Typographical Nearest-Neighbor Search in a Finite-State Lexicon and Its Application to Spelling Correction

Agata Savary

LADL, IGM, Université de Marne-la-Vallée
5, bd Descartes, Champs-sur-Marne, 77454 Marne-la-Vallée, France
xsavary@free.fr

Abstract. A method of error-tolerant lookup in a finite-state lexicon is described, as well as its application to automatic spelling correction. We compare our method to the algorithm by K. Oflazer [14]. While Oflazer's algorithm searches for all possible corrections of a misspelled word that are within a given similarity threshold, our approach is to retain only the most similar corrections (nearest neighbours), reducing dynamically the search space in the lexicon, and to reach the first correction as soon as possible.

1 Introduction

K. Oflazer [14] proposed an efficient and elegant algorithm of error-tolerant lookup in a finite-state dictionary, and its application to morphological analysis and spelling correction of simple words. For a given input string that is not contained in the dictionary the algorithm searches for all possible corrections that are within the given distance threshold. We present a similar method in which only those candidates are retained that have the minimal distance from the input word, and the first solution can be obtained rapidly.

2 Related Work

Many aspects of a natural language can be treated through finite-state machines in their classical [16, 7] and extended [8] versions, due to their time and space efficiency obtained by determinisation and minimisation [19, 13, 2].

Automatic spelling correction is one of the oldest applications in the field of natural language processing, and it has a very rich bibliography, a good review of which is presented in [9]. The author divides the existing approaches into three classes: nonword error detection, isolated-word error correction, and context-dependent word correction. Many problems faced by the methods of the first class in the early research (e.g. [12], due to the size of the lexicon and its access time, found a solution in the finite-state model of the lexicon. One of the main remaining problems, the recognition of spelling errors resulting in valid words

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 251–260, 2002.

(e.g. *from* → *form*) requires approaches of the third class, based most of the time on a syntactic and/or stochastic analysis of a local context of words supposed to be erroneous (e.g. [17, 5].

In the second type of approach, i.e. isolated error correction, errors are most often of typing origin, of phonetic origin (e.g. [10], or both. This paper addresses only typing errors. They are traditionally interpreted as resulting from one or more *editing operations* on letters: insertions, deletions, replacements and inversions of adjacent letters [3]. Their correction is related to the theoretical problem of approximate string matching [6], in which the distance between two strings is the minimum cost of all sequences of editing operations that transform one string into another. Different sequences of editing operations may be allowed and different cost functions may be assigned to these editing operations. With the distance measure called *edit distance* proposed in [18, 11], editing operations may be assigned arbitrary costs, and they may act on arbitrary positions in the string in arbitrary order (e.g. *ca* can be obtained from *abc* by two operations: deletion of *b*, inversion of *a* and *c*). However, an efficient algorithm for edit distance calculation exists only if $W_I + W_D \leq 2W_S$, where W_S, W_I, W_D are costs assigned to inversion, insertion and deletion operations, respectively.

In [4] this distance measure is modified and renamed to *error distance* by assigning cost 1 to each editing operation and by admitting that errors occur in linear order from left to right so that a later operation may not cancel the effect of an earlier operation. Thus, inversions occur only between letters that are adjacent in the original word and remain adjacent in the erroneous word (e.g. the error distance between *abc* and *ca* is 3). Due to the equal cost of each editing operation, the error distance becomes a *metric*, i.e. a function satisfying four properties: non-negative values, reflexivity, symmetry, and triangular inequality.

The computational solution for the (editing or error) distance calculation, belonging to the class of *dynamic programming* algorithms, is based on a matrix *H[0:n,0:m]*, where n and m are the lengths of the two strings to be compared, and *H[i,j]* contains the distance between the prefixes of lengths i and j of the two strings. The calculation is particularly efficient for the error distance matrix, since the value of the element *H[i+1,j+1]* depends only on the values of the elements *H [i-1,j-1]*, *H[i,j]*, *H[i+1,j]*, and *H[i,j+1]*. Oflazer [14] made the calculation of the error distance matrix even more efficient in that he applied it to the finite-state representation of the lexicon. Thus, when a word is searched for in the lexicon, a part of the matrix is calculated only once for all lexicon words that have the same common prefix.

