Dictionary-Based Syntactic Pattern Recognition Using Tries

B. John Oommen[1] and Ghada Badr[2]

[1] Carleton University, *Fellow of the IEEE*
School of Computer Science, Carleton University, Ottawa, Canada : K1S 5B6
oommen@scs.carleton.ca
[2] Carleton University, Ph.D student
School of Computer Science, Carleton University, Ottawa, Canada : K1S 5B6
badrghada@hotmail.com

Abstract. This paper deals with the problem of estimating a transmitted string X^* by processing the corresponding string Y, which is a noisy version of X^*. We assume that Y contains substitution, insertion and deletion errors, and that X^* is an element of a finite (but possibly, large) dictionary, H. The best estimate X^+ of X^*, is defined as that element of H which minimizes the Generalized Levenshtein Distance $D(X, Y)$ between X and Y, for all $X \in H$. All existing techniques for computing X^+ requires a separate evaluation of the edit distances between Y and every $X \in H$. In this paper, we show how we can evaluate $D(X, Y)$ for every $X \in H$ simultaneously, without resorting to any parallel computations. This is achieved by resorting to the use of an additional data structure called the Linked List of Prefixes (LLP), which is built "on top of" the trie representation of the dictionary. The computational advantage (for a dictionary made from the set of 1023 most common words augmented by computer-related words) gained is at least 50% and 80% measured in terms of the time and the number of operations required respectively. The accuracy forfeited is negligible.

1 Introduction

We consider the traditional problem involved in the syntactic Pattern Recognition (PR) of strings, namely that of recognizing garbled words (sequences). Let Y be a misspelled (noisy) string obtained from an unknown word X^*, which is an element of a finite (but possibly, large) dictionary H, where Y is assumed to contain Substitution, Insertion and Deletion (SID) errors. Various algorithms have been proposed to obtain an appropriate estimate X^+ of X^*, by processing the information contained in Y.

Damarreau [3], [13], [19] was probably the first researcher to observe that most of the errors found in strings were either a single substitution, insertion, deletion or a reversal (transposition) error. Thus the question of computing the dissimilarities between strings was reduced to that of comparing them using these edit transformations. In much of the existing literature, the transposition operation has been modelled as a sequence of a single insertion and deletion.

A. Fred et al. (Eds.): SSPR&SPR 2004, LNCS 3138, pp. 251–259, 2004.

The first breakthrough in comparing strings using the three (the SID) edit transformations was the concept of the Levenshtein metric introduced in coding theory [11], and its computation. The Levenshtein distance, $D(X, Y)$, between two strings, X and Y, is defined as the minimum number of edit operations required to transform one string to another. Many researchers, among whom are Wagner and Fisher [23], generalized it by using edit distances which are symbol dependent, and can be perceived as a metric [3], [19], [20]. The latter distance goes by many names, but we shall call it the Generalized Levenshtein Distance (GLD). The GLD has also been studied for parameterized [4],[17] inter-symbol distances. Wagner and Fisher [23] and others [19] also proposed an efficient algorithm for computing this distance by utilizing the concepts of dynamic programming. This algorithm is optimal for the infinite alphabet case. Various amazingly similar versions of the algorithm are available in the literature, a review of which can be found in [3], [19], [20]. Masek and Paterson [12] improved the algorithm for the finite alphabet case, and Ukkonen [21] designed solutions for cases involving other inter-*substring* edit operations. Related to these algorithms are the ones used to compute the Longest Common Subsequences (LCS) of two strings [3], [8], [19], [20]. String correction using GLD-related criteria has been done for noisy strings [3], [18], [19], [20], substrings [19], [20], and subsequences [13], and also for strings in which the dictionaries are treated as grammars [19], [20], [22]. Besides these, various probabilistic methods have also been studied in the literature [2], [18]. Indeed, more recently, probabilistic models which *attain the information theoretic bound* have also been proposed [15], [16].

All the algorithms proposed earlier for estimating X^+, requires the separate evaluation of the edit distance between Y and every element of $X \in H$. However, they do not generally utilize the information it has obtained in the process of evaluating any one $D(X_i, Y)$, to compute any other $D(X_j, Y)$. Suppose X_i and X_j have the same prefix $X^{(P)} = a_1 a_2 ... a_p$. Then, previous algorithms would compute the distance $D(a_1 a_2 ... a_p, Y)$ for both of X_i and X_j, and would thus unnecessarily repeat the same comparisons and minimizations for the substring $a_1 a_2 ... a_p$ and *all its prefixes*. Thus, the previous algorithms usually, have many redundant computations.

The first pioneering attempt to avoid the repetitive computations for a finite dictionary, was the one which took advantage of this prefix information, as proposed by Kashyap *et al.* [10]. The authors of [10] proposed a *set-based* algorithm which we refer to as *Algorithm Prefix-Set-Based*, to compute $X^+ \in H$, which minimizes $D(X, Y)$ for a given Y. In contrast to the previous algorithms, in Algorithm Prefix-Set-Based, $D(X, Y)$ was *not* individually evaluated for every $X \in H$. Rather, it calculated $D(X, Y)$ for all $X \in H$ simultaneously, and this was done by treating the dictionary as one integral unit and by using "dictionary-based" dynamic programming principles, as explained presently. It thus took maximum advantage of the information contained in the prefixes of the words of the dictionary. However, the algorithm in [10] was computationally expensive, as we shall see presently, because it required *set-based* operations in its entire execution.

In this paper, we shall show how we can get a feasible implementation for the concepts introduced in [10]. This is achieved by the introduction of a new data structure called the Linked Lists of Prefixes (LLP), which can be constructed when the dictionary is represented using a trie. The LLP is an enhanced, but modified, representation of the trie, which can be used to facilitate the "dictionary-based" dynamic programming calculations. The LLP-based algorithm for the syntactic PR of strings has been rigorously tested. The dictionaries are subsets of a file consisting of the 1023 most common words augmented by words used in the computer literature. The algorithm was tested by recognizing noisy strings generated using the model discussed in [15]. Numerous experiments were done using these noisy strings to compare the accuracy, the time and the number of computations required by the LLP-based enhanced method, and the sequential current-day algorithms. The results demonstrate that by forfeiting a negligible PR accuracy, we can often reduce the time and the number of operations by about 50% and 80% respectively.

In terms of notation, A is a finite alphabet, H is a finite (but possibly large) dictionary, and μ is the null string, distinct from λ, the null symbol. The left derivative of order one of any string $Z = z_1 z_2 \dots z_k$ is the string $Z_p = z_1 z_2 \dots z_{k-1}$. Z_g, the left derivative of order two of Z, is the left derivative of order one of Z_p, and so on. Also, in the interest of brevity, the pertinent results are merely cited here. Their details can be found in [14].

2 Individual and Dictionary-Based Computations

In string-processing applications, the distance metrics employed traditionally quantify $D(X, Y)$, the minimum cost of transforming one string , X, into the other. Y. This distance is intricately related to the costs associated, typically with the individual edit operations, the SID operations. As mentioned earlier, these inter-symbol distances can be of a 0/1 sort, parametric [4], [17] or entirely symbol dependent [10], [19], in which case, they are usually assigned in terms of the confusion probabilities. In all of these cases, the primary dynamic programming rule used in computing the inter-string distance $D(X, Y)$ is:

$$\begin{aligned} D(x_1 \dots x_N, \quad y_1 \dots y_M) = \min [& \{D(x_1 \dots x_{N-1}, \quad y_1 \dots y_{M-1}) + d(x_N, y_M)\}, \\ & \{D(x_1 \dots x_N, \quad y_1 \dots y_{M-1}) + d(\lambda, y_M)\}, \\ & \{D(x_1 \dots x_{N-1}, \quad y_1 \dots y_M) + d(x_N, \lambda)\}]. \quad (1) \end{aligned}$$

Recognition using distance criteria is obtained by essentially evaluating the string in the dictionary which is "closest" to the noisy one as per the metric under consideration.

Rather than compute the individual string edit distance separately, Kashyap *et.al.* in [10], developed a recursive procedure to compute $D(X, Y)$ for all the relevant prefixes in the entire dictionary. This involved only a fixed finite number of the prefixes of X and the left derivative of Y. They introduced a new distance measure, $D_1(X, Y)$, (an intermediate computational tool called a *pseudodistance* because it assumes that the last symbol of X was not inserted during

editing) between X and Y. The measure $D_1(X, Y)$ has the desirable properties that it can be computed "recursively" and that the final distance $D(X, Y)$ can be obtained from *it* using only a single additional symbol comparison. The relationship between $D_1(X, Y)$ and $D(X, Y)$ is formalized below (see [10] and [14] for the details):

$D(X, Y) = \min[D_1(X, Y), \{D_1(X_p, Y) + d(x_N, \lambda)\}]$.

Similarly, the recursive properties of the pseudodistances between an arbitrary string $X \in A^*$ and $Y^{(K)}$ can be summarized as follows (see [10] and [14]).

If $X = X_1bc$ (where $|X| \geq 2$) with $|X_1| \geq 0$, since c is not inserted:

$$\begin{aligned} D_1(X_1bc, Y^{(K+1)}) = \min\ [\ & \{D_1(X_1bc, Y^{(K)}) + d(\lambda, y_{K+1})\}, \\ & \{D_1(X_1b, Y^{(K)}) + d(c, y_{K+1})\}, \\ & \{D_1(X_1, Y^{(K)}) + d(b, \lambda) + d(c, Y^{(K+1)})\}]. \end{aligned} \quad (2)$$

Observe that in the above expressions the number of terms included to obtain the distance between X_1bc and $Y^{(K)}$ is merely three, and is superior to the expression valid for finite-state representations for Regular Languages [22]. The reasons for this are explained in detail in [10] and [14].

3 Procedure and Data Structure for Obtaining X^+

In [10], Kashyap *et al.* showed that the pseudodistances $D_1(X, Y^{(K)})$ can be recursively computed using the dynamic programming principle given by Equation (2). For this purpose, they defined two sets $R^{(K)}$ and $S^{(K)}$, where $R^{(K)}$ is the set of prefixes of H into which $Y^{(K)}$ can be transformed with finite pseudodistances, and $S^{(K)}$ associates every element in $R^{(K)}$ with its corresponding pseudodistance with $Y^{(K)}$, for all $K = 1, ..., M$ where $M = |Y|$. The problem with this method of calculation (i.e., using these two sets) , is that the computation is so complicated and is not feasibly implemented. It also requires extensive *set-based computations*. The details of the conceptual representation of the solution of [10], its FSM model, and how it differs from other FSM models given in the literature, are included in [14]. We shall, however, use the dynamic programming recursive relation explained above, and enhance it using a new structure called the Linked List of Prefixes (LLP), which makes the computations feasible.

A *trie* is an alternative to a BST for storing strings in a sorted order [7]. Tries are both an abstract structure and a data structure that can be superimposed on a set of strings over some fixed alphabet [5]. As an abstract structure, they are based on a splitting scheme, which, in turn, is based on how the letters are encountered in the strings. In any specific implementation, the nodes of the trie can be modelled and represented in different ways. They can be implemented using an array [5], [9], a linked list, or even a binary search tree [1]. Fig. 1 (left side) shows an example of a simple dictionary represented as a trie. Observe the relationships between the prefixes of a string and its ancestors in the trie.

We now explain a feasible way of obtaining X^+ when the dictionary is represented using tries. To calculate the best estimate X^+, what we need is to divide

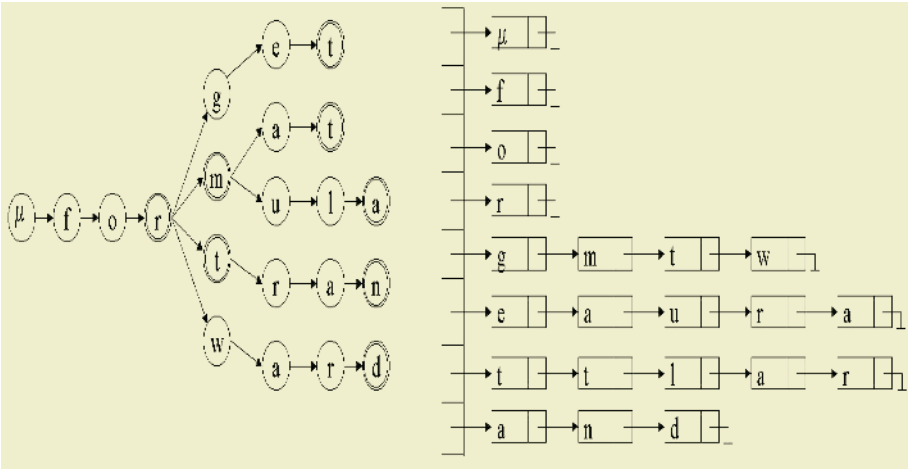

Fig. 1. An example of a dictionary stored as a trie and the corresponding LLP with the words {for, form, fort, fortran, formula, format, forward, forget}.

the dictionary into its sets of prefixes. Each set $H^{(p)}$ is the set of all the prefixes of H of length less than or equal to p, for $1 \leq p \leq N_m$, where N_m is the length of longest word in H. The trie itself divides the prefixes and the dictionary in the way we want, as each sub-trie starting from the root to level p corresponds to all the prefixes in the set $H^{(p)}$. What we need is a data structure that facilitates the trie traversal, and gives us a unique data structure that can always be used to effectively compute the pseudodistances for the prefixes. We called this data structure the Linked Lists of Prefixes (LLP).

The LLP consists of a linked list of levels, *where each level is a level in the corresponding trie*. Each level, in turn, consists of a linked list of all prefixes that have the same length p. The levels are ordered in an increasing order of the length of the prefixes, exactly as in the case of the trie levels. The figure on the right side of Fig. 1 shows the corresponding LLP for the trie shown on the left side of Fig. 1. The character written in each node is actually a pointer to the node of the trie itself, and so we can access the parent nodes in the trie in a straightforward manner, as will be seen in the algorithm presently. The values of D_1 used during calculations is stored in the trie nodes. Thus, actually the LLP is a data structure used to facilitate the traversing of the trie in the proposed string correction algorithm.

Finally, we need to store a linked list of pointers to all the nodes in the trie which corresponds to the words in the dictionary. This list is called"dictionary-words". The pseudo-code for constructing the LLP from a trie and the correctness of the algorithm are given in [14].

3.1 The Procedure for Obtaining X^+

Let U be a prefix corresponding to some node in the trie T. It can be seen from (2) that if the pseudodistance between a certain node U and $Y^{(K+1)}$ has to be

computed, it can be done with merely the knowledge of the pseudodistances between each of U, U_p (the parent of U), Ug (the grandparent of U), and the string $Y^{(K)}$ respectively. In each node of the trie we store two values for the pseudodistances, namely the previous value of D_1 and its new value, new_D_1, (both of which are initialized to ∞ for every node in the trie, or rather, in the conceptual LLP). Observe that we need both of these quantities because the computation of D_1 will require the "old" values of D_1 calculated in the previous iteration, which we refer to as $D_1(U, Y^{K-1})$. We then proceed along the trie from the root towards the leaves, and at each iteration, fill in the distances for nodes at the same level and for nodes which are two levels deeper than at the previous iteration. Indeed, it is at this stage that the LLP becomes useful. Rather than work with *set based operations* as in [10], the LLP permits the access to nodes *level by level*, while details of the parents and grandparents are gleaned from the trie itself. The pseudo-code for the computation that formalizes this along the trie and LLP and are omitted here. They can be found in [14].

4 Experimental Results

To investigate the power of our new method with respect to computation various experiments were conducted. The results obtained were remarkable with respect to the gain in time and the number of computations. The new method was compared with Algorithm_GLD, a PR scheme which used any traditional editing [10], [11], [12], [18], [19], [23] algorithm using symbol-dependent costs as described in Section 2, where the costs were assigned for elementary distances using the GLD, and the inter-string distances were computed sequentialy.

The Dictionary consisted of 342 words obtained as a subset of the most common English words [6] augmented with words used in computer literature[1]. The length of the words was greater than or equal 7 and the average length of a word was approximately 8.3 characters. Other experiments (see [14]) were also done for larger subsets of the most common English words [6].

From these 342 words, five sets, SA, SB, SC, SD, and SE, of 1368 words each were generated using the noise generator model described in [15]. We assumed that the number of insertions was geometrically distributed with parameter $\beta = 0.7$. The conditional probability of inserting any character $a \in A$ given that an insertion occurred was assigned the value $1/26$; and the probability of deletion was set to be $1/20$. The table of probabilities for substitution (typically called the confusion matrix) was based on the proximity of character keys on the standard QWERTY keyboard and is given in [15][2]. The statistics associated with each of the five sets are given in Table 1. Some of the words in the dictionary are very similar even before garbling such as "official" and "officials"; "attention", "station" and "situation". These are words whose noisy versions can themselves easily be mis-recognized. The errors per word associated with these five sets was bounded by 30%, 40%, 50%, 60% and 100% respectively.

[1] This file is available at `www.scs.carleton.ca/~oommen/papers/WordWldn.txt`

[2] It can be downloaded from `www.scs.carleton.ca/~oommen/papers/QWERTY.doc`

Table 1. Noise statistics of the set SA, SB, SC, SD, and SE.

Errors	SA	SB	SC	SD	SE
Number of insertions	873	1105	1545	1766	2763
Number of deletions	461	503	549	567	633
Number of substitutions	808	931	1028	1051	1120
Total number of errors	2142	2539	3122	3384	4516
Average % error	18.91	22.41	27.26	29.87	39.87
Maximum % error per word	30.00	40.00	50.00	60.00	100.00

Table 2. The experimental results obtained from each of the three sets for the 1368 noisy words. The results are given in terms of the number of operations needed, the time and accuracy. The results also shows the percentage of savings in the total number of operations and the time used when utilizing the LLP-based method as opposed to the sequential method using Algorithm-GLD.

Operation	SA		SC		SE	
	GLD	LLP	GLD	LLP	GLD	LLP
Add	430529184	69466624	451174752	73954200	491263920	82668160
Min	132990720	18770224	139606272	19892118	152452224	22070608
Oper.	563519904	88236848	590781024	93846318	643716144	104738768
Savings	84.34		84.11		83.72	
Time (sec.)	19	8	19	8	21	10
Savings	57.89		57.89		52.38	
Accuracy	98.83	98.54	97.59	97.37	96.05	95.76

The two algorithms, Algorithm_GLD (the algorithm which sequentially computed the GLD for the entire dictionary) and our algorithm, Algorithm_LLP, were tested with the five sets of noisy words. We report the results obtained in terms of the number of computations (additions and minimizations), the time, and the accuracy for only the three sets SA, SC, SE, in Table 2. The results shows the significant benefits of the LLP-based method with respect to the time and number of computations. For example, for the set SA, the number of operations is 563,519,904 for Algorithm_GLD, and 88,236,848 for the LLP-based method with a saving of 84.34%. The time taken (on a Pentium II processor, 1000 GHZ) is 19 seconds for Algorithm_GLD and just 8 seconds for the LLP-method, which is a saving of 57.89%.

The savings in the computations are more than 80% for all sets which is interesting, and the saving in time is more than 50%. The accuracy is very slightly less than what can be obtained from the sequential computation, because some of the words which contained two successive deletions, were not correctly recognized by the LLP method. Indeed if these words were removed from the test data, the accuracy will be the same for both schemes. For example, for the set SE the test results shows that both schemes give the same recognition except for one string in the LLP method, namely for the word "property". The noisy

word in this case was "opertzg" in which the first two symbols were successively deleted. X^*, which generated Y, was estimated as the word "experts" by the LLP method.

5 Conclusion and Future Work

In this paper we have presented a feasible solution for the problem of estimating a transmitted string X^* by processing the corresponding string Y, which is a noisy version of X^*, an element of a finite (but possibly, large) dictionary H, when the whole dictionary is considered simultaneously. The method builds on the concepts introduced by Kashyap *et al.* [10], where the set model used in the computations was not feasible. We enhanced this by the introduction of a new data structure called the Linked Lists of Prefixes (LLP), which can be constructed when the dictionary is represented using a trie. The LLP is an enhanced, but modified, representation of the trie, which can be used to facilitate the "dictionary-based" dynamic programming calculations. The LLP-based algorithm for the syntactic PR of strings has been rigorously tested. The results showed significant benefits (with respect to the time and number of computations) when compared with Algorithm_GLD, the algorithm which sequentially computes the GLD for the entire dictionary.

As a future work we would like to extend this method for probabilistic computations and "two-sided tries".

References

1. J. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. *Eighth Annual ACM-SIAM Symposium on Discrete Algorithms New Orleans*, January 1997.
2. P. Bucher and K. Hoffmann. A sequence similarity search algorithm based on a probabilistic interpretation of an alignment scoring system. In *Proceedings of the Fourth International Conference on Intelligent Systems for Molecular Biology, ISMB-*, volume 96, pages 44–51, 1996.
3. H. Bunke. Structural and syntactic pattern recognition. *in Handbook of Pattern Recognition and Computer Vision. Edited by C.H.Chen, L.F.Pau and P.S.P. Wang, World Scientific, Singapore*, 1993.
4. H. Bunke and J. Csirik. Parametric string edit distance and its application to pattern recognition. *IEEE Trans. Systems, Man and Cybern, SMC-*, 25:202–206, 1993.
5. J. Clement, P. Flajolet, and B. Vallee. The analysis of hybrid trie structures. *Proc. Annual A CM-SIAM Symp. on Discrete Algorithms, San Francisco, California*, pages 531–539, 1998.
6. G. Dewey. *Relative Frequency of English Speech Sounds.* Harvard Univ. Press, 1923.
7. S. Heinz, J. Zobel, and H. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20(2):192–223, 2002.
8. J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Comm. Assoc. Comput. Mach.*, 20:350–353, 1977.

9. P. Jacquet and W. Szpankowski. Analysis of digital tries with markovian dependency. *IEEE Trans. Information Theory, IT-*, 37(5):1470–1475, 1991.
10. R. L. Kashyap and B. J. Oommen. An effective algorithm for string correction using generalized edit distances -i. description of the algorithm and its optimality. *Inf. Sci.*, 23(2):123–142, 1981.
11. A. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Phys. Dokl.*, 10:707–710, 1966.
12. W. J. Masek and M. S. Paterson. A faster algorithm computing string edit distances. *J. Comput. System Sci.*, 20:18–31, 1980.
13. B. J. Oommen. Recognition of noisy subsequences using constrained edit distances. *IEEE Trans. on Pattern Anal. and Mach. Intel.,PAMI-*, 9:676–685, 1987.
14. B. J. Oommen and G. Badr. Dictionary-based syntactic pattern recognition using tries. Unabridged version of the present paper. Can be made available by contacting the authors.
15. B. J. Oommen and R. L. Kashyap. A formal theory for optimal and information theoretic syntactic pattern recognition. 31:1159–1177, 1998.
16. B. J. Oommen and R. K. S. Loke. Syntactic pattern recognition involving traditional and generalized transposition errors: Attaining the information theoretic bound. Submitted for Pubication.
17. B. J. Oommen and R. K. S. Loke. Designing syntactic pattern classifiers using vector quantization and parametric string editing. *IEEE Transactions on Systems, Man and Cybernetics SMC-*, 29:881–888, 1999.
18. J. L. