Spelling Correction in Agglutinative Languages

Kemal Oflazer and Cemaleddin Güzey

BILKENT UNIVERSITY

Department of Computer Engineering
and
Information Science

Technical Report BU-CEIS-94-01

Spelling Correction in Agglutinative Languages

Kemal Oflazer and Cemaleddin Güzey
Department of Computer Engineering and Information Science
Bilkent University
Bilkent, Ankara, 06533, Turkey
e-mail: {ko,guzey}@cs.bilkent.edu.tr, Fax: (90-312) 266-4126

Abstract: This paper presents an approach to spelling correction in agglutinative languages that is based on two-level morphology and a dynamic programming based search algorithm. Spelling correction in agglutinative languages is significantly different than in languages like English, because the concept of a word in such languages is much wider that the entries found in a dictionary, owing to productive word formation by derivational and inflectional affixations. After an overview of certain issues and relevant mathematical preliminaries, we formally present the problem and the our proposed solution. We then present results from our experiments with spelling correction in Turkish, a Ural–Altaic agglutinative language

1 Introduction

Spelling correction is an important component of any system for processing text. Creation of textual information is prone to many errors: people make errors when they type, or sometimes are ignorant of the correct spelling of a word. Optical character recognition systems often make errors during recognition due to bad quality of the input, change in fonts or typefaces, or insufficient training. Nowadays, many word processors employ a spelling checking and correction functionality. However this functionality is mainly available for languages like English, French, etc. Recently, Kukich [10] has presented a very comprehensive overview of the spelling correction problem (mainly for English) and approaches many researchers have employed over years.

Agglutinative languages such as Turkish or Finnish, differ from languages like English in the way lexical forms are generated. The concept of word in such languages is much wider than the roots or lemmata found in a dictionary or lexicon. Words are formed by productive affixations of derivational and inflectional suffixes to roots or stems like, "beads-on-a-string" [14]. Furthermore, roots and suffixes (morphemes) may undergo changes at the boundaries due to various phonetic interactions, and may have to obey constraints like vowel harmony. A typical nominal or a verbal root may have thousands of valid forms which never appear in the dictionary. For instance, we can give the following (rather exaggerated) example from Turkish:

uygarlaştıramayabileceklerimizdenmişsinizcesine

whose root is the adjective *uygar* (civilized).[1] Its morpheme breakdown is:

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

uygar+laş+tır+ama+yabil+ecek+ler+imiz+den+miş+siniz+cesine

The portion of the word following the root, consists of 11 morphemes each of which which either adds further 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, this example serves to point out the fundamental difference of the spelling checking and correction problem in such languages. Any errors made in the transcription, such as insertion, deletion, replacement, or transposition of characters may necessitate a whole set of techniques not typically used in systems for languages like English.

Our prior work has mainly been on spelling checking in Turkish [12, 13], and two-level morphological analysis of Turkish [11]. In this work, we develop an algorithm for spelling correction for agglutinative languages that we have applied to Turkish. Our approach uses a two-level morphological analyzer and generator, coupled with a dynamic-programming like search procedure for intelligently enumerating candidate lexical forms from a given misspelled form. However, our use of two-level morphology for spelling correction, is fundamentally different from that of Aduriz et.al., [1]. In the following sections, we overview the spelling correction problem in general and in agglutinative languages. We then present some preliminary definitions and mathematical background and introduce an algorithm for spelling correction for agglutinative languages. We finally present results from our implementation for Turkish.

2 The spelling correction problem

Du and Chang [3] define the spelling correction problem as follows:

> From a set of known words (dictionary), find those words that most resemble a given (misspelled) character string.

The keyword in this definition is "resemble." It is difficult to express rigorously how two strings resemble. Generally, a distance metric is used to compare two strings. Then the problem becomes that of finding those words that are neighbors of a given character string with respect to a given distance metric.

There have been a number of proposals to be used as the distance metric in comparing two strings [8, 10, 15]. The most popular and widely used metrics are *q-gram* and *linear trace* based metrics. In the q-gram metric, two strings are compared according to the number of different substrings of length q they share. In the linear trace method, two strings are compared according to an *edit distance* metric which measures the extent of changes one needs to apply to one of the strings to get the other string.

3 Spelling correction in agglutinative languages

As briefly discussed earlier, agglutinative languages have certain aspects that make the spelling correction problem substantially harder and different than that for languages like English. The expression "from a set of known words" no longer implies what is usually found in typical word list, and now means "all possible words that can be generated from a given root word by derivational and inflectional suffixes." For example, Finnish nouns have about 2000 distinct forms while Finnish

verbs have about 12,000 forms ([4], pp. 59–60). The case in Turkish is also similar where nouns may have about 170 basic different forms, not counting the forms for adverbs, verbs, adjectives, or other nominal forms, generated (sometimes recursively) by derivational suffixes. Hankamer [5], however, reports a much higher number (in the millions) of forms for Turkish nouns and verbs.

If we look closely into the problem, it will not be difficult to observe that it consists of two subproblems.[2] Given a misspelled word

1. determine all the roots from the dictionary that can be the root of the misspelled word, and
2. generate (systematically) all the possible words that "resemble" the given character string, from roots identified in subproblem 1.

The first step of the problem is relatively easy because of the static structure of the root dictionary. Various techniques developed for spelling correction, say, in English can usually be applied here. However, sometimes the roots are partially or totally deformed either by morphophonemic variations, or by misspellings which makes this step a bit more complicated, or in the latter case rather intractable. We will opt not to deal with cases where a root can not be determined, especially due to total or near-total deformation.

The second step is the heart of the problem. Producing all the possible words from all the known roots requires an exhaustive generate and test search procedure. This search must be done in an intelligent way such so that all the words that resemble the misspelled string are produced and words that do not resemble it are not produced. Obviously, once all such forms are generated, there is still the problem of ranking them so that the most "plausible" suggestions are ranked higher according to editing distance, word usage statistics, etc. We do not consider this aspect in this paper.

In the following sections we will formally introduce the problem after a brief overview of the various notational tools we employ.

3.1 Notation

We denote the set of the surface forms of the roots in the language[3] by R, and the set of lexical forms of the roots by R_{lex}.[4] We use $X = x_1, x_2, ..., x_m, Y = y_1, y_2, ..., y_n$ to denote strings from the alphabet Σ of the language. X will denote the surface form of the incorrect or misspelled string, and Y will typically denote the surface string that is a (possibly partial) candidate word. Y_{lex} will denote the lexical form of this candidate string.[5] The notation $X[i : j] = x_i, x_{i+1}, ..., x_j$ refers to the

[2] In this paper, we do not deal with languages that have productive prefixes.

[3] From now on, *language* will refer to an agglutinative language.

[4] Here, we are referring to the two levels of forms in the two-level morphology terminology: the *lexical form* which essentially corresponds to the structure of a word in terms of morphemes etc., and the *surface form* which is the surface realization of the lexical form as allowed by the automata implementing the two-level correspondence rules [14, 2, 9, 6].

[5] Just to make this clear we can give an example from Turkish. For instance

`ev+lAr+nHn` (house+PLU+GEN)

represents such a lexical form where `A` represents a low unrounded vowel (a and e in Turkish) which is unresolved for frontness, and `H` represents a high vowel (ı, i, u, and ü) which is unresolved for other features. The `+`'s indicate the morpheme boundaries. When this lexical form is processed by the generation component of a two-level morphological analyzer, the surface form obtained is:

`evlerin`

substring (from characters i to j inclusive) of any string X If i is missing, then the substring refers to the prefix of the string up to and including the jth character. $X[0]$ denotes empty substring and $|X|$ denotes the length of string X. We assume the existence of a function, $surface()$ to generate surface strings from lexical strings, i.e., $surface(Y_{lex}) = Y$. The function $surface()$ applies the constraints imposed by the automata implementing the two-level rules for the language.

3.2 Edit distance metrics

In both parts of the problem, we need some criteria to measure how much two strings resemble each other. The most widely accepted and readily applicable metrics are based on *linear traces* and *q-grams.*

3.2.1 Edit distance with linear traces

The edit distance measures how many unit operations are necessary to convert one string into another. The unit operations are *insertion, deletion, replacement* of single character and *transposition* of two adjacent characters.

Definition 1 (Edit Distance) *Given two strings X and Y of length m and n respectively, then $ed(X[m], Y[n])$*[6] *computed according to the recurrence*

$$
\begin{aligned}
ed(X[i+1], Y[j+1]) &= ed(X[i], Y[j]) && \text{if } x_{i+1} = y_{j+1} \\
&= 1 + min\{ed(X[i-1], Y[j-1]), && \text{if both } x_i = y_{j+1} \\
&\qquad ed(X[i+1], Y[j]), && \text{and } x_{i+1} = y_j \\
&\qquad ed(X[i], Y[j+1])\} && \\
&= 1 + min\{ed(X[i], Y[j]), && \text{otherwise} \\
&\qquad ed(X[i+1], Y[j]), && \\
&\qquad ed(X[i], Y[j+1])\} && \\
\\
ed(X[0], Y[j]) &= j && 1 \le j \le n \\
ed(X[i], Y[0]) &= i && 1 \le i \le m
\end{aligned}
$$

gives the minimum number insertions, deletions, replaces and transpositions one needs to perform to convert one string to the other.