3 Spelling Correction Problem

The distance measure between two strings admitted in this paper, as well as in Oflazer's one, is the error distance of Du and Chang [4] (although Oflazer still uses the notion of *edit distance*), as described in the previous section. There is no theoretical distance limit between an erroneous word and its right correction. Hence a trade-off is necessary between three factors: the search time efficiency

(in the case of our algorithm and of Oflazer's one it corresponds to the size of the section of the automaton that is to be explored), the length of the resulting correction candidate list (the user may be unwilling to consult a long list), and the chance that the intended word be on that list. Thus, two of the possible spelling correction problem definitions are:

- Finding all valid words which are no more distant from the input word than a given threshold.
- Finding the nearest-neighbours, i.e. the valid words with the minimal distance from the input word (the minimal distance possibly being no bigger than a given threshold).

Note that none of the two approaches guarantees that the right correction will be found. The first approach is more often admitted (e.g. in [4, 14] since the right correction candidate for a misspelled word may not be its nearest neighbour. In our opinion, the second approach is preferable for many applications for three reasons: statistical studies show that words with multiple errors are rare (0.17% till 1.99% of unknown words in a corpus, with [15], users are easily discouraged by long lists of correction candidates, and the search time grows exponentially with the admitted distance threshold. Therefore, the tolerant lookup algorithm we propose finds only the nearest neighbours and concentrates on reaching the first solution (which often is the right one) as soon as possible.

4 Example

The interpretation of a spelling error can be ambiguous. For instance, the erroneous English word **aply* has some one-operation corrections: *apply* (omission of *p*), *paly* (inversion of *p* and *a*), *ply* (insertion of *a*), some 2 two-operation corrections: *ape* (replacement of *e* by *l*, insertion of *y*), *apple* (omission of *p*, replacement of *e* by *y*), *pale* (inversion of *p* and *a*, replacement of *e* by *y*), some three-operation corrections: *apples* (omission of *p* and *s*, replacement of *e* by *y*), *pales* (inversion of *p* and *a*, replacement of *e* by *y*, omission of *s*), etc.

We will show how the nearest neighbours with threshold 2 (in our example these are the one-operation corrections) can be found by an error-tolerant look-up in a deterministic finite-state lexicon. Let us consider a small extract of English lexicon of simple words, containing some possible corrections of **aply* (Fig. 1). The terminal states are represented by double circles. We say that state *w* is reachable from state *v* if there is a transition leading from *v* to *w*. The algorithm follows at first the standard look-up procedure to find the longest correct prefix. It begins in the initial state number 1. Parsing from left to right of *aply* brings us to a non-terminal state 4, where reading the input letter *l* is not possible. Since the automaton is a deterministic one, no backtracking is necessary to be sure that the parsed sequence is not contained in the lexicon. That is where we start the error-tolerant look-up procedure searching for similar words through admission of any of the 4 elementary operations at any of the 5 possible positions in the erroneous word:

a	p	l	y	
1	2	3	4	5

At word position 3, where the standard input blocked, we can make the following suppositions:

- Letter *l* has been wrongly inserted. We omit *l* and try to recognize suffix *-y* starting from the current state 4. That is not possible, so we have to make a second supposition about a possible error. Apart from the wrong insertion of *l*, we may simultaneously have:
  - Wrong insertion of *y*. We try to recognize the empty suffix starting from the current state 4. That is not possible since this state is not a terminal one. No more supposition about a possible error is allowed since we reached the admitted threshold 2.
  - Omission of the correct letter before *y*. We try to recognize suffix *-y* starting from all states reachable from state 4. That is not possible without any further error admission.
  - Replacement of the correct letter by *y*. We consider all transitions leading from state 4 to a final state. There is one such transition: (4,*e*,9). Thus, we get the first two-operation correction candidate *ape* with the error distance 2.
- Letters *l* and *y* have been wrongly inverted. We try to recognize the inverted suffix *-yl* starting from the current state 4 and considering a possible omission at the end of the word (no second error is admitted between *y* and *l* due to the condition that inverted letters must remain adjacent in the target word). That is impossible.
- The correct letter has been omitted at the current position 3. We try to recognize suffix *-ly* starting from any state that is reachable from state 4. In state 9 the recognition of *-ly* is not possible with no more than 1 further error supposition. In state 5 the recognition of *-ly* is possible with no further error supposition, which yields a new 1-operation candidate *apply*. The error distance threshold is reduced to 1. Therefore the 2-operation candidate *apple* is not reached and the previously obtained candidate *ape* is eliminated as it is more distant from the original word than the new candidate.
- The correct letter at position 3 has been replaced by *l*. We try to recognize the suffix *-y* from any state that is reachable from state 4. That is not possible without any further modification. Since the new threshold is 1 this supposition is eliminated.

