

An Empirical Study of “Spelling Correction” for Prospecting Application

Wing Ning Li, CSCE Dept. University of Arkansas, Fayetteville, AR 72701 wingning@uark.edu

Abstract

In this paper the problem of Lexicon Look-up with Unknown, a kind of “spelling correction” problem, for prospecting application is introduced. We learned the connection between spelling correction and prospecting from the problem statement of FY 2006 ALAR RFP. The lexicon look-up with unknown problem in the context of prospecting became the class team project for our graduate design and analysis of algorithms course in Spring 2006. We present basic analyses of the problem and approaches that could be investigated further. We present different algorithmic approaches to solve the lexicon look-up with unknown problem and report how some of the spell checkers developed by students in their team projects perform empirically.

1. Introduction

One of the challenges in many database related applications, in businesses and governments, is to find all “related” records in the “database” to a given record. A straightforward solution for this problem would be to analyze the given record against each record in a database one by one and pick the record that “matches” or “relates” to the given record. Usually the analyses are nontrivial and demand significant computing time. The analyses might also need to consult other knowledge bases. For sizable databases or files with hundreds of millions or even billions of records, the program might take hours or days if not weeks when this approach is used [9]. One promising way to speed up the processing is to adopt a two-step approach, where potential candidate records are quickly determined in step one, and detailed analyses are performed in step two. The term *prospecting* sometimes refers to the first step, where candidate records are prospects. This approach may achieve substantial time reduction if prospect set is small (usually bounded by a small number) and can be computed quickly, since the time-consuming analyses are performed on a handful of records instead of the hundreds of millions records in the database.

As a leading company in Customer Data Integration, Acxiom has developed such a prospecting tool and has used it to answer many challenging business questions. Due to the proprietary nature, we will not elaborate in detail on how the tool works. Instead, we will consider one of the abstract problems that the tool needs to solve: the problem of lexicon look-up with unknown in prospecting. Efficient, accurate methods of solving this problem could improve the performance of the current version of the tool.

A lexicon is a list of words. A word is a known word if it is in the list, and an unknown word otherwise. The problem of lexicon look-up with unknown is informally defined as the following.

Lexicon look-up with unknown problem:

Input: a lexicon (once) and a word (many times).

Output: If the input word is a known word, a position in the lexicon at which the input word locates; If the input word is an unknown word, a short list of positions, ranked in the order of confidence, at which the input word might locate if “spelling” errors in the input word are “corrected”.

One way to think of the lexicon look-up problem is to treat it as a dictionary look up problem, where the lexicon is the dictionary. If the word is in the dictionary, the page in which the word shows up is returned. If the word is not in the dictionary, whether the unknown status might be due to “spelling” errors needs to be determined, hence, “spelling” error corrected. Since multiple corrections might be possible, a list of pages where the corrected words show up can be returned, in the order of confidence. This analogy is used to help demonstrate the core issues and questions that we intend to deal with.

Formally, a lexicon, **L**, can be any finite set of strings over a given alphabet. A word, **w**, is just a string. An unknown word could either have nothing to do with the lexicon at all or could be “similar” to some of the words in the lexicon, meaning, some known words with minor variations could become the unknown word. Examples of such simple variations include *insertion*, *deletion*, *substitution*, and *transposition*. For example, unknown word **acress** can be obtained from **acres** (*insertion* of s at the end), **actress** (*deletion* of t), **access** (*substitution* of the second c by r), and **caress** (*transposition* of the first two characters). More complicated or drastic variations may be viewed as combinations of simple variations. For example, **nillier** can be obtained from **miller** by substituting n for m and inserting an i after the second l.

For this paper, we intentionally avoid the introduction of semantic elements. In light of the fact that prospecting is a quick first step of a two-step process, we leave the semantic information and its added value to the second step, though good automatic spelling correction programs usually must consider the semantics and context to make the right decision [3,15]. The above example of **acress** and its four possible corrections **acres**, **actress**, **access**, and **caress** illustrates the challenge of spelling correction without the semantic context. The example also demonstrates the challenge of ranking each word of its likelihood of being the “right” word for **acress**. The

formulation has the advantage that the prospector can be made natural language independent. The string and alphabet treatment makes the problem and its solutions as applicable to English as to other languages.

The lexicon look-up with unknown problem is basically an isolated-word error correction problem [12] that may be broken into four sub problems: i) lexicon representation, ii) known or unknown detection, iii) generation of correction candidates, and iv) ranking of correction candidates.

The problem and its sub problems were given to the students in the design and analysis of algorithms class as a project and students were given the options of forming their own team. Each team worked independently with other teams and was responsible for selecting its own algorithmic approaches to address the sub problems so that a “spell checker” program could be developed and evaluated empirically. Students were pleased to work on an interesting problem and to observe the run time reduction due to clever algorithmic techniques. Some of “spell checker” could do a better job than Microsoft Word and Google search engine on certain unknown words. Details of the empirical studies will be provided in later sections.

Considerable work in automatically correcting “misspelling” has been reported in the literature and different techniques have been developed for different applications [12]. Depending on parameters such as typographic errors (keyboard effects or fat finger effects), cognitive errors, OCR errors, and phonetic errors, one method may exhibit strength over the others. For the isolated-word error correction problem, empirical study on a small test set has been conducted and reported in [12] for Minimum Edit Distance, Similarity Key, Simple N-gram Vector Distance, SVD N-gram Vector Distance, Probabilistic, and Neural Net approaches. The accuracy rate ranged from 52% to 81%, though no run time data was provided.

Due to the performance requirement placed on prospecting, execution speed will be one very important measure of success for any solution to the lexicon look-up problem. Therefore it remains to be seen which of these methods may be a good match to our problem. Also a fundamental difference exists between the isolated-word error correction problem and our lexicon look-up for prospecting problem. In automatic spelling correction, the one correction is sought; whereas in lexicon look-up for prospecting, possible corrections are sought so that the prospect pool contains all those records of interest, and equally importantly the pool size and the time of obtaining the pool are minimized. These similarities and difference will guide us in searching an optimal solution to the problem and evaluate different algorithmic approaches.

The reported work done by our students is an initial step in answering the above questions and lays the groundwork for future study of the problem.

2. Basic Analyses and Approaches

We briefly discuss the issues and review approaches of solving the four sub problems related to lexicon look-up with unknown. The approaches adapted and implemented by our students will be described in more detail later.

2.1 Lexicon Representation

From the definition of lexicon look-up with unknown, the lexicon representation needs to support two basic operations: 1) determining whether a given string is in the lexicon or not, 2) generating correction candidates if any for the given string. The first operation is a search question and the second operation may be viewed a search question as well in that we search for approximately matched strings from the lexicon. In addition to these operations, we may need to consider operations for updating the lexicon such as insertion and deletion. With the assumption that our lexicon is relatively stable and static, efficient support of these update operations will not be taken into consideration in designing the presentation of the lexicon.

Thus, an array of strings is an efficient representation to support the first operation. Using array we can efficiently perform linear search, binary search (when strings are sorted), and hashing (each string is mapped to an index of the array). Most of the “spell checkers” of our students use array representations for the lexicon. Array representations also allow efficient implementation of the second operation when we have to perform linear search to find approximately matched strings to the given string.

For lexicons containing strings of various lengths, an even more efficient representation is to group strings of the same length together. Each group is sorted or hashed and located in one region of the array. Or we may have a separated array for each group. The length of the input string is used as an index to the region or to the array of its group. We would then perform linear search, binary search, or hashing operations with respect to the group. In general each group is much smaller than the whole lexicon, hence, by limiting searches to those strings belonging to the same group, we are able to further reduce the run time in principle. This representation could also speed up the second operation. The empirical study of the “spell checkers” leads to the observation of this representation. Therefore, the representation was not used in any projects. We will study its effectiveness and efficiency empirically in the future.

Since a lexicon is a finite set, the set of lexicon strings is a regular language. We would build a deterministic finite automaton to recognize strings in the lexicon [10]. Then the first operation can be performed with time in proportion to the length of the input string and independent of the size of the lexicon. We will study empirically the effectiveness and efficiency of automaton representations in the future.

2.1 Known or Unknown Detection

Once the lexicon representation is determined, it is relatively straightforward to solve this problem. For unsorted array, linear search has to be performed to compare each string to the given string to see if the given string is a known word. For sorted array, binary search should be used. For hashing, the hash value of the given string is used as an index to location of the array from which additional comparisons might need to be performed depending on the collision resolution schemes used in the hashing. For deterministic finite automaton, each character of the given string is used to move along the transition and if the last state reached is an accepting state, the input string is a known word.

2.3 Generation of Correction Candidates

The best way to “correct” the possible mistakes in input strings is to establish error models of unknown word and to understand error patterns. For example, transcription-typing errors tend to reflect typewriter keyboard adjacency such as substitution of d for e. In contrast, errors introduced by optical-character recognizer (ORC) tend to indicate confusion due to feature similarities such as substitution of D for O. The errors in prospecting application might be classified as (1) typographic errors; (2) phonetic errors; (3) cognitive errors; or other categories. In addition to error category classification, errors can be classified as single errors, where an unknown word results from a single insertion, deletion, substitution, or transposition; double errors, where an unknown word results from any combination of two single errors, and so on. Not knowing any error model for prospecting application and whether such a mode could be derived, we are interested in general methods of determining lexical-similarity between strings. Levenshtein Edit Distance [1,6,7,11,13,14,16,17,19] is the most popular lexical similarity measure. The Edit Distance calculates a score that represents the number of steps required to transform one string into another. The allowed transformations are single character insertion, deletion, and substitution. Other variations may allow additional operations and/or different costs for different operations. Edit Distance was used by all the teams and we will consider it in more detail later.

For English, Soundex and Metaphone are other similarity measures between words based on the sound of the words. Several projects combined edit distance method with Soundex or Metaphone methods.

Soundex algorithm uses codes based on each letter to transform a word or string into a code of four characters. It capitalizes all letters; retains the first character (the mapping is not used for it); maps 7 groups of characters to digits 0 to 6; adjusts digits in the resulting mapped code; and returns first four positions. The idea of the algorithm is that similar sounding words should produce the same codes [8,20]. For example, ‘reynold’ and ‘re naud’ produce the code R543 [20].

Metaphone algorithm is a step up from the relatively simple Soundex algorithm. It takes a word and returns a very rough approximation of the sound of that word. It takes into account that the English pronunciation of letters depends on their neighbors. The basic idea is to transform a word into a variable length code of up to four characters and similar sounding words should map to the same code, just like what the Soundex algorithm does. The reader is referred to [18] for details of the algorithm.

For Edit Distance method, correction candidates are those strings of which the edit distances to the input string are less than certain threshold. For Soundex and Metaphone methods, correction candidates are those strings that map to the same code as the input string does. Codes of a lexicon should be computed in a preprocessing step since the lexicon strings have been given.

Methods [12] such as “reverse” minimum edit distance techniques, rule-based techniques, n-gram-based techniques, probabilistic techniques, and neural nets and support vector machine [2,4] could be useful for generation of correction candidates but were not investigated nor implemented by any student team.

2.4 Ranking of Correction Candidates

Since the algorithms chosen for generating correction candidates are edit distance, Soundex, and Metaphone, the outcomes from these algorithms are used to rank the candidates. For example, strings with smaller edit distances rank higher than that with larger edit distance. For approaches combining multiple algorithms, strings that are given the same rank by the first algorithm are ranked again by the second algorithm.

3. Edit Distance Computation.

Given two strings $x[1..n]$ and $y[1..m]$, the Levenshtein edit distance between x and y is the minimum number of insertion, deletion, and substitution operations needed to transform x to y (or y to x).

Let $x[1], x[2], \dots, x[n]$ be the first, second, ..., n^{th} character in x respectively, and let $y[1], y[2], \dots, y[m]$ be the first, second, ..., m^{th} character in y respectively. The sub string of x that begins at the first character and ends at the i th character is denoted by $x[1..i]$. Similarly, the sub string of y from the first character to the j th character is denoted by $y[1..j]$. Let $\text{dist}[i][j]$ be the edit distance between $x[1..i]$ and $y[1..j]$. The edit distance between x and y is $\text{dist}[n][m]$.

Let $\text{equal}[i][j]$ be 1 if $x[i]$ and $y[j]$ are the same, and 0 otherwise. The following recurrence captures the computation of edit distance and can be computed using dynamic programming technique.

$$\text{dist}[i][j] = \min\{\text{dist}[i-1][j-1] + \text{equal}[i][j], \text{dist}[i-1][j]+1, \text{dist}[i][j-1]+1\}$$

where initially $\text{dist}[0][k]=k$ for k from 0 to m and $\text{dist}[k][0]=k$ for k from 0 to n .

The three expressions within the min operator account for the cost of substitution, deletion, and insertion of $x[i]$ to transform $x[1..i]$ to $y[1..j]$. The following algorithm is a straightforward implementation of the recurrence from bottom up.

```

For (k=0, k<=m, k++) dist[k][0]=k; //initialization
For (k=0, k<=n,k++) dist[0][k]=k;//initialization
For (i=1, i<=n ,i++) // main loop of computation
    For (j=1, j<=m, j++)
        dist[i][j] =min {dist[i-1][j-1] + equal[i][j],
            dist[i-1][j]+1,                dist[i][j-1]}

return dist[n][m] // the result of edit distance

```

Let $x='acress'$ ($n=6$) and $y='actress'$ ($m=7$). The following table of $dist[][]$ shows the computation of the algorithm, which fills in the table one row at a time.