This is a slight modification of edit distance formulas given by Du and Chang [3] and by Wagner et.al., [15]. For example, the edit distance between *iflobra* and *foobar* is 3: an i is to be deleted, l has to be replaced with an o, and ra has to be transposed.

3.2.2 q-grams

A q-gram is a simply substring of length q. The q-gram distance is based on counting the number of occurrences of different q-grams in two strings. The most popular q-grams are *one-gram* and

where vowel harmony rules have resolved the `A` and the `H`, and the first `n` in the last morpheme has disappeared since the previous morpheme ends with a consonant. (See Oflazer [11].)

[6] We may occasionally drop the index of one or both arguments to indicate that we are referring to the whole string.

bi-gram. Increasing q, decreases the number of q-grams in a given string and increases position dependency between characters. We can define the q-gram distance between two strings.

Definition 2 (q-gram distance) *Let $g \in \Sigma^q$ be a q-gram. Let $G(X[m])[g]$ denote the total number of the occurrences of g in the string X of length m. The q-gram distance between $X[m]$ and $Y[n]$ is*

$$D_q(X[m], Y[n]) = \sum_g |G(X[m])[g] - G(Y[n])[g]|$$

3.3 Recognizing and generating strings in the language

We would like to capture and abstract the behavior of a morphological generator and analyzer for the given language by two finite state automata.

Definition 3 *A finite state generator $M_g = (P, \delta, V, S, F)$ where P is a set of states, δ is the state transition function, V is the output alphabet, S is the starting state, and F is a set of final states, generates, all correctly formed words of the language. The transitions of M_g are of the form $\delta(p_i) = p_j$ (p_i and $p_j \in P$), with an output $v_k \in V$ which denotes the lexical form of a morpheme in the language and also labels the transition. It should be noted that it is possible to go from one state p_i to another p_j by more than one transition, outputting a different morpheme. We say a string Y_{lex} is generated by M_g, if Y_{lex} is formed by concatenating, in order, the outputs of the machine as we traverse starting from S to one of the states in F. We denote by $L(M_g)$ as the set of all lexical strings generated by M_g.*

M_g essentially captures the morphotactics of the language, and in general may contain circular transition sequences (as is the case in Turkish). Applying the function $surface()$ to a string generated by M_g will give us a valid surface string in the language.

We also have a finite state recognizer M_r which recognizes whether given surface strings are in the language or not. When a word in language is input to M_r, if M_r reaches one of its final states, the input surface word is a legal word in the language; hence M_r implements a spelling checking functionality for the language. It should be noted that the actual finite state machine may actually be very complicated dealing with morphophonemic changes as required by parallel automata implementing the two-level rules. The details of M_r are not really necessary for our exposition.

We can view M_g as a *directed graph* with states as the vertices and the state transitions as the edges. For misspelled words that are rejected by M_r, the spelling correction problem is then to find all directed paths from the start state S of M_g, to one of the states in F so that the distance of the surface form of the lexical word formed by concatenating the morphemes (that is, v's) along this path, is within a certain edit distance threshold of the misspelled word.

3.4 Formal description of the spelling correction problem

We can now define the spelling correction problem as:

Definition 4 *For given an incorrect word X (rejected by M_r), and a distance threshold t, find the solution set of possible correct words*

$$S(X, t) = \{Y | ed(X, Y) \leq t \text{ and } Y = surface(Y_{lex}) \text{ and } Y_{lex} \in L(M_g)\}.$$

in viable time and space.

We will now consider two subproblems of the problem.

