Error-tolerant Finite State Recognition

Kemal Oflazer
Department of Computer Engineering and Information Science,
Bilkent University, Ankara, TR-06533, Turkey
ko@cs.bilkent.edu.tr

Abstract

Error-tolerant recognition enables the recognition of strings that deviate slightly from any string in the regular set recognized by the underlying finite state recognizer. In the context of natural language processing, it has applications in error-tolerant morphological analysis, and spelling correction. After a description of the concepts and algorithms involved, we give examples from these two applications: In morphological analysis, error-tolerant recognition allows misspelled input word forms to be corrected, and morphologically analyzed concurrently. The algorithm can be applied to the morphological analysis of any language whose morphology is fully captured by a single (and possibly very large) finite state transducer, regardless of the word formation processes (such as agglutination or productive compounding) and morphographemic phenomena involved. We present an application to error-tolerant analysis of agglutinative morphology of Turkish words. In spelling correction, error-tolerant recognition can be used to enumerate correct candidate forms from a given misspelled string within a certain edit distance. It can be applied to any language whose morphology is fully described by a finite state transducer, or with a word list comprising all inflected forms with very large word lists of root and inflected forms (some containing well over 200,000 forms), generating all candidate solutions within 10 to 45 milliseconds (with edit distance 1) on a SparcStation 10/41. For spelling correction in Turkish, error-tolerant recognition operating with a (circular) recognizer of Turkish words (with about 29,000 states and 119,000 transitions) can generate all candidate words in less than 20 milliseconds (with edit distance 1). Spelling correction using a recognizer constructed from a large word German list that simulates compounding, also indicates that the approach is applicable in such cases.

1 Introduction

Error-tolerant finite state recognition enables the recognition of strings that deviate *slightly* from any string in the regular set recognized by the underlying finite state recognizer, by perturbations such as deletion, insertion or replacement of symbols. For example, suppose we have a recognizer for the regular set over $\{a, b\}$ described by the regular expression $(aba + bab)^*$, and we would like to recognize inputs which may be corrupted (but not too much) due to a number of reasons: e.g., *abaaaba* may be matched to *abaaba* correcting for a spurious *a*, or *ababba* may be matched to either *abaaba* correcting a *b* to an *a*, or to *ababab* correcting the reversal of the last two symbols. There have been a number of approaches to error-tolerant recognition, e.g., [19, 12]. These are however suited for applications where the pattern or the regular expression is small, and the sequence to be matched is large which is typical in information retrieval applications. For example, Myers and Miller [12], present an $O(MN)$ algorithm with M being the length of the string and N, being the length of the regular expression. In the applications we are concerned with, the recognizers (and the corresponding regular expressions) are very large. The approach presented in this paper uses the finite state recognizer that recognizes the regular set, but relies on a very efficiently controlled recognition algorithm based on depth-first search of the state graph of the recognizer.

2 Error-tolerant Finite State Recognition

We can informally define error-tolerant recognition with a finite state recognizer as the recognition of all strings in the regular set (accepted by the recognizer), and additional strings which can be obtained from any string in the set *by a small number of unit editing operations.* Error-tolerant recognition requires an error metric for measuring how much two strings deviate from each other. The *edit distance metric* between two strings measures the minimum number of unit editing operations of *insertion, deletion, replacement of a symbol, and transposition of adjacent symbols* [3], that are necessary to convert one string into another. Let $X = x_1, x_2, \ldots, x_m$, and $Y = y_1, y_2, \ldots, y_n$ denote strings of length $mandnrespectivelyofm$ symbols from an alphabet A. $X[j]$ ($Y[j]$) denotes the initial substring of X (Y) up to and including the jth symbol. Given X and Y, the edit distance $ed(X[m], Y[n])$ computed according to the recurrence below, gives the minimum number of unit editing operations to convert one string to the other [4].

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

For example $ed(recoginze, recognize) = 1$, since transposing *i* and *n* in the former string would give the latter. Similarly $ed(sailn, failing) = 3$ as in the former string, one could change the initial *f* to *s*, insert an *i* before the *n*, and insert a *g* at the end to obtain the latter.

Given a (deterministic) finite state recognizer, *R*,[1] let $L \subseteq A^*$ be the regular language accepted by *R*, and let $t > 0$, be the edit distance error threshold. We define a string $X[m] \notin L$ to be recognized by *R* with an error at most *t*, if the set $C = \{Y[n] \mid Y[n] \in L \text{ and } ed(X[m], Y[n]) \leq t\}$ is not empty.