dist	y	A	C	T	R	E	S	S
x	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
C	2	1	0	1	2	3	4	5
R	3	2	1	1	1	2	3	4
E	4	3	2	2	2	1	2	3
S	5	4	3	3	3	2	1	2
S	6	5	4	4	4	3	2	1

In the above table, the letters following y horizontally show the value of y, and the letters following x vertically show the value of x. Numerical values correspond to the computed edit distant values. The numerical values horizontally following x is computed in the first initialization loop and may be viewed column indices of $dist[][]$, even though they the values of $dist[0][0]$, $dist[0][1]$, ..., and $dist[0][7]$. The numerical values vertically following y is computed in the second initialization loop and may be viewed row indices.

Since $dist[6][7]$ is 1, the edit distance between x and y is 1. The one edit operation that transforms x into y is the insertion, which inserts t after ac. Notice that $dist[2][3]$ is also 1, which is the edit distance between 'ac' and 'act'.

Let us consider a few more entries. $dist[3][3]$ is 1, which implies the edit distance between 'acr' and 'act' is 1. The edit operation that transforms 'acr' to 'act' is the substitution where 't' is substituted for 'r'. This operation, however, is part of the minimum operations that transform 'acress' to 'actress'. Since $dist[3][7]=4$, the edit distance between 'acr' and 'actress' is 4. Notice that 4 also happens

to length difference of the two strings. The length of 'acr' is 3 and that of 'actress' is 7. The generalization of this case is that the edit distance of two strings x and y, of lengths n and m respectively, cannot be smaller than $m-n$ (with out loss of generality assuming $m \geq n$), and $m-n$ is achievable only if y contains all the characters of x and for each character showing up in both x and y its multiplicity in y is no smaller that its multiplicity in x.

Hence, when the difference in length between two strings is no less than the given threshold value, we can reject the string in the lexicon as a correction candidate without actually computing the edit distance since the edit distance is guaranteed to no less than the threshold. This observation was used and led to substantial reduction in run time.

Another observation from the above table, even though it is not that obvious, is that the minimum value of next row cannot be smaller than that of the previous row. The same also holds for columns. For example, row 3 has minimum of 1 and row 2 has minimum of 0. Based on this, when the minimum value of just completed row is no less than the threshold, the decision of rejecting the string in the lexicon can be made without finishing the remaining computation. This observation was used and led to additional reduction in run time.

The banding effect where values are smaller along the diagonal shown in the table is true in general and could be used to limit the number of computed entries. This technique was not investigated or implemented by any student team. It will be investigated empirically in the future.

The whole table is needed in order to transform one string to another using a minimum number of edit operations. If we are only interested in the edit distance as in generating correction candidates, however, only two rows are needed based on the $dist[][]$ recurrence. In fact, the algorithm can be implemented using a single row of memory. The development of that algorithm is left for the reader. This technique was not investigated or implemented by any student team. It will be investigated empirically in the future.

4. Experiments

Since we were not able to obtain lexicons from business prospecting application, we had to use various English Dictionaries available from the operating systems and the Internet, and sets of cDNA strings as our lexicons. For English Dictionaries, the lexicon sizes are 25,000 (small), 52,000 and 96273 (large), and 377,000 and 616,500 (huge). For cDNA, the lexicon sizes are 100, 500, and 1000 and each string in cDNA averages 2700 characters long.

The programming languages used to implement the "spell checker" prototypes are Java, C, and C++. Due to variations in design alternatives, programming languages,

lexicons, and runtime environments under which the prototypes were developed and tested, the run time characteristic of the prototypes varied widely. For better prototypes, the time for generating correction candidates was under milliseconds. In general, for a given threshold the techniques of using length information and premature loop termination could reduce the run time by at least 50% compared to not using such techniques.

Students were eager to see how their prototypes compared with MS-WORD and GOOGLE search engine. Here are a few test cases of one prototype:

1. Input word is "lood"
 - Candidates from our program (top 5): blood, bood, flood, food, good;
 - Candidates from MS-WORD: loud, load, loot, blood, lord
 - Candidates from GOOGLE: lood
2. Input word is "mapple"
 - Candidates from our program with small lexicon (top 5): apply, maple,dapple,apple,grapple;
 - Candidates from our program with huge lexicon (top 5) :capple, dapple, apple, happple, maple
 - Candidates from MS-WORD: apply, maple, maples, ample
 - Candidates from GOOGLE: maple
4. Input word is "moleculr"
 - Candidates from our program (top five): bimolecular, molehill, molecule, molecular, monocular;
 - Candidates from MS-WORD: no candidate
 - Candidates from GOOGLE: molecular
5. Input word is "abcdefghijklmnpqrst"
 - Candidates from our program (top five): no candidate;
 - Candidates from MS-WORD: no candidate
 - Candidates from GOOGLE: no candidate