3.5 Determining the root

In an agglutinative language, presenting alternatives for a given incorrect string requires determination of all possible roots.[7] The criteria used to select roots is based on the edit distance between the (surface form) of a root and the prefixes of X. If any root word has a edit distance from all prefixes of the misspelled word less than the threshold t, then it is a candidate root. An example from Turkish makes this clear. For the misspelled Turkish word $X = kalayhlamak$, *kalayla* and *kalas* (among others) are possible roots when $t = 1$ because $ed(kalayhla, kalayla) = ed(kalay, kalas) \leq 1$. However, *yatay* is not a possible root since $ed(kala, yatay) = 3 > 1$, $ed(kalay, yatay) = 2 > 1$, and $ed(kalayh, yatay) = 3 > 1$. Other prefixes of X are either too long or too short to consider, since the edit distance in such cases exceeds the threshold. This observation leads to the following definition:

Definition 5 *The set of all the possible roots for the incorrect word X is,*

$$PR(X,t) = \{r \mid ed(X[i], r) \leq t \text{ and } 1 \leq i \leq m \text{ and } r \in R\}$$

3.5.1 Root determination with q-grams

In general, the cardinality of R – the set of all roots– is usually in the tens of thousands, thus one needs a fast search algorithm that works on a pre-constructed data structure for efficient determination of $PR(X,t)$. We have chosen to represent the q-gram information associated with root words with an inverted bit vector structure so that the bit-vector corresponding to a q-gram has 1's at positions corresponding to the root words containing that q-gram. Since the root list is static, such a structure can be constructed off-line, and can be accessed randomly by using the q-gram as a key. Let us denote by k, how many q-grams in a root that we would like to consult, and by t_q, how many q-grams we are willing to leave out and yet call the root a possible candidate root. To generate the set of such roots, we take the first k $q-grams$ of the incorrect word and then consider all $\binom{k}{k-t_q}$ subsets of the $(k - t_q)$ q-grams. For each such subset, we intersect the bit vectors corresponding to the q-grams in that subset. We then union the bit vectors resulting from each subset. The resulting bit vector then has 1's corresponding to root words which are "close" to a prefix of the misspelled word X. These roots are then filtered by the edit distance constraint in Definition 5 to compute $PR(X,t)$. The parameters k and t_q are in general fixed once according to the average length of the root words in a language.

3.6 Generating candidate words from a given root

Assuming that we have a set of root words found as described above, we now have to generate words in the language having this root, that do not deviate from the given misspelled string by more than the threshold. We can state this problem as

[7] Recall that we do not deal with the case where the language also has productive prefixes.

Subproblem: Given the misspelled word X, the threshold value t, and the generator automaton M_g, and the set of roots $PR(X,t)$, generate all elements of the solution set $C(X,t)$.

We will first consider solutions where the root portion of the word may be misspelled and the rest may be okay. We call such solutions as being *on the left edge of the word.*

3.6.1 Getting solutions on the left edge

The edit distances between an element r of $PR(X,t)$ and certain prefixes of X[8] are between 0 and t. Sometimes, these distances are equal to t, which means no further mismatches between X and Y – the candidate string– are to be tolerated. In such cases, there is no need for further checking by generating a morpheme sequence. Just concatenating the portion of X that remains after aligning r with a prefix of X, to r_{lex}, and then the generating the surface string will give us candidate Y strings. However, determination of the alignment of the root word r with X is somewhat tricky because the root in X may be deformed.

Let us now define a new edit distance measure between r, an element of $PR(X,t)$, and X. This is the minimum of the edit distances between r and any prefix of X.

Definition 6 (Prefix edit distance) *The prefix edit distance between r and X is*

$$pred(X,r) = min\{ed(X[i],r) \mid 1 \leq i \leq m\}.$$

Definition 7 (Alignment index) *The set of alignment indexes of r in X is*

$$index(X,r) = \{i \mid ed(X[i],r) = pred(X,r)\}.$$

For the example given before

$pred(kalayhlamak, kalayla) = 1$ and $index(kalayhlamak, kalayla) = \{8\}$ and $pred(kalayhlamak, kalas) = 1$, and $index(kalayhlamak, kalas) = \{4,5\}$.

When $pred(X,r) = t$, the remaining part of X after alignment with the root r must completely occur in Y after r to satisfy $ed(X,Y) \leq t$. That is Y must be in the form $Y = surface(concatenate(r_{lex}, X[i+1:m]))$, $i \in index(X,r)$. For the example above, the candidate from root *kalayla*, is *kalaylamak*, which happens to be the correct solution and hence is accepted by M_r. The candidates due to *kalas* are *kalashlamak* and *kalashylamak*, both of which are rejected by M_r. Constructing Y's for all the elements of the index set and all elements of the candidate root set, gives all possible solutions on the edge. If the intended word is among these (as can be possibly be indicated by a user) then we can eliminate the remaining work.