Any finite state recognizer can also be viewed as a directed graph with arcs are labeled with symbols in A.[2] Standard finite state recognition corresponds to traversing a path (possibly involving cycles) in the graph of the recognizer, starting from the start node, to one of the final nodes, so that the concatenation of the labels on the arcs along this path matches the input string. For error-tolerant recognition one needs to find *all paths from the start node to one of the final nodes, so that when the labels on the arcs along a path are concatenated, the resulting string is within a given edit distance threshold t, of the (erroneous) input string.* With $t > 0$, the recognition procedure becomes a search on this graph.

Searching the graph of the recognizer has to be very fast if error-tolerant recognition is to be of any practical use. This means that paths that can lead to no solutions have to be pruned so that the search can be limited to a very small percentage of the search space. Thus we need to make sure that any candidate string that is generated as the search is being performed, does not deviate from certain initial

[1] R is described by a 5-tuple $R = (Q, A, \delta, q_0, F)$ with Q denoting the set of states, A denoting the input alphabet, $\delta : Q \times A \rightarrow Q$, denoting the state transition function, $q_0 \in Q$ denoting the initial state, and $F \subseteq Q$ denoting the final states.The approach presented here not limited to deterministic machines, though.

[2] We may use state and node, and transition and arc, interchangeably.

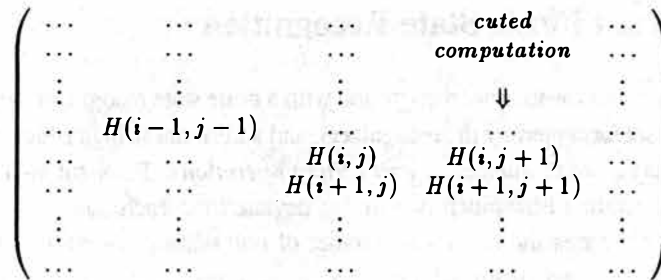

Figure 2: Computation of the elements of the H matrix.

matrix H with element $H(i,j) = ed(X[i], Y[j])$ [4]. We can note that the computation of the element $H(i+1,j+1)$ recursively depends on only $H(i,j), H(i,j+1), H(i+1,j)$ and $H(i-1,j-1)$, from the earlier definition of the edit distance. During the depth first search of the state graph of the recognizer, entries in column n of the matrix H have to be (re)computed, only when the candidate string is of length n. During backtracking, the entries for the last column are discarded, but the entries in prior columns are still valid. Thus all entries required by $H(i+1,j+1)$, except $H(i,j+1)$, are already available in the matrix in columns $i-1$ and i. The computation of $cuted(X[m], Y[n])$ involves a loop in which the minimum is computed. This loop (indexing along column $j+1$), computes $H(i,j+1)$ before it is needed for the computation of $H(i+1,j+1)$. (See Figure 2.)

We now present in Figure 3 a simple example for this search algorithm for a simple finite state recognizer for the regular expression $(aba + bab)^*$, and the search graph for the input string $ababa$. The

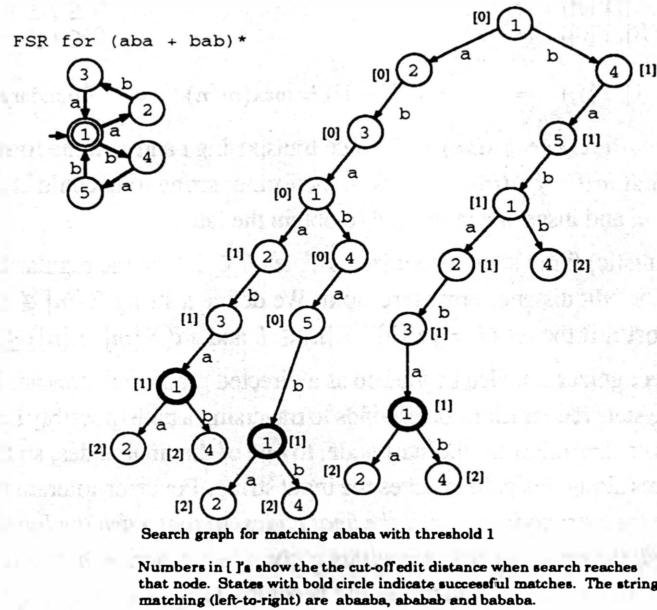

Figure 3: Recognizer for $(aba + bab)^*$ and search graph for $ababa$.