5. Conclusions

This paper introduces and formalizes the lexicon look-up with unknown problem, which is further divided into four sub problems. Various approaches for solving these sub problems have been surveyed and in depth analyses have been provided for some approaches. A preliminary study of the lexicon look-up with unknown problem was conducted and several prototypes were built and empirically studied. Techniques of speeding up Levenshtein edit distance computation with respect to a given threshold have been proposed of which some are empirically evaluated. The prototypes were compared to

MS-WORD and GOOGLE search engine for the ability of generating correction candidates.

The future work of the current investigation includes using lexicons from prospecting application to evaluate the prototypes; developing a new prototype that incorporates good ideas from the current prototypes as well as new ideas; and developing more accurate error models.

Acknowledgments

I would like to thank Tom Schweiger of Axiom for bringing the prospecting problem to my attention. I would also like to thank all the students in the CSCE5033 design and analysis of algorithms in Spring 2006: Suman Bharath, Roay Cabaniss, Rachil Chandran, Senthikumar Chinnappa-Gounderp, Wesley Deneke, Joshua Eno, Donald Hayes, Evan Kirkconnell, Swathi Musunuri, Linh Ngo, Naveen Ramanathan, Hadi Sabaa, Jaauns Uudmae for their participation and contribution to the "spell checker" projects. The empirical study reported is a summary of their projects.

References

- [1] R. Baeza-Yates and G. Gonnet. "A new approach to text searching," CACM 35, 10, 74-82, 1992.
- [2] Cherkassky and F. Mulier. "Learning from data: concepts, theory, and methods. Wiley, 1998.
- [3] Davidson. "Retrieval of misspelled names in an airline passenger record system," CACM 5, 169-171.
- [4] R. Duda et al. Pattern Classification. 2nd edition, Wiley, 2001.
- [5] E. Fox, Q. Chen, and L. Heath. "A faster algorithm for constructing minimal perfect hash functions," Proc. 15th Annual International SIGIR meeting, 266-273, 1992.
- [6] Z. Galil and R. Giancarlo. "Data structures and algorithms for approximate string matching," J. Complexity 4, 33-37, 1988
- [7] G. Gonnet and R. Baeza-Yates. Handbook of algorithms and data structures. Addison-Wesley, 2001.
- [8] P. Hall and G. Dowling. "Approximate string matching," ACM Computing Survey, 12, 4, 381-402,1980.
- [9] M. Hernandez and S. Stolfo. "The merge/purge problem for large database," Proc. 1995 ACM SIGMOD International Conference on Management of Data, 127-138, 1995.
- [10] Hopcroft et al. Intro. to automata theory, languages, and computation. Addison Wesley, 2001
- [11] Jokinen, P., Tarhio, .J, Ukkonen, E. (1996) "A comparison of approximate string matching

algorithms”, *Software Practice and Experience*, 26(12): pp. 1439-1458.

- [12] K. Kukich. “Techniques for Automatically Correcting Words in Text,” *ACM Computing Survey*, 24, 4, 377-439, 1992.
- [13] G. Landau and U. Vishkin. “Fast parallel and serial approximate string matching,” *J. Algorithm* 10, 158-169, 1989.
- [14] U. Manber. “A text compression scheme that allows fast searching directly in the compressed file,” *ACM Trans. on Information Systems*, 15, 2, 124-136, 1997.
- [15] E. Mays et al. “Context based spelling correction,” *Inf. Process. Manage.* 27, 5, 517-522, 1991.
- [16] E. Moura et al. “Fast and flexible word searching on compressed text,” *ACM Trans. on Information Systems*, 18, 2, 113-139, 2000.
- [17] Navarro, G., Baeza-Yates, R., Sutinen, E., Tarhio, J. (2001) “Indexing methods for approximate string matching” *IEEE Data Engineering Bulletin* 24(4): pp. 19-27.
- [18] L. Philips. “Hanging on the Metaphone,” *Computer Language*, Vol. 7, No. 12, 1990.
- [19] S. Wu and U. Manber. “Fast text searching allowing errors,” *CACM* 35, 10, 83-91, 1992.
- [20] J. Zobel and P. Dart. “Phonetic String Matching: Lessons from Information Retrieval,” *Proc. 19th Int. Conf. On Research and Development in Information Retrieval (SIGIR’96)*, 166-172, 1996.