to find this pattern in D, allowing k spelling errors for each token and free permutation among all tokens. The algorithm presented in [Ref05] and shortly sketched in section 3 is the foundation

of the ideas described in this report. On its basis we present ways to process more advanced search tasks where the permutation of tokens can be restricted in various ways. The corresponding modifications of the base algorithm are described in section 4.

2 Approximate search with multi-token expressions

The first method presented here covers approximate matching in a dictionary D of multi-token expressions without permutation. That is, an expression $d = (d_1, .., d_m) \in D$ is called a match for pattern $p = (p_1, .., p_m)$ iff each token d_i is a (perhaps) misspelled version of p_i with at most k errors. For a search pattern p, the result set R_p of matched expressions is:
$R_p := \{d \in D \mid \forall d_{i=1..m} : (d_L(d_i, p_i) \leq k)\}$

Aside from its immediate application, the method described in the following serves for two other purposes connected to search modulo permutation. First, it defines a baseline that allows us to determine the overhead produced by allowing permutations. Second, it turns out that this algorithm can also be used for search modulo permutation if D is represented in a special way explained in section 3.1.

For the computation of the result set R an extended version of the parallel traversal algorithm in [MS04] is used. A more elaborate description of this algorithm and the technical details of the universal levenshtein automata can not be given here, so let us assume we have at our disposal a universal levenshtein automaton LEV of fixed degree k, its input be characteristic input vectors $charvecs_i[c]$ pre-computed from all symbols $c \in \Sigma$ and each of the tokens p_i of the pattern.
D is represented by a finite state automaton $A_D = (\Sigma \cup \{\nu\}, Q_D, i_D, F_D, \delta_D)$. Its alphabet is enriched with a delimiter symbol $\nu \notin \Sigma$ which is used to separate the tokens in the FSA representation of D. $\mathcal{L}(A_D) = \{d_1 \circ \nu \circ ... \circ \nu \circ d_{|d|} \circ \nu \mid d \in D\}$[1]

2.1 Traversal procedure

During the traversal of A_D we perform various sub-tasks, namely approximate searches for one of the tokens p_i in certain parts of A_D. At first a path has to be found beginning at i_D that approximately matches the first token p_1 of the pattern. This is achieved by triggering a parallel traversal of A_D (starting at $pos_D = i_D$) and the universal levenshtein automaton LEV (starting at $pos_{LEV} = i_{LEV}$) using the set of characteristic vectors $charvecs_1$. Whenever there are transitions $(pos_D, c \in \Sigma, pos'_D) \in \delta_D$ and $(pos_{LEV}, charvecs_0[c \in \Sigma], pos'_{LEV}) \in \delta_{LEV}$, the search for p_0 is continued recursively from pos'_D and pos'_{LEV}.

In the original algorithm, an occurence is reported whenever the current states both

[1] Note that also the end of the expression is marked by a delimiter symbol ν

function search_noperm(queryString)

1: $A_D := (\Sigma \cup \{,\}, Q_D, \delta_D, i_D, F_D)$, FSA with $\mathcal{L}(A_D)$ = all database entries in original order
2: $LEV :=$ universal Levenshtein-automaton for distinct distance k with i_{LEV} as initial state and F_{LEV} as set of final states
3: **for** each token $t_{i:=0..m-1}$ of query **do**
4: $\quad charvecs_i := calcCharvecs(t_i)$
5: **end for**
6: $noperm\_recursive(i_D, 0, i_{LEV})$

function noperm_recursive(dbPos, curToken, levPos)

1: **for all** $c \in \Sigma$ **do**
2: $\quad newDbPos := \delta_D(dbPos, c)$
3: $\quad$ **if** $newDbPos \neq FAIL$ **then**
4: $\quad\quad newLevPos := \delta_{LEV}(levPos, charvecs_t[c])$
5: $\quad\quad$ **if** $newLevPos \neq FAIL$ **then**
6: $\quad\quad\quad noperm\_recursive(newDbPos, curToken, newLevPos)$
7: $\quad\quad$ **end if**
8: $\quad$ **end if**
9: **end for**
10: $newDbPos := \delta_D(dbPos, \nu)$ /* Try to find delimiter-transition */
11: **if** $(newDbPos \neq FAIL)$ AND $(levPos \in F_{LEV})$ **then**
12: $\quad$ **if** $(curToken = m - 1)$ AND $(newDbPos \in F_D)$ **then**
13: $\quad\quad$ report match
14: $\quad$ **end if**
15: $\quad noperm\_recursive(newDbPos, curToken + 1, i_{LEV})$
16: **end if**