thick circles from left to right indicate the nodes at which we have the matching strings $abaaba$, $ababab$ and $bababa$, respectively. Prior visits to the final state 1, violate the final edit distance constraint. (Note that the visit order of siblings depend on how one orders the outgoing arcs from a state.)

```
/*push empty candidate, and start node to start search */
push((ε, q0))
while stack not empty
    begin
       pop((Y', qi)) /* pop partial surface string Y'
                        and the node */
       for all qj and a such that δ(qi, a) = qj
          begin /* extend the candidate string */
             Y = concat(Y', a) /* n is the current length of Y */
          /* check if Y has deviated too much, if not push */
             if cuted(X[m], Y[n]) ≤ t then push((Y, qj))
          /* also see if we are at a final state */
             if ed(X[m], Y[n]) ≤ t and qj ∈ F then output Y
          end
    end
```

Figure 1: Algorithm for error-tolerant recognition

substrings of the erroneous string by more than the allowed threshold. To detect such cases, we use the notion of a *cut-off edit distance.* The cut-off edit distance measures the minimum edit distance between an initial substring of the incorrect input string, and the (possibly partial) candidate correct string. Let Y be a partial candidate string whose length is n, and let X be the incorrect string of length m. Let $l = \min(1, n - t)$ and $u = \max(m, n + t)$. The cut-off edit distance $cuted(X[m], Y[n])$ is defined as

$$cuted(X[m], Y[n]) = \min_{l \leq i \leq u} ed(X[i], Y[n]).$$

For example, with $t = 2$:

$$cuted(\text{reprter}, \text{repo}) = \{\min\{ed(\text{re}, \text{repo}) = 2, ed(\text{rep}, \text{repo}) = 1, ed(\text{repr}, \text{repo}) = 1, ed(\text{reprt}, \text{repo}) = 2, ed(\text{reprte}, \text{repo}) = 3\} = 1.$$

Note that except at the boundaries, the initial substrings of the incorrect string X considered are of length $n - t$ to length $n + t$. Any initial substring of X shorter than $n - t$ needs more than t insertions, and any initial substring of X longer than $n + t$ requires more than t deletions, to at least equal Y in length, violating the edit distance constraint.

Given an incorrect string X, a partial candidate string Y is generated by successively concatenating relevant labels along the arcs as transitions are made, starting with the start state. Whenever we extend Y, we check if the cut-off edit distance of X and the partial Y, is within the bound specified by the threshold t. If the cut-off edit distance goes beyond the threshold, the last transition is backed off to the source node (in parallel with the shortening of Y) and some other transition is tried. Backtracking is recursively applied when the search can not be continued from that state. If, during the construction of Y, a final state is reached without violating the cutoff edit distance constraint, and $ed(X[m], Y[n]) \leq t$ at that point, then Y is a valid correct form of the incorrect input string.[3]

Denoting the states by subscripted q's (q_0 being the initial state) and the symbols in the alphabet (and labels on the directed edges) by a, we present the algorithm for generating all Y's by a (slightly modified) depth-first probing of the graph in Figure 1. The crucial point in this algorithm is that the cut-off edit distance computation can be performed very efficiently by maintaining a *global* m by n

[3]Note that we have to do this check since we may come to other irrelevant final states during the search.

3 Application to Error-tolerant Morphological Analysis

Error-tolerant finite state recognition can be applied to morphological analysis, in which, instead of rejecting a given misspelled form, the analyzer attempts to apply the morphological analysis to forms that are within a certain (configurable) edit distance of the incorrect form. Two-level transducers [10] provide a suitable model for the application of error-tolerant recognition. Such transducers capture all morphotactic and morphographemic phenomena, and alternations in the language in a uniform manner. They can be abstracted as finite state transducers over an alphabet of lexical and surface symbol pairs `l:s`, where either `l` or `s` (but not both) may be the null symbol `0`. It is possible to apply error-tolerant recognition to languages whose word formations employ productive compounding and/or agglutination, and in fact to any language whose morphology is described completely as one (very large) finite state transducer. Full scale descriptions using this approach already exist for a number of languages like English, French, German, Turkish, Korean [8].

Application of error-tolerant recognition to morphological analysis proceeds as described earlier. After a successful match with a surface symbol, the corresponding lexical symbol is appended to the output gloss string. During backtracking the candidate surface string and the gloss string are again shortened in tandem. The basic algorithm for this case is given in Figure 4.[4] The actual algorithm is a slightly optimized version of this where transitions with null surface symbols are treated as special during forward and backtracking traversals to avoid unnecessary computations of the cut-off edit distance.

```
/*push empty candidate string, and start node
to start search on to the stack */
push((ε, ε, q0))
while stack not empty
    begin
       pop((surface', lexical', qi)) /* pop partial strings
             and the node from the stack */
       for all qj and l: s such that δ(qi, l: s) = qj
          begin /* extend the candidate string */
             surface = concat(surface', s)
             if cuted(X[m], surface[n]) ≤ t then
                begin
                     lexical = concat(lexical', l)
                     push((surface, lexical, qj))
                     if ed(X[m], surface[n]) ≤ t and qj ∈ F then
                       output lexical
                end
          end
    end
```

Figure 4: Algorithm for error-tolerant morphological analysis.