3.6.2 Generating candidate words

Getting solutions on edge will ease the computation of the correct word if the erroneous part happens to be in the root, but it does not solve the problem completely. The solution requires

[8]These prefixes are $X[|r|-t]$ through $X[|r|+t]$.

a depth-first generate and test probing of the finite-state automaton M_g, starting with the start state S. We now have to find all the paths from this state to one of the final state using the roots in $PR(X,t)$, so that when the morphemes along this path are concatenated and surface string is generated, it is within an edit distance t of X.

When the search starts morphemes are concatenated and the length of the candidate lexical string Y_{lex} increases. After one step of the search, the partial surface string Y is compared with a suitable prefix of X. In most of the cases the candidate Y will deviate from these prefixes of X by more than the threshold without reaching a final state, so that it can no longer lead to a viable solution. In such cases we do not consider any further transitions from that state.

The following theorems from Du and Chang [3] helps us to determine when a partial candidate Y will not yield any result.

Theorem 1 *The error distance matrix for all prefixes of X and Y, $H_{m \times n}$ where $H(i,j) = ed(X[i],Y[j])$ $1 \leq i \leq m$ and $1 \leq j \leq n$, has the following properties:*

a) $H(i,j) - 1 \leq H(i+1,j) \leq H(i,j) + 1$ *for* $0 \leq i < m, 0 \leq j \leq n$*,*

b) $H(i,j) - 1 \leq H(i,j+1) \leq H(i,j) + 1$ *for* $0 \leq i \leq m, 0 \leq j < n$*,*

c) $H(i,j) \leq H(i+1,j+1) \leq H(i,j) + 1$ *for* $0 \leq i < m, 0 \leq j < n$*.*

Proof: See Du and Chang [3].

Theorem 2 *Let $H_{m \times n}$ be the error distance matrix in Theorem 1. Assume that $m \geq n$ and let $d = m - n$. Then, the sequence of elements of H, along the path $H(1,1) - H(2,1) - H(3,1) - \ldots - H(d+1,1) - H(d+2,2) - \ldots - H(m,n)$, are non-decreasing.*

Proof: (Du and Chang [3]) From Theorem 2-b, $H(d+1,1) \geq H(d+1,0) - 1 = d$. Also, from Theorem 1-c, $H(1,1) \leq H(0,0) + 1 = 1$. Hence, $H(d+1,1) - H(1,1) \geq d - 1$. Since we know from Theorem 1-a, $H(i+1,1) \leq H(i,1) + 1$, $H(1,1), H(2,1), \ldots, H(d+1,1)$ can only be non-decreasing. Remaining part of the path $H(d+1,1), H(d+2,2), \ldots, H(m,n)$ (by Theorem 2-c) is non-decreasing.

Theorem 2 determines a non-decreasing path in error distance matrix $H_{m \times n}$. This is not exactly what we need since the theorem requires that the length of the candidate string Y be known. In our case, we know that this length has to be in the range $m - t$ to $m + t$ for Y to be a candidate.

3.6.3 Limiting search during word generation

Due to the limitation above, we can not cut a branch of the search by looking at only a single path in $H_{m \times n}$ as defined in Theorem 2. First we construct H for the current (possibly partial) Y, then consider column n (n being the current length of Y), and then find the minimum of the edit distance values along this column between rows $n - t$ and $n + t$ inclusive. If this value exceeds the threshold t, then there is no point in further pursuing this path. Formally, we define a cut-off distance metric:

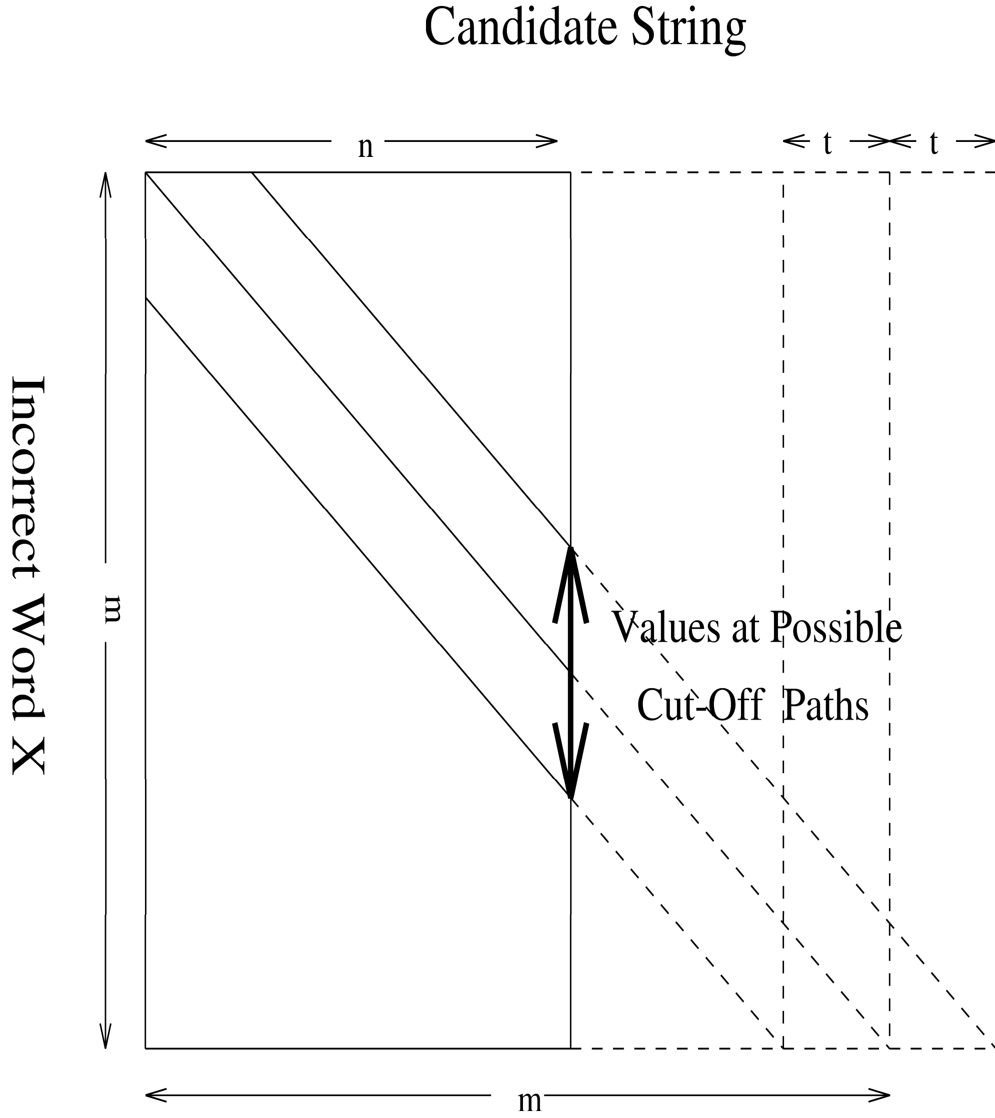

Figure 1: Determination of Cut-Off Paths in $H_{m \times n}$

Definition 8 (Cut-off distance)

$$cuted(X[m], Y[n]) = \begin{cases} min\{H[i,n] \mid 1 \leq i \leq n+t\} & \text{if } n < t \\ min\{H[i,n] \mid n-t \leq i \leq n+t\} & \text{if } t < n \leq m \\ min\{H[i,n] \mid n-t \leq i \leq m\} & \text{if } m < n \leq m+t \\ n-m & \text{if } m+t \leq n \end{cases}$$

The idea is similar to $pred(X, r)$ defined earlier, in that prefixes of X are again considered. If the cut-off edit distance between X and the current Y does not exceed the threshold, further transitions along from the state in M_g currently reached by Y_{lex}, have to be pursued.

After these observations we can state our algorithm for word generation as follows:

`Initialize` $C(X,t)$ `to the empty set`
`for all` $r \in PR(X,t)$

```
Push((r_lex, S)) /* on to search stack*/
while stack not empty
    Pop((Y_lex, p_i)) /* pop the next state to check */
    For all p_j such that δ(p_i) = p_j with morpheme v ∈ V as the output
        Y ← surface(concat(Y_lex, v)) /* generate surface form Y */
        If cuted(X[m], Y[n]) ≤ t
            Push((concat(Y_lex, v), p_j))
            if ed(X[m], Y[n]) ≤ t and p_j ∈ F, then insert Y into C(X, t)
```

Theorem 3 *The algorithm above produces exactly the solution set $C(X,t)$ when $PR(X,t)$ is given.*

Proof: Every element of $PR(X,t)$ is pushed into the stack. If $Y \in C(X,t)$ then $Y_{lex} \in L(M_g)$, that is, there is sequence of states in M_g

$$S \xrightarrow{r_{lex}} p_{i_1} \xrightarrow{v_{i_1}} p_{i_2} \xrightarrow{v_{i_2}} p_{i_3} \cdots p_{i_{k-1}} \xrightarrow{v_{i_{k-1}}} p_{i_k} \; (p_{i_k} \in F)$$

so that

$$Y = surface(concat(r_{lex}, v_{i_1}, v_{i_2}, \ldots, v_{i_k}))$$

and for all $Y_j = surface(concat(r_{lex}, v_{i_1}, v_{i_2}, \ldots, v_{i_j}))$ $1 \leq j \leq k$ we have $cuted(X, Y_j) \leq t$.