Algorithm 1: Approximate search without permutations

in the dictionary automaton and LEV are final. Here, the end of a token in state $p \in Q_D$ is indicated by an outgoing transition (pos_D, ν, pos'_D). This is why at each step the existence of such a transition is checked. If $(pos_D, \nu, pos'_D) \in Q_D$ and $pos_{LEV} \in F_{LEV}$, we successfully matched p_0 and the first token of one or more expressions $d \in D$. In this case we begin to search for the next token p_2 (in the general case, after having matched p_i, we continue with p_{i+1}), so we switch to the appropriate set of characteristic vectors, $charvec_2$ ($charvec_{i+1}$, resp.). Starting points of the search are pos'_D in A_D and the initial state i_{LEV} of the levenshtein automaton.
The complete search procedure is displayed in algorithm 1.

3 A method for handling approximate search modulo permutation

We now turn to the problem of searching a multi-token expression $p = (p_1, .., p_m)$ in a dictionary D allowing the tokens to be misspelled and to appear in arbitrary order. More formally, an expression $d = (d_1, .., d_m) \in D$ is called a *match* for p iff there is a variant $p' = (p'_1, .., p'_m)$ of the pattern so that $d_L(p_i, p'_i) \leq k$ for $1 \leq k \leq m$ and the multisets $\{d_1, .., d_m\}_M$ and $\{p'_1, .., p'_m\}_M$ are equal.

3.1 A brute-force idea to handle permutations

A trivial first idea for performing a search modulo permutation would be to explicitly generate all permutations of the pattern and then to use the approximate search method described in section 2 to handle the spelling errors. But a search pattern p containing m tokens leads to $m!$ permutations to be processed at runtime, which makes this idea unpracticable.

However, a less intuitive variant of this approach turns out to be more promising: instead of searching for all permutations of the pattern, every possible variant can be stored in a dictionary $\breve{D}$ to come to the same result. We compile an automaton $A_{\breve{D}}$ that recognizes all permutations of all $d \in D$:
$\mathcal{L}(A_{\breve{D}}) = \{(s_1, .., s_m) \mid \exists d \in D : \{d_1, .., d_m\}_M = \{s_1, .., s_m\}_M\}$
The objection of exponential growth might seem even more appropriate here: For example, one single expression containing 6 tokens leads to 720 permutations, instead of a dictionary D containing 50000 such expressions we have to handle $\breve{D}$ with 36 million entries. Nevertheless, if $\breve{D}$ is transferred into a *minimal* deterministic automaton[2], extensive sharing of prefixes and suffixes among the permutations leads to reasonable sizes of the automata. To continue with the above example, a listing of the 36 million entries of a test database is 1.2 GB large if stored in a text file, while only 600MB are needed to store the resulting minimal automaton.
Whenever $\breve{D}$ is small enough to fit into the RAM of a machine, first tests promise very fast search times. Limitations and precise speed results remain to be examined.

In the following section we present a more elaborate solution to this same search problem, which is applicable also to very large dictionaries with several hundred-thousands of entries.

[2] see [DMWW00] for construction algorithms

3.2 An algorithm for huge dictionary-sizes

We now assume that we're dealing with data that is too large to build an automaton $A_{\breve{D}}$ containing all possible permutations. For these cases, new ways have to be found for efficiently matching multi-token expressions in very large collections of data.
A trick often used in computer sciences to efficiently perform operations on sets is to sort the sets in question first. In particular, equivalence of two sets A and B can be checked in linear time if both A and B are sorted according to some ordering function. As explained before, the matching problem described here can be seen as a check for equivalence of multisets, so we follow a similar strategy for our purposes: sorted variants p' of the pattern are compared to a sorted variant S of the dictionary that contains all $d \in D$ in sorted order.