To continue searching for other candidates we have to backtrack from state 4 to state 2 (and from word position 3 to 2), where 4 possible hypotheses are analysed again: wrong insertion of *p* (suffix *-ly* is unrecognizable from state 2, no candidate is proposed), wrong inversion of *l* and *p* (suffix *-lpy* is unrecognizable), omission of the right letter at position 2 (the only state reachable from state 2 is 4, from which the suffix *-ply* is recognizable yielding the same candidate *apply* as previously obtained), replacement of the right letter at position 2 by *p* (impossible since there is only one transition from 2 to 4).

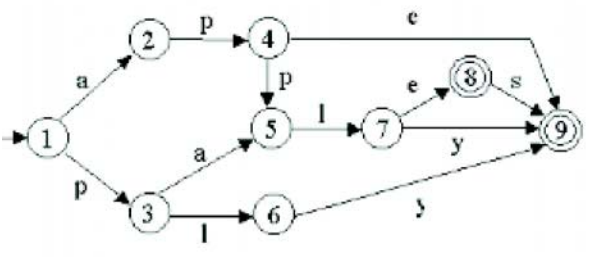

Fig. 1. Extract of a final state lexicon

Finally, backtracking from state 2 to state 1 (and from word position 2 to 1), yields two more one-operation candidates *paly* and *ply*. The two- and three-operation candidates *pale* and *pales* are not reached due to the reduced threshold.

5 Algorithm

An outline of our error-tolerant finite-state lookup algorithm is shown on figure Fig. 5. Let $[l_1 l_2 ... l_n]$ be the word to be looked up, and n its length. Let $wp = 1, ..., n+1$ be the current word position. Let st be the current state, and t the error distance threshold between two suffixes (i.e. the number of elementary operations that we admit in a correct suffix so that it may still be considered a valid correction candidate for a misspelled suffix).