We can demonstrate error-tolerant morphological analysis with a two-level transducer for the analysis of Turkish morphology. Agglutinative languages such as Turkish, Hungarian or Finnish, differ from languages like English in the way lexical forms are generated. Words are formed by productive affixations of derivational and inflectional affixes to roots or stems. Furthermore, roots and affixes may undergo changes due to various phonetic interactions. A typical nominal or a verbal root gives rise to thousands of valid forms which never appear in the dictionary. For instance, we can give the following (rather exaggerated) adverb example from Turkish:

uygarlaştıramayabileceklerimizdenmişsinizcesine

[4] Note that transitions are now labeled with $l : s$ pairs.

whose root is the adjective *uygar* (civilized).[5] The morpheme breakdown (with morphological glosses underneath) is:[6]

uygar	*+laş*	*+tır*	*+ama*	*+yabil*	*+ecek*
civilized	+AtoV	+CAUS	+NEG	+POT	+VtoA(AtoN)
+ler	*+imiz*	*+den*	*+miş*	*+siniz*	*+cesine*
+3PL	+POSS-1PL	+ABL(+NtoV)	+PAST	+2PL	+VtoAdv

The portion of the word following the root consists of 11 morphemes each of which either adds further syntactic or semantic information to, or changes the part-of-speech, of the part preceding it. Though most words one uses in Turkish are considerably shorter than this, the example serves to point out some of the fundamental difference of the nature of the word structures in Turkish and other agglutinative languages, from those of languages like English.

Our morphological analyzer for Turkish is based on a lexicon of about 28,000 root words and is a re-implementation of PC-KIMMO version of the same description [1, 13], using Xerox two-level transducer technology [9]. This description of Turkish morphology has 31 two-level rules that implement the morphographemic phenomena such as vowel harmony and consonant changes across morpheme boundaries etc., and about 150 additional rules, again based on the two-level formalism, that fine tune the morphotactics by enforcing sequential and long-distance feature sequencing and co-occurrence constraints, in addition to constraints imposed by standard alternation linkage among various lexicons to implement the paradigms. Turkish morphotactics is circular due to the relativization suffix in the nominal paradigm, and multiple causative suffixes in the verb paradigm. There is also considerable linkage between nominal and verbal morphotactics due to productive derivational suffixes. The minimized finite state transducer constructed by composing the transducers for root lexicons, morphographemic rules and morphotactic constraints, has 32,897 states and 106,047 transitions, with an average fan out of about 3.22 transitions per state (including transitions with null surface symbols). It analyzes a given Turkish lexical form into a sequence of *feature-value tuples* (instead of the more conventional sequence of morpheme glosses) that are used in a number of natural language applications.[7]

This transducer has been used as input to an analyzer implementing the error-tolerant recognition algorithm in Figure 4. The analyzer first attempts to parse the input with $t = 0$, and if it fails, relaxes t up to 2, if it can not find any parse with a smaller t, and can process about 150 (correct) forms a second on a Sparcstation 10/41.[8,9] Below, we provide a transcript of a run:[10]

```
ENTER WORD > getirn
Threshold 0 ...  1 ...


getiri   =>   ((CAT NOUN)(ROOT getiri)(AGR 3SG)(POSS NONE)(CASE NOM))
              (return (on investment))
getirin  =>   ((CAT NOUN)(ROOT getiri)(AGR 3SG)(POSS 2SG)(CASE NOM))
              (your return)
getiren  =>   ((CAT VERB)(ROOT getir)(SENSE POS)(CONV ADJ YAN))
              (s/he who brings)
getirin  =>   ((CAT VERB)(ROOT getir)(SENSE POS)(MOOD IMP)(AGR 2PL))
              (bring!  (imp 2pl))
getir    =>   ((CAT VERB)(ROOT getir)(SENSE POS)(MOOD IMP)(AGR 2SG))
              (bring!  (imp 2sg))
getire   =>   ((CAT VERB)(ROOT getir)(SENSE POS)(MOOD OPT)(AGR 3SG))
              (let him bring)
```

[5] This is a manner adverb meaning roughly "(behaving) as if you were one of those whom we might not be able to civilize."

[6] Glosses in parentheses indicate derivations not explicitly indicated by a morpheme.

[7] The Xerox software allows the resulting finite state transducer to be exported in a tabular form which can be imported to other applications.

[8] No attempt was made to compress the finite state recognizer.

[9] The Xerox *infl* program working on the proprietary compressed representation of the same transducer can process about 1000 forms/sec on the same platform.

[10] The outputs have been slightly edited for formatting. The feature names denote the usual morphosyntactic features. CONV denotes derivations to the category indicated by the second token with a suffix or derivation type denoted by the third token, if any. We have also added English glosses that most closely approximate the semantics of the word.

```
getirt  =>  ((CAT VERB)(ROOT getir)(VOICE CAUS)(SENSE POS)(MOOD IMP)(AGR 2SG))
            (have him/her bring!  (imp 3sg))


ENTER WORD > akıllınnikiler
Threshold 0 ...  1 ...  2 ...


akıllınınkiler => ((CAT NOUN)(ROOT akıl)(CONV ADJ LI)
                     (CONV NOUN)(AGR 3SG) (POSS NONE)(CASE GEN)
                     (CONV PRONOUN REL)(AGR 3PL)(POSS NONE)(CASE NOM))
                  ((things) of the smart (one))
akıllınınkiler => ((CAT NOUN)(ROOT akıl)(CONV ADJ LI)
                     (CONV NOUN)(AGR 3SG)(POSS 2SG)(CASE GEN)
                     (CONV PRONOUN REL)(AGR 3PL)(POSS NONE)(CASE NOM))
                  ((things) of your smart (one))
akıllındakiler => ((CAT NOUN)(ROOT akıl)(CONV ADJ LI)
                     (CONV NOUN)(AGR 3SG)(POSS 2SG)(CASE LOC)
                     (CONV ADJ REL)(CONV NOUN)(AGR 3PL)(POSS NONE)(CASE NOM))
                  ((things) at your smart (one))


ENTER WORD > eviminkinn
Threshold 0 ...  1 ...


eviminkini =>     ((CAT NOUN)(ROOT ev)(AGR 3SG)(POSS 1SG)(CASE GEN)
                     (CONV PRONOUN REL)(AGR 3SG)(POSS NONE)(CASE ACC))
                  ((thing) of my house (accusative))
eviminkine =>     ((CAT NOUN)(ROOT ev)(AGR 3SG)(POSS 1SG)(CASE GEN)
                     (CONV PRONOUN REL)(AGR 3SG)(POSS NONE)(CASE DAT))
                  (to the (thing) of my house)
eviminkinin =>    ((CAT NOUN)(ROOT ev)(AGR 3SG)(POSS 1SG)(CASE GEN)
                     (CONV PRONOUN REL)(AGR 3SG)(POSS NONE)(CASE GEN))
                  (of the (thing) of my house)


ENTER WORD > uygarlaştıramadıklarmıızdanmışsınızcasına
Threshold 0 ...  1 ...


uygarlaştıramadıklarımızdanmışsınızcasına =>
            ((CAT ADJ)(ROOT uygar)(CONV VERB LAS)(VOICE CAUS)(SENSE NEG-C)
                  (CONV ADJ DIK)(AGR 3PL)(POSS 1PL)(CASE ABL)
                  (CONV VERB)(TENSE NARR-PAST)(AGR 2PL)
                  (CONV ADVERB CASINA)(TYPE MANNER))
            (behaving) as if you were one of those whom we could not civilize


ENTER WORD > okatulna
Threshold 0 ...  1 ...  2 ...


okutulma =>       ((CAT VERB)(ROOT oku)(VOICE CAUS)(VOICE PASS)(SENSE NEG)
                     (MOOD IMP)(AGR 2SG))
                  (don't be forced to read!  (imp 2sg))
okutulma =>       ((CAT VERB)(ROOT oku)(VOICE CAUS)(VOICE PASS)(SENSE POS)
                     (CONV NOUN MA)(TYPE INFINITIVE)
                     (AGR 3SG)(POSS NONE)(CASE NOM))
                  ((the act of) being forced to read))
okutulan =>       ((CAT VERB)(ROOT oku)(VOICE CAUS)(VOICE PASS)(SENSE POS)
                     (CONV ADJ YAN))
                  ((thing) that is caused to be read)
okutulana =>      ((CAT VERB)(ROOT oku)(VOICE CAUS)(VOICE PASS)(SENSE POS)
                     (CONV ADJ YAN)(CONV NOUN)(AGR 3SG)(POSS NONE)(CASE DAT))
                  (to the (thing) that is caused to be read)
okutulsa =>       ((CAT VERB)(ROOT oku)(VOICE CAUS)(VOICE PASS)(SENSE POS)
                     (MOOD COND)(AGR 3SG))
                  (if (it) were caused to be read)
okutula =>        (CAT VERB)(ROOT oku)(VOICE CAUS)(VOICE PASS)(SENSE POS)
                     (MOOD OPT)(AGR 3SG))
                  (let it be caused to be read)
```

In an application context, the candidates that are generated by such a morphological analyzer can be disambiguated or filtered to a certain extent by constraint-based tagging techniques, e.g., [16, 18] that take into account syntactic context for morphological disambiguation.

4 Applications to Spelling Correction

Spelling correction is an important application for error-tolerant recognition. There has been substantial amount of work on spelling correction (see the excellent review by Kukich [11]). Most methods essentially enumerate plausible candidates which resemble the incorrect word, and use additional heuristics to rank the results.[11] However, most techniques assume a word list of all words in the language. These approaches are suitable for languages like English for which it is possible to enumerate such a list. They are not directly suitable or applicable to languages like German, that have very productive compounding, or to agglutinative languages like Finnish, Hungarian or Turkish, in which the concept of a word is much larger than what is normally found in a word list. For example, Finnish nouns have about 2000 distinct forms while Finnish verbs have about 12,000 forms [6, pages 59–60]. The case in Turkish is also similar where, for instance nouns may have about 170 different forms, not counting the forms for adverbs, verbs, adjectives, or other nominal forms, generated (sometimes circularly) by derivational suffixes. Hankamer [7] gives much higher figures (in the millions) for Turkish, presumably by taking into account derivations.

There have been some recent approaches to spelling correction using morphological analysis techniques. Veronis [17] presents a method for handling quite complex combinations of typographical and phonographic errors, the latter being the kind of errors usually made by language learners using computer-aided instruction. This method takes into account phonetic similarity in addition to standard types of errors. Aduriz *et al.* [5] have used a two-level morphology approach to spelling correction in Basque which uses two-level rules to describe common insertion and deletion errors, in addition to the two-level rules for the morphographemic component. Oflazer and Güzey [15] have used a two-level morphology approach to spelling correction in agglutinative languages, which has used a coarser morpheme-based morphotactic description instead of the finer lexical/surface symbol approach presented here. More recently, Bowden and Kiraz [2] have used a multi-tape morphological analysis technique for spelling correction in Semitic languages which, in addition to the insertion, deletion, substitution and transposition errors, allows for various language specific errors.

For languages like English where all inflected forms can be included in a word list, the word list can be used to construct a finite state recognizer structured as a standard letter tree recognizer (which has an acyclic graph) to which error-tolerant recognition can be applied. Furthermore, just as in morphological analysis, transducers for morphological analysis can obviously be used for spelling correction, so one algorithm can be applied to any language whose morphology is described using such transducers. We demonstrate the application of error-tolerant recognition to candidate generation in spelling correction by constructing finite state recognizers in the form of letter tree from large word lists that contain root and inflected forms of words for 10 languages, obtained from a number of resources on the Internet. Table 1 gives statistics about the word lists used. The Dutch, French, German, English (two different lists), and Italian, Norwegian, Swedish, Danish and Spanish word lists contained some or all inflected forms in addition to the basic root forms. The Finnish word list contained unique word forms compiled from a corpus, although the language is agglutinative.

For edit distance thresholds, 1, 2, and 3, we selected randomly, 1000 words from each word list and perturbed them by random insertions, deletions, replacements and transpositions so that each misspelled word had the respective edit distance from the correct form. Kukich [11], citing a number of studies, reports that typically 80% of the misspelled words contain a single error of one of the unit operations, though there are specific applications where the percentage of such errors are lower. Our earlier study of an error model developed for spelling correction in Turkish also indicated similar results [15].

[11] Ranking is dependent on the language, the application, and the error model. It is an important component of the spelling correction problem, but is not addressed in this paper.

Table 1: Statistics about the language word lists used

Language	Words	Arcs	Average Word Length	Maximum Word Length	Average Fan-out
Finnish	276,448	968,171	12.01	49	1.31
English-1	213,557	741,835	10.93	25	1.33
Dutch	189,249	501,822	11.29	33	1.27
German	174,573	561,533	12.95	36	1.27
French	138,257	286,583	9.52	26	1.50
English-2	104,216	265,194	10.13	29	1.40
Spanish	86,061	257,704	9.88	23	1.40
Norwegian	61,843	156,548	9.52	28	1.32
Italian	61,183	115,282	9.36	19	1.84
Danish	25,485	81,766	10.18	29	1.27
Swedish	23,688	67,619	8.48	29	1.36

Table 2: Correction Statistics for Threshold 1

Language	Average Misspelled Word Length	Average Correction Time (msec)	Avg. Time to First Solution (msec)	Average Number of Solutions Found	Average % of Space Searched
Finnish	11.08	45.45	25.02	1.72	0.21
English-1	9.98	26.59	12.49	1.48	0.19
Dutch	10.23	20.65	9.54	1.65	0.20
German	11.95	27.09	14.71	1.48	0.20
French	10.04	15.16	6.09	1.70	0.28
English-2	9.26	17.13	7.51	1.77	0.35
Spanish	8.98	18.26	7.91	1.63	0.37
Norwegian	8.44	16.44	6.86	2.52	0.62
Italian	8.43	9.74	4.30	1.78	0.46
Danish	8.78	14.21	1.98	2.25	1.00
Swedish	7.57	16.78	8.87	2.83	1.57

Table 2 present the results from correcting these misspelled word lists for edit distance threshold 1. The runs were performed on a Sparcstation 10/41. The second column in these tables gives the average length of the misspelled string in the input list. The third column gives the time in milliseconds to generate *all* solutions, while the fourth column gives the time to find the first solution. The fifth column gives the average number of solutions generated from the given misspelled strings with the given edit distance. Finally, the last column gives the percentage of the search space (that is, the ratio of forward traversed arcs to the total number of arcs) that is searched when generating all the solutions. Due to space limitations, we are not able to give full results for thresholds 2, and 3, but for $t = 2$, the average correction time ranged from 80 msecs to 320 msecs, with about 1.3% to 7.5% of the search space being searched. For $t = 3$, which is very unlikely in real applications, the average correction time ranged from 200 msecs to 1200 msecs, with about 3% to 17% of the search space being searched. (See Oflazer [14] for details on these results.)

Table 3: Spelling correction results for the Turkish finite state recognizer.

Threshold t	Average Misspelled Word Length	Average Correction Time (msec)	Avg. Time to First Solution (msec)	Average Number of Solutions Found	Average % of Space Searched
1	8.63	17.90	7.41	4.92	1.23
2	8.59	164.81	57.87	55.12	11.12
3	8.57	907.02	63.59	442.17	60.00

4.1 Spelling Correction for Agglutinative Word Forms

The transducer for Turkish developed for morphological analysis, using the Xerox software, was also used for spelling correction. However, the original transducer had to be simplified into a recognizer for two reasons. For morphological analysis, the concurrent generation of the lexical gloss string requires that occasional transitions with an empty surface symbol be taken, to generate the gloss properly. Secondly, a given surface form can morphologically be interpreted in many ways which is important in morphological processing. In spelling correction, the presentation of only one of such surface forms is sufficient. To remove all empty transitions and analyses with same surface forms from the Turkish transducer, a recognizer recognizing only the surface forms was extracted by using the Xerox tool *ifsm*. The resulting recognizer had 28,825 states and 118,352 transitions labeled with just surface symbols. The average fan-out of the states in this recognizer was about 4. This transducer was then used to perform spelling correction experiments in Turkish.

In the first set of experiments three word lists of 1000 words each were generated from a Turkish corpus, and words were perturbed as described before, for error thresholds of 1, 2, and 3 respectively. The results for correcting these words are presented in Table 3. It should be noted that the percentage of search space searched may not be very meaningful in this case since the same transitions may be taken in the forward direction, more than once.

On a separate experiment which would simulate a real correction application, about 3000 misspelled Turkish words again compiled from a corpus, were processed by successively relaxing the error threshold starting with $t = 1$. Of these set of words, 79.6% had an edit distance of 1 from the intended correct form, while 15.0% had edit distance 2, and 5.4% had edit distance 3 or more.[12] The average length of the incorrect strings was 9.63 characters. The average correction time was 77.43 milliseconds (with 24.75 milliseconds for the first solution). The average number of candidates offered per correction was 4.29, with an average of 3.62% of the search space being traversed, indicating that this is a very viable approach for real applications. For comparison, the same recognizer running as a spell checker ($t = 0$) can process correct forms at a rate of about 500 words/sec.

4.2 Spelling correction with compounding

Using the 174,249 word German word list used in the experiments above, a finite state recognizer recognizing G^+, where G is a German word in the list, was constructed. The intention was to simulate the compounding process though clearly the finite state recognizer constructed is not a correct compounding recognizer for German. One hundred (pseudo-) German compounds were formed by concatenating words form the original word list, and then introducing errors randomly. The results with thresholds

[12] In almost all of these, the misspelling were due to rendering of non-ASCII Turkish characters with the nearest ASCII neighbors, e.g., *ü* typed as *u*.

Table 4: Spelling correction results for (simulated) German compounding.

Threshold t	Average Misspelled Word Length	Average Correction Time (msec)	Avg. Time to First Solution (msec)	Average Number of Solutions Found	Average % of Space Searched
1	32.96	332.58	84.50	1.77	0.30
2	32.03	1138.63	308.52	5.52	2.45
3	31.53	4757.70	966.27	9.71	10.17

1,2, and 3 are presented in Table 4. It should however be stressed again that the numbers for search space percentages reflect the percentage with respect to the number of static recognizer links. One can, however, get a reasonable approximation to the actual percentage of the search space by observing that by the average length of the words in the test file is about three times the average word length in the original German word list. Thus on the average about three traversals of the whole finite state recognizer are made and thus the search percentages above should be divided by three.

5 Conclusions

This paper has presented an algorithm for error-tolerant finite state recognition which enables a finite state recognizer to recognize strings that deviate mildly from some string in the underlying regular set, along with results of its application to error-tolerant morphological analysis, and candidate generation in spelling correction. The approach is very fast and applicable to any language with a given a list of root and inflected forms, or with a finite state transducer recognizing or analyzing its word forms. It differs from previous error-tolerant finite state recognition algorithms in that it uses a given finite state machine, and is more suitable for applications where the number of patterns (or the finite state machine) is very large and the string to be matched is small.

6 Acknowledgments

This research was supported in part by a NATO Science for Stability Project Grant TU-LANGUAGE. I would like to thank Xerox Advanced Document Systems, and Lauri Karttunen of Xerox Parc and of Rank Xerox Research Centre (Grenoble) for providing us with the two-level transducer development software. Kemal Ülkü and Kurtuluş Yorulmaz of Bilkent University implemented some of the algorithms.

References

[1] Evan L. Antworth. *PC-KIMMO: A two-level processor for Morphological Analysis.* Summer Institute of Linguistics, Dallas, Texas, 1990.

[2] Tanya Bowden and George A. Kiraz. A morphographemic model for error correction in nonconcatenative strings. In *Proceedings of the* 33rd *Annual Meeting of the Association for Computational Linguistics*, pages 24–30, Boston, MA, 1995.

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

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

[5] I. Aduriz *et al*. A morphological analysis based method for spelling correction. In *Proceedings of the Sixth Conference of the European Chapter of the Association for Computational Linguistics*, Utrecht, The Netherlands, April 1993.

[6] Gerald Gazdar and Chris Mellish. *Natural Language Processing in PROLOG, An Introduction to Computational Linguistics*. Addison-Wesley Publishing Company, 1989.

[7] Jorge Hankamer. Morphological parsing and the lexicon. In W. Marslen-Wilson, editor, *Lexical Representation and Process*. MIT Press, 1989.

[8] Lauri Karttunen. Constructing lexical transducers. In *Proceedings of the* 16th *International Conference on Computational Linguistics*, volume 1, pages 406–411, Kyoto, Japan, 1994. International Committee on Computational Linguistics.

[9] Lauri Karttunen and Kenneth. R. Beesley. Two-level rule compiler. Technical Report, XEROX Palo Alto Research Center, 1992.

[10] Lauri Karttunen, Ronald M. Kaplan, and Annie Zaenen. Two-level morphology with composition. In *Proceedings of the* 15th *International Conference on Computational Linguistics*, volume 1, pages 141–148, Nantes, France, 1992. International Committee on Computational Linguistics.

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

[12] Eugene W. Myers and Webb Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51(1):5–37, 1989.

[13] Kemal Oflazer. Two-level description of Turkish morphology. In *Proceedings of the Sixth Conference of the European Chapter of the Association for Computational Linguistics (A full version appears in* Literary and Linguistic Computing, *Vol.9 No.2, 1994.)*, Utrecht, The Netherlands, April 1993.

[14] Kemal Oflazer. Error-tolerant finite state recognition with applications to morphological analysis and spelling correction. *Computational Linguistics*, 22(1), 1996. To appear.

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

[16] Kemal Oflazer and İlker Kuruöz. Tagging and morphological disambiguation of Turkish text. In *Proceedings of the* 4th *Conference on Applied Natural Language Processing*, pages 144–149, Stuttgart, Germany, October 1994.

[17] Jean Veronis. Morphosyntactic correction in natural language interfaces. In *Proceedings of* 13th *International Conference on Computational Linguistics*, pages 708–713. International Committee on Computational Linguistics, 1988.

[18] Atro Voutilainen and Pasi Tapanainen. Ambiguity resolution in a reductionistic parser. In *Proceedings of the Sixth Conference of the European Chapter of the Association for Computational Linguistics*, pages 394–403, Utrecht, The Netherlands, April 1993.

[19] Sun Wu and Udi Manber. Fast text searching with errors. Technical Report TR91–11, Department of Computer Science, University of Arizona, 1991.