This idea leads to the search method presented in [Ref05]. We now give only a short description of the used mechanisms and structures, as far as they are important for the remainder of the report. Please consult the cited paper for a more detailed description.

3.2.1 A sorted index of the dictionary

We introduce a simple index structure S where all entries of D appear sorted according to a certain ordering function:
$S = \{(s_1, .., s_m) \mid \exists (d_1, .., d_m) \in D : \{d_1, .., d_m\}_M = \{s_1, .., s_m\}_M \text{ AND } (s_i < s_{i+1}, (1 \leq i \leq m-1))\}$
Similarly to section 2, we build a finite-state automaton $A_S = (\Sigma \cup \{\nu\}, Q_S, i_S, F_S, \delta_S)$ that recognizes the following language:
$\mathcal{L}(A_S) = \{s_1 \circ \nu \circ ... \circ \nu \circ s_{|s|} \circ \nu \mid s \in S\}$

3.2.2 Search procedure

The search proceeds in two major steps: At first, correction candidates have to be found for every token of the pattern p_i. In a second step, sequences of correction candidates have to be searched in the index automaton S.

Computation of correction candidates for each token First we address the problem of spelling errors by computing a list of correction candidates for each single token, using the algorithm of Mihov and Schulz (see above). In the general case the effiency and success of lexical methods for word correction depend largely on the choice of a good dictionary, or a collection thereof (cf. [SRSM03]). In our case, the set of all correct word forms, that is, the set of words appearing as tokens in the dictionary D, is well-known. This *perfect dictionary* T of tokens containing all relevant word forms but no others, still increases the speed and accuracy of this correction method.

annete	meier	münchen	lotstraße	bayern
annett	beier	muenchen	brotstraße	baier
annette	geier	münchen	lothstrasse	bayer
jannet	maier		lothstraße	bayern
	mair		rothstrasse	
	meier		rothstraße	
	meir			
	meyer			
	meyr			
K_0	K_1	K_2	K_3	K_4

Figure 1: Hypothetical table of correction candidates for pattern "annete,meier,münchen,lotstraße,bayern" in a dictionary of German post-addresses

The result of this first step are lists of correction candidates $Cand_i$ for each token $p_i, (1 \leq i \leq m)$ of the pattern. $Cand_i = \{t \in T | d_L(p_i, t) \leq k\}$. With $Cand_i[x]$ we address the x-th candidate of $Cand_i$. Note that for p_i that have no spelling errors, $p_i \in T$ and $d_L(p_i, p_i) = 0$, so p_i is included in $Cand_i$.

Valid sequences of candidates For a pattern $p = (p_1, .., p_m)$ and the respective candidate lists $Cand_1, .., Cand_m$, we call a sequence $c = (c_1, .., c_m)$ of correction candidates *valid*, iff

- There is a bijective relationship between $c_1, .., c_m$ and $Cand_1, .., Cand_m$: c is composed of exactly one member of each candidate list.
- The tokens of c are in sorted order: $c_i < c_{i+1} (1 \leq i \leq m-1)$

The valid sequences are exactly those sequences of candidates that have to be searched for in the index automaton A_D: the first constraint avoids that several candidates of one token are used in one sequence. The second constraint addresses the properties of the index structure S: since all entries are in sorted order, search can be restricted to sorted queries. In the following we describe a method to compute the set of valid sequences of correction candidates in a way that allows a very efficient parallel search in S.