The *tolerant_lookup* function tries to recognize the suffix $[l_{wp}...l_n]$ starting from the current state st and admitting t elementary operations on letters at most. This function returns a pair (ed, S) where S is the set of recognized (exact or modified) suffixes, and ed is the edit distance between the suffix $[l_{wp}...l_n]$ and each of the suffixes in S (all suffixes in S always have the same edit distance from $[l_{wp}...l_n$; if S is empty then $ed = INF$ (a large number, bigger than the maximum edit distance ever possible). The first call to *tolerant_lookup* is done for the entire word $[l_1...l_n]$, the initial state, and the desired edit distance threshold. Then we follow the standard look-up procedure, first without admitting any operation on letters. Thus we can immediately recognize the input word if it belongs to the lexicon, and then quit (the threshold value t becomes 0 in line 9 and lines 13–36 are omitted). If the word doesn't belong to the lexicon the standard look-up ends up with failure in one of the two cases: 1) the input sequence has been read in and the last state is not a terminal one, 2) the input sequence has not been read in completely and no further transition from the current state is possible. If the exact suffix $[l_{wp}...l_n]$ couldn't be recognized, t remains positive (code line 9) and we admit that an error occurred at position wp in the intended word. We try to recognize the input suffix $[l_{wp}...l_n]$ by admitting one of the four elementary operations:

```
1. tolerant_lookup ([l_wp ... l_n],st,t)
2. begin
3. S ← ∅; ed ← INF;
4. if (wp > n)
5.    if terminal(st) then return (0,{ϵ}); endif;
        /*the empty suffix recognized*/
6. endif;

        /*look up the exact suffix, reduce the threshold so as not to admit more
        modifications than in the suffixes already found*/
7.    if (wp ≤ n) and (there is a transition (st,l_wp,st_s))
8.      (ed,S)← tolerant_lookup([l_wp+1 ... l_n],st_s,t);
9.      t = min(t,ed);
        /*concatenate the current letter l_wp
        with all suffixes similar to [l_wp+1 ... l_n]*/
10.     for each (suff ∈ S) do suff ← l_wp ∘ suff; endfor;
11.   endif ;

        /*look up modified suffixes*/
12.   if (t>0)

        /*suppose an insertion at position wp*/
13.     if (wp ≤ n)
14.        (ed_n,S_n)← tolerant_lookup([l_wp+1 ... l_n],st,t-1);
        /*only the suffixes with the smallest edit distance are retained*/
15.        (ed,S) ← add_or_replace(ed,S,ed_n+1,S_n);
16.        t = min(t,ed);
17.     endif;

        /*suppose an inversion of letters at positions wp and wp+1,
        these letters must remain adjacent*/
18.     if ((wp < n) and (l_wp ≠ l_wp+1))
19.        if (∃ ((st,l_wp+1,st_s) and (st_s,l_wp,st_v)))
20.          (ed_n,S_n)← tolerant_lookup([l_wp+2...l_n],st_v,t-1);
21.          for each (suff ∈ S_n) do suff ← [l_wp+1 l_wp] ∘ suff;
22.          endfor;
23.          (ed,S) ← add_or_replace (ed,S,ed_n+1,S_n);
24.          t = min(t,ed);
25.        endif; endif;

26.     for each transition (st,l,st_s)

        /*suppose an omission of a letter at position wp*/
27.        (ed_n,S_n)← tolerant_lookup([l_wp...l_n],st_s,t-1);
28.        for each (suff ∈ S_n) do suff ← l ∘ suff; endfor;
29.        (ed,S) ← add_or_replace (ed,S,ed_n+1,S_n);
30.        t = min(t,ed);

        /*suppose a replacement of the right letter by l_wp if the word not finished*/
31.        if (wp ≤ n)
32.          (ed_n,S_n)← tolerant_lookup([l_wp+1...l_n],st_s,t-1);
33.          for each (suff ∈ S_n) do suff ← l ∘ suff; endfor;
34.          (ed,S) ← add_or_replace (ed,S,ed_n+1,S_n);
35.          t = min(t,ed);
36.        endif; endfor; endif;
37. return(ed,S); end
```

Fig. 2. Error-tolerant lookup algorithm

- Insertion of the letter l_{wp} (if we haven't read the whole word yet; lines 13–17). We omit letter l_{wp} and try to recognize the suffix $[l_{wp+1}...l_n]$ starting from the current state st. We retain only the best solutions (see comment on function *add_or_replace* below).
- Inversion of letters at positions wp and $wp+1$ (if at least two letters are left; lines 18–25). First we try to recognize the inverse infix $[l_{wp+1}l_{wp}]$ starting from the current state st and allowing no modification because we require that inverted letters must remain adjacent. Then we try to recognize the suffix $[l_{wp+2}...l_n]$ starting from the arrival state stv.
- Omission of a letter at position wp (lines 27–30). For each transition leading from the current state st to a state st_s through a label l (line 26), we try to recognize the current suffix $[l_{wp}...l_n]$ starting from the state st_s. Each solution found is concatenated with the transition label l (line 28).
- Replacement of the right letter at position wp through letter l_{wp} (if we haven't read the whole word yet; lines 31–36). For each transition leading from the current state st to a state st_s through label l, we try to recognize the suffix $[l_{wp+1}...l_n]$ starting from the state st_s.