4 Results from experiments with spelling correction in Turkish

We first present two spelling correction examples from our implementation. For our implementation we used bi-grams (hence $q = 2$), and we chose k as 3 and t_q as 2.[9]

EXAMPLE 1

```
   Misspelled word:          verememetkeyim
   Threshold t:              1
   Solutions on left edge:   None
   Candidate Roots:          vere verem ver
   Solutions:                Lexical                  Surface
------------------------------------------------------------------------
   Edit distance 1           ver+yAmA+mAktA+yHm       verememekteyim
```

EXAMPLE 2

```
   Misspelled word:          çaışmalarıyla
   Threshold t:              2
   Solutions on left edge:   yazışmalarıyla           yatışmalarıyla
                             yapışmalarıyla           yakışmalarıyla
                             takışmalarıyla           sayışmalarıyla
                             mayışmalarıyla           katışmalarıyla
```

[9]The duplicate entries in the list of candidate roots for the second example, are in fact not duplicate; they have different part-of-speech categories and hence different morphotactics.

```
                    kapışmalarıyla          kakışmalarıyla
                    kaşışmalarıyla          çıkışmalarıyla
                    çağrılmalarıyla         çağırmalarıyla
                    atışmalarıyla           alışmalarıyla
                    yarışmalarıyla          tanışmalarıyla
                    karışmalarıyla          danışmalarıyla
                    çarpışmalarıyla         çakılmalarıyla
                    çağrışmalarıyla         barışmalarıyla
                    apışmalarıyla
Candidate Roots:    çağ çakı çal çalı çam çan çap çar çat çatı çav çay
                    çağ çak çakış çal çalış çap çat çatış çav

Solutions:          Lexical                 Surface
---------------------------------------------------------------------------
Edit distance 1     çat+Hş+mA+lArH+ylA      çatışmalarıyla
                    çap+Hş+mA+lArH+ylA      çapışmalarıyla
                    çalış+mA+lArH+ylA       çalışmalarıyla (correct form)
                    çakış+mA+lArH+ylA       çakışmalarıyla
                    çağ+Hş+mA+lArH+ylA      çağışmalarıyla
Edit Distance 2     çav+mA+lArH+ylA         çavmalarıyla
                    çav+Hş+mA+lAr+Hm+ylA    çavışmalarımla
                    çav+Hş+mA+lAr+Hn+ylA    çavışmalarınla
                    çav+Hş+mA+lArH+ysA      çavışmalarıysa
                    çat+Hl+mA+lArH+ylA      çatılmalarıyla
                    çat+mA+lArH+ylA         çatmalarıyla
                    çat+Hş+mA+lAr+Hm+ylA    çatışmalarımla
                    çat+Hş+mA+lAr+Hn+ylA    çatışmalarınla
                    çat+Hş+mA+lArH+ysA      çatışmalarıysa
                    çap+Hl+mA+lArH+ylA      çapılmalarıyla
                    çap+mA+lArH+ylA         çapmalarıyla
                    çap+Hş+mA+lAr+Hm+ylA    çapışmalarımla
                    çap+Hş+mA+lAr+Hn+ylA    çapışmalarınla
                    çap+Hş+mA+lArH+ysA      çapışmalarıysa
                    çalış+mA+lAr+Hm+ylA     çalışmalarımla
                    çalış+mA+lAr+Hn+ylA     çalışmalarınla
                    çalış+mA+lArH+ysA       çalışmalarıysa
                    çal+Hn+mA+lArH+ylA      çalınmalarıyla
                    çal+mA+lArH+ylA         çalmalarıyla
                    çakış+mA+lAr+Hm+ylA     çakışmalarımla
                    çakış+mA+lAr+Hn+ylA     çakışmalarınla
                    çakış+mA+lArH+ysA       çakışmalarıysa
                    çak+mA+lArH+ylA         çakmalarıyla
                    çağ+Hl+mA+lArH+ylA      çağılmalarıyla
                    çağ+mA+lArH+ylA         çağmalarıyla
                    çağ+Hş+mA+lAr+Hm+ylA    çağışmalarımla
                    çağ+Hş+mA+lAr+Hn+ylA    çağışmalarınla
                    çağ+Hş+mA+lArH+ysA      çağışmalarıysa
```

Table 1: Average number of operations per misspelled word, for $t = 1$

k	Recognitions	Generations	Edit Distance Operations	Solutions Offered	% Accuracy
3	30.9	311.2	2498.4	3.6	95.1
4	10.4	194.7	1068.8	2.4	78.2
5	3.9	88.5	471.7	1.5	54.0

Table 2: Average number of operations per misspelled word, for $t = 2$

k	Recognitions	Generations	Edit Distance Operations	Solutions Offered	% Accuracy
3	108.4	4462.0	20680.4	52.0	95.1
4	46.5	2247.8	10386.6	35.5	78.2
5	13.6	817.1	3799.9	20.3	54.0

The algorithm described above was tested on a set of 141 randomly selected incorrect words from Turkish text. Among these misspelled words, 14% had edit distance of 2, and the remaining 86% had edit distance 1, to their intended correct form. The morphological analyzer and generator that we used was our two-level specification for Turkish [11], developed using the PC-KIMMO system. This system has a rather comprehensive coverage of Turkish morphology and uses a root lexicon of about 24,000 words. It is, however, rather slow and can analyze only about 2 forms per second and can generate about 50 forms a second on Sun Sparcstations. So, instead of using timings, we counted the number of times the morphological analyzer and generator, and the edit distance computations, were called as these were the most expensive operations our algorithm.

These statistics show the average number of morphological recognitions and generations, and the edit distance operations required, and the number of correct solutions offered *per misspelled input word*. The last column indicates the percentage of cases the intended correct form was found. The results in Table 1 are for threshold $t = 1$ and the results in Table 2 are for for threshold $t = 2$. In both cases, bi-grams were used with $t_q = 2$. We varied k which determines how many bi-grams from the begining of the incorrect word are to be considered, between 3 and 5. This range was considered because according to some limited statistics we have on Turkish text, the average root length is about 4.5 characters. Choosing $k = 3$ allows more deformed roots to be handled at the expense of more computation, while choosing $k = 5$ sometime will not find roots with minor deformations but it runs faster.

Although, our PC-KIMMO based morphological analyzer and generator that we have used for this study, is rather slow, we have now ported our morphological analyzer system to the XEROX-PARC system by Karttunen [7], and intend to integrate it to our system. This system can recognize and generate Turkish forms in few milliseconds on Sun Sparcstations. With this system it will be possible to generate all solutions in about 1 to 2 seconds for $t = 1$ and in a few seconds for $t = 2$, on Sun Sparcstations.

5 Conclusions

This paper has presented a spelling correction algorithm for agglutinative languages that is based on a two-level morphological generator and analyzer, and a intelligent generate and test search procedure. The algorithm uses a q-gram based approach to determine the candidate roots words, and then from each root word, generates valid forms in the language, that are guaranteed not to deviate from the given misspelled string by more than a threshold, using morphological generator. We have applied this approach to Turkish, and our results indicate that we can find the intended correct word in 95% of the cases.

References

[1] I. Aduriz and 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*, April 1993.

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

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

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

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

[6] L. Karttunen. KIMMO: a general morphological processor. *Texas Linguistic Forum*, 22:163 – 186, 1983.

[7] L. Karttunen and K. R. Beesley. Two-level rule compiler. Technical Report, XEROX Palo Alto Research Center, 1992.

[8] J. Y. Kim and J. Shawe-Taylor. An approximate string-matching algorithm. *Theoretical Computer Science*, 92:107–111, 1992.

[9] K. Koskenniemi. Two-level morphology: A general computational model for word form recognition and production. Publication No: 11, Department of General Linguistics, University of Helsinki, 1983.

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

[11] K. Oflazer. Two-level description of Turkish morphology. In *Proceedings of the Sixth Conference of the European Chapter of the Association for Computational Linguistics*, April 1993.

[12] A. Solak and K. Oflazer. Parsing agglutinative word structures and its application to spelling checking for Turkish. In *Proceedings of the* 15th *International Conference on Computational Linguistics*, volume 1, pages 39 – 45, Nantes, France, 1992. International Commitee on Computational Linguistics.

[13] A. Solak and K. Oflazer. Design and implementation of a spelling checker for Turkish. *Linguistic and Literary Computing*, 8(3), 1993.

[14] R. Sproat. *Morphology and Computation.* MIT Press, 1992.

[15] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the Association for Computing Machinery*, 21:168–173, 1974.