Joint list of all candidates To compute the valid sequences, all candidate lists $Cand_i$ are joined in a single list L and sorted according to the same ordering function that was used to sort S. For each candidate $L[i]$ at position i in L, a function *origin* maps *pos* to $origin(pos) = n$ to indicate that $L[pos]$ "originates" from candidate list $Cand_n$. Additionally, another function *candsAvailable* assigns a bit-vector of length m to each member of the list. The meaning of these vectors will become clear shortly.

	cand[n]	origin[n]	candsAvailable[n]
0	geier	2	11111
1	lothstrasse	4	11111
2	jannet	1	11111
3	rothstrasse	4	11111
4	beier	2	11111
5	annett	1	11111
6	bayer	5	11111
7	lothstraße	4	11111
8	rothstraße	4	11111
9	baier	5	11101
10	meier	2	11101
11	maier	2	11101
12	annette	1	11101
13	mair	2	01101
14	meir	2	01101
15	meyer	2	01101
16	meyr	2	01101
17	münchen	3	00101
18	muenchen	3	00101
19	bayern	5	00001

Figure 2: correction candidates in a joint list L

Computation and matching of the sequences Using this joint list L it turns out to be very easy to compose, word by word, all sequences c of candidates that both satisfy the constraints for valid sequences and are equivalent to a successful path in the index automaton A_S. The second constraint concerning the sorted order of the candidates is met simply by choosing c_i only from below the position of c_{i-1} in the sorted list L. To check for the first constraint, that is, to assure exactly one candidate is taken from each candidate list, we maintain a bitvector *blocked* with the initial value $blocked = 0^m$. The n-th bit is set iff the current sequence already includes a candidate from $Cand_n$.
At first, we search from the top of L downwards to find the first candidate $c_1 = L[x_1]$ so that $pos_1 = \delta^*(i_S, c_1 \circ \nu)$ is defined. In order to avoid choosing another candidate originating from the same token $p_n(n = origin(x_1))$ of p at a later stage, the n-th bit of *blocked* is set to 1.
In the following, the search for appropriate candidates is continued in a similar manner downwards in L - $c_i = L[x_i]$ must meet the following constraints:

- $x_i < x_{i-1}$
- $(blocked \ \& \ 0^{n-1}10^{m-n}) = 0^m \qquad (n = origin(x_i))$
- $pos_i = \delta_S^*(pos_{i-1}, c_i \circ \nu)$ defined

In case no candidate can be found to fill c_i, a backtracking procedure starts to find a new fitting candidate $c'_{i-1} = L[x'_{i-1}]$ with $x'_{i-1} > x_{i-1}$. That way all possible valid sequences

c of correction candidates are compared to the index automaton. Whenever a proper candidate c_m can be found, the current sequence $c_1, .., c_m$ is returned as match for pattern p.

Speedup using *candsAvailable* Let's recall the first constraint for a sequence of correction candidates to be valid: Each token p_n of the pattern has to contribute exactly one candidate to a valid sequence c. A simple additional piece of information can be used to recognize configurations at an early stage which can never lead to a valid sequence and thus to the finding of a match: with each position x of L we associate a bit-vector $candsAvailable(x)$ with the n-th bit set if $\exists L[x'] : (x' \leq x \text{ AND } origin(x') = n)$.

Imagine a situation where a candidate $L[x_{i+1}]$ for has to be found somewhere below position x_i. If there is some token p_n of the pattern that neither has contributed a candidate to $c_1, .., c_i$ nor are there any candidates originating from p_n left in the remaining part of L. In this case, $c_1, ..c_i$ can never be completed to be a valid sequence. Using the bit-vectors *blocked* and $candsAvailable(x_i)$ this situation can be recognized very easily: If such a p_n exists, the n-th bit is 0 in both vectors. So we can formulate a further restriction for the process while searching for a candidate c_i at a position $\geq x_i$:
$(blocked \ \& \ candsAvailbale(x_i)) = 1^m$. Otherwise backtracking is started.

The search algorithm is illustrated in pseudocode, see algorithm 2.

function search

1: $A_S := (\Sigma \cup \{\nu\}, Q_S, i_S, F_S, \delta_S)$
2: $p := (p_0, .., p_{m-1})$ /* the query */
3: $L :=$ list of correction candidates of all Q_i in Q
4: sort L according to chosen order
5: $recursive\_search\_v2(trie.root, 0, 0^m, 0)$

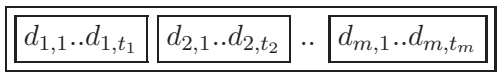
function recursive_search(triePos, listIndex, blocked, depth)