Notice that each time new solutions are found the value of ed and the contents of S are updated by the function *add_or_replace* (lines 15, 23, 29, 34). If new solutions are closer to the original word than the solutions already in S then S is replaced by the set of new solutions, and the value of ed by the new error distance. Otherwise the union of the two sets is done and ed remains unchanged. Thus, only those solutions are retained that have the smallest error distance from the original suffix. Then t gets reduced (lines 9, 16, 24, 30, 35), which limits the range of further searches.

The above algorithm can take as parameter any value of the edit distance threshold, but for languages like English and French, which we tested the program with, the reasonable limit seems to be 2 operations because admitting a bigger edit distance would often result in a great number of irrelevant corrections. Besides the look-up time for a high edit distance threshold would require the exploration of a very big section of the automaton, thus making the search time hardly acceptable for large corpus applications (cf section 6).

6 Complexity and Performance

The exact complexity of our error-tolerant look-up algorithm is difficult to find because it depends not only on the word's length, but also on the size of the dictionary and its precise contents (i.e. the number and length of words that have common subsequences with the input word). Nevertheless, we can make some average case estimation. Let n be the length of the input word, t the error distance threshold, and f_{max} the maximal fan-out of the automaton. Let l_{wp} be the current letter in the input word, s the current state, and f_s the fan-out of s. Depending on what modification is admitted parsing of l_{wp} from state s requires at most:

Table 1. Spelling correction performances

Correction time (**ms**)			
correct sequences	one-error sequences	two-error sequences	sequences with more than 2 errors
7	40	211	233

- 1 transition in case of inversion (the transition that matches l_{wp+1});
- no transition in case of insertion (l_{wp} is omitted, we remain in the current state),
- f_s transitions in case of omission or replacement (all transitions starting from s).

In the worst case, i.e. when the threshold is not reduced during the whole look-up, there are at most $n!/t!(n-t)!$ possible distributions of t modifications over n word positions. For each distribution $(1 + 2 * f_{max})^t$ paths at most must be followed, each path being of length $n+t$ at most. Hence, the worst case complexity is

$$O(n!/t!(n-t)! * (n+t) * 2^t * f_{max}^t).$$

In particular, for t=0 we get $O(n)$, for t=1 $O(n^2 * f_{max})$, for t=2 $O(n^3 * f_{max}^2)$, etc.

We have run the algorithm with threshold 2 on three sets of erroneous strings: sequences belonging to the lexicon, sequences containing one spelling error, and sequences containing two spelling errors or more. The average search time results are presented in the table below. Notice that the correction of 2 errors or more is over 5 times longer than of a single error.

7 Comparison with Oflazer's Algorithm

As we've already mentioned, our algorithm and Oflazer's one admit different definitions of the correction problem (cf section 3).

The main difference though is in the way the calculation of the error (edit) distance is done in the two approaches. In Oflazer's algorithm a matrix H is maintained as described in section 2. Each time a transition is followed in the automaton a new column of the matrix is to be calculated by a function of linear complexity. In our approach the error distance calculation is embedded in the algorithm: each time we admit a modification in the standard lookup procedure the error distance increases. This allows us not to maintain the H-matrix but has also the two major disadvantages:

- It is difficult to adapt the error distance calculation to a particular application or language, e.g. by considering phonetically motivated interchanges of certain letters or groups of letters, as it was done in [1] for Polish.

– A correction candidate may be reached several times with different intermediate error distance values. For example while looking up the word **aply** in the lexicon extract from section 4 with the edit distance threshold 3, the correction candidate *ape* would be first recognized twice as a 3-operation candidate: insertion of l + insertion of y + omission of e, and insertion of l + omission of e + insertion of y. Then the same candidate *ape* would be reached by 2 modifications: insertion of l + replacement of e by y, which would invalidate the two previous solutions. That can make us follow the same path in the automaton several times, which is not time-efficient for bigger values of the edit distance threshold.

For applications in which most errors are of 1 or 2 operations, and in which reaching quickly the first solution is important, our algorithm will often be more efficient due to the fact that we first match the longest correct prefix. Note that in a finite state lexicon the fan-out is very big for the states close to the initial state. Oflazer's algorithm explores most of them at the beginning so it may take a longer time before a solution is found. Our algorithm first skips most of those states (unless the error occurred at the initial position) and follows only the exact path. Since most of misspelled words contain only one error, there is a big chance that the point where the exact path was blocked is the position where the error occurred.

8 Conclusion

We have presented a method of typographical nearest neighbour search in a finite-state lexicon and its comparison to a similar algorithm by Oflazer [14]. Our method is designed for applications where only the least distant corrections are looked for and where the first correction is to be reached as soon as possible. Oflazer's algorithm is simpler and more elegant in the sense that the edit distance calculation is independent from the look-up algorithm.

References

[1] Daciuk, J.: Incremental Construction of Finite-State Automata and Transducers, and Their Use in the Natural Language Processing. Ph.D. Thesis, Politechnika Gdanska, Gdansk (1988) 258

[2] Daciuk, J., Mihov, S., Watson, B., Watson, R.: Incremental Construction of Minimal Acyclic Finite State Automata. Computational Linguistics vol. 26(1). MIT Press, Massachusetts (2000) 3–16 251

[3] Damerau, F. J.: A Technique for Computer Detection and Correction of Spelling Errors. Communications of the ACM, Vol. 7(3) (1964) 171–176 252

[4] Du, M. W., Chang, S. C.: A model and a fast algorithm for multiple errors spelling correction. Acta Informatica, Vol. 29. Springer Verlag (1992) 281–302 252, 253

[5] Golding, A., Schabes, Y.: Combining Trigram-based and Feature-based Methods for Context-Sensitive Spelling Correction. Proceedings, 34th Annual Meeting of the Association for Computational Linguistics (ACL), Santa Cruz. Association for Computational Linguistics (1996) 71–78 252

[6] Hall, P., Dowling, G.: Approximate String Matching. ACM Computing Surveys, Vol. 12(4). ACM, New York. (1980) 381–402 252
[7] Kaplan, R., Kay, M.: Regular Models of Phonological Rule Systems. Computational Linguistics, Vol. 20(3). Cambridge, Massachusetts, MIT Press (1994) 251
[8] Kornai, A. (ed.): Extended Finite State Models of Language. Cambridge University Press, Cambridge, UK - New York, USA - Melbourne, Australia (1999) 251
[9] Kukich, K.: Techniques for Automatically Correcting Words in Text. ACM Computing Surveys, Vol. 24(4) (1992) 251
[10] Laporte, E., Silberztein, M.: Vérification et correction orthographiques assistées par ordinateur, Actes de la Convention IA 89 (1989) 252
[11] Lowrance, R., Wagner, R. A.: An Extension of the String-to-String Correction Problem. Journal of the ACM, Vol. 22(2) (1975) 177–183 252
[12] McIlroy, M. D.: Development of a Spelling List. IEEE Transactions on Communications, COM-30(1) (1982) 91–99 251
[13] Mohri, M.: Minimization of sequential transducers. Lecture Notes in Computer Science, Vol. 807. Springer Verlag. Berlin. (1994) 251
[14] Oflazer, K.: Error-tolerant finite state recognition with applications to morphological analysis and spelling correction. Computational Linguistics, Vol. 22(1). MIT Press, Cambridge, Massachusetts (1996) 73–89 251, 252, 253, 259
[15] Ren, X., Perrault, F.: The Typology of Unknown Words: An Experimental Study of Two Corpora. Proceedings, 15th International Conference on Computational Linguistics (COLING), Nantes. International Committee on Computational Linguistics (1992) 408–414 253
[16] Roche. E., Schabes, Y. (eds.): Finite-State Language Processing. MIT Press, Cambridge, Massachusetts (1997) 251
[17] Véronis, J.: Morphosyntactic correction in natural language interfaces. Proceedings, 13th International Conference on Computational Linguistics (COLING), Budapest. International Committee on Computational Linguistics (1988) 708–713 252
[18] Wagner, R. A., Fischer, M. J.: The String-to-String Correction Problem. Journal of the ACM, Vol. 21(1) (1974) 168-173 252
[19] Watson, B.: Taxonomies and Toolkits of Regular Language Algorithms. Ph.D. Thesis, Eindhoven University of Technology, the Netherlands (1995) 251