1: **if** $trie.isFinal(triePos)$ AND $(depth = m)$ **then** /* check if match was found */
2: $matches := trie.getAnnotations(triepos)$
3: report matches /* Care for duplicates here */
4: **end if**
5: **while** $(listIndex < sizeOfList)$ **do**
6: **if** $((candsAvailable[listIndex] \mid blocked) \neq 1^m)$ **then**
7: return /* Some bits are set neither in *candsAvailable* nor in *blocked* */
8: **end if**
9: $i := origin[listIndex]$
10: **if** $((blocked \mid 0^i 10^{m-i-1}) \neq blocked)$ AND $(trie.walk(triePos, cand[listIndex]) \neq FAIL)$ **then**
11: $blocked := (blocked \mid 0^i 10^{m-i-1})$ /* block appropriate bit */
12: $newTriePos := trie.walk(triePos, cand[listIndex])$
13: $recursive\_search\_v2(newTriePos, listIndex + 1, depth + 1)$
14: $blocked := (blocked \ \& \ 1^i 01^{m-i-1})$ /* release blocked bit */
15: **end if**
16: $listIndex++$
17: **end while**

Algorithm 2: Search algorithm

4 Approximate Record Matching

We now turn to a collection of useful variants of the matching algorithm described in the previous chapter. In contrast to the basic algorithm, the dictionary D is now assumed to be not only to be a list of tokens. Instead, an additional structural layer is introduced: each entry consists of m *parts*, which in turn each are composed of t (atomic) tokens:



A structure like this appears natural in various applications like matching of mailing-addresses or bibliographic references. In these cases, it rarely makes sense to allow free permutation of all tokens: For example, in a dictionary D of bibliographic references tokens can be grouped in different fields for authors, title, journal etc. We can safely assume that no tokens from the *author*-part and the *title*-part interchange their positions. Furthermore, while the ordering of the different names in the *author*-part might accidently be wrong, the tokens of the *title*-part are very unlikely ever to be transposed.

In the following we provide search methods for the following cases:

- Permutation is allowed only inside parts
- Only complete parts may be permuted - word order inside the parts is fixed.
- Permutation may occur both inside and among parts

On the side of the dictionary D, it is of course a pre-condition that the structure of all entries is known. For each of the cases below, an index structure in the form of a finite state automaton A_D is computed similarly to the basic method. However the re-sorting process is different for the single cases. To separate the parts in A_D a second delimiter symbol τ is used ($\tau \neq \nu, \tau \notin \Sigma$).

The structure of the search pattern p can not be considered to be known beforehands. We assume that the input is a string - in practice the markers separating the parts are usually not standardised and/or ambiguous. A good example are bibliographic references, where the parts are separated seemingly by random with different delimiters: mostly commas or full stops are used, both symbols that also appear in other contexts, e.g. in abbreviations or titles. So, for the search pattern p, we do not assume to know any structure but the plain token boundaries: $p = p_1, .., p_{plen}$

4.1 Permutation inside parts

The first variant we describe allows permutation inside each part. More precisely, a pattern p matches an entry $d = ((d_{1,1}..d_{1,t_1}), .., (d_{m,1}..d_{m,t_m})) \in D$ iff

- there is a variant $p' = (p'_1, .., p'_{plen})$ of p where each p'_i is a correction candidate for p_i: $d_L(p_i, p'_i) \leq k$ for $1 \leq i \leq plen$
- p' can be divided into m parts so that the i-th part is equivalent to $d_{i,1}, .., d_{i,t_i}$ modulo permutation:
  $p' = (p'_{1,1}, .., p'_{1,t_1}, p'_{2,1}, .., p'_{m-1,t_{m-1}}, p_{m,1}, .., p_{m,t_m})$,
  $\forall i \in [1; m] : \{p'_{i,1}, .., p'_{i,t_i}\}_M = \{d'_{i,1}, .., d'_{i,t_i}\}_M$

Intuitively, this definition of a match leads to something similar to a sequential application of the basic matching algorithm: At first, a prefix of p has to be matched with the initial part of some $d \in D$, then a consecutive sub-sequence of p has to matched with the second part of d and so on.

Index structure: Following this idea of sequential runs, the re-sorting of all entries $d \in D$ for the index automaton A_S (see section 3.2.1) is carried out for each part of d individually: The tokens of each part are re-arranged following a certain ordering function, but the order of the different parts remains the same. As mentioned before, a new delimiter symbol τ is used to separate the parts.

Joint list of candidates: The list L of candidates is constructed and sorted exactly as for the basic algorithm.

Search procedure: The search starts in a very similar manner as for the basic algorithm: Sequences of correction candidates $c_1, c_2, ..$ from L ($c_i = L[x_i]$) are searched according to these criteria (similarly to 3.2.2):

- $x_i < x_{i-1}$
- $(blocked \ \& \ 0^{n-1}10^{m-n}) = 0^m \qquad (n = origin(x_i))$
- $pos_i = \delta^*_S(pos_{i-1}, c_i \circ \nu) \quad$ or $\quad pos_i = \delta^*(pos_{i-1}, c_i \circ \tau)$ defined

Whenever c_i could be read with a following token-delimiter ν, the search is continued recursively in a normal fashion. But if c_i can be followed by a part-delimiter τ, this indicates that $c_1, .., c_i$ matches a complete part of some entry (or several entries) in A_S. But to confirm this sequence as a valid first part of a possible match, more things have to be checked: Imagine the situation where a first part is searched beginning from the root i_S with an initial empty vector $blocked = 0^m$. As stated before, a first part of some $s \in S$ has to match some prefix of p. We can easily check in *blocked*, if the chosen candidates c_i for the sequence correspond to a prefix of p: This is the case iff *blocked* has the form 1^i0^{m-i}. This constraint holds also for the matching of successive parts: if a token c_i can be

read in A_S with a following part-delimiter and $blocked = 1^i0^{m-i}$, a part was successfully matched. In this situation, pos_i is a state of A_S where at least one new part (with tokens in sorted order) begins. So we start a new sub-search from position pos_i and, again, at the top of the list L.

Speedup using candsAvailable The backtracking procedure is handled exactly as for the basic algorithm. However, the usage of *candsAvailable* has to be adjusted to the idea of sequential runs. As stated above, the completion of a part can only succeed if $blocked = 1^i0^{m-i}$. Intuitively, *blocked* contains only one sequence of set bits, beginning at the first bit, without any gaps. However, during the matching process of a part, gaps can occur. Using *candsAvailable*, we can check at any time of the process, if all those gaps of the current *blocked*-vector can possibly be filled with the remainder of the list L:
Be the r-th bit of *blocked* the rightmost-set bit: $blocked = ?^{r-1}10^{m-r}$. Then the following equation serves as a check if all gaps can be filled:
$((blocked \mid stillPossible) \ \& \ 1^r0^{m-r}) = 1^r0^{m-r}$

5 Closing remarks

Our search method for approximate search modulo permutation has turned out to be very flexible in various respects. As shown here for one case, it can easily be modified to carry out more sophisticated search tasks. In a future version of this report, also the other mentioned cases are to be described - part of the algorithms and implementations could already be developed. We are convinced that we will be able to apply the new search algorithms in practice to match bibliographic references. Experiments to use the methods in a framework described in [And05] are about to be started.

References

[And05] E. Anderl. Computerlinguistische Analyse bibliographischer Referenzen. Master's thesis, CIS, Ludwig-Maximilian-Universität München, 2005.

[DMWW00] Jan Daciuk, Stoyan Mihov, Bruce Watson, and Richard Watson. Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1):3–16, April 2000.

[MS04] S. Mihov and K. Schulz. Fast approximate search in large dictionaries. *Computational Linguistics*, 30, 2004.

[Ref05] U. Reffle. Varianten approximativen Matchings. Master's thesis, CIS, Ludwig-Maximilian-Universität München, 2005.

[SRSM03] C. Strohmaier, C. Ringlstetter, K. Schulz, and S. Mihov. Lexical postcorrection of ocr-results: The web as a dynamic secondary dictionary. In *Proceedings of ICDAR-03*, pages 1133–1137, Edinburgh, 2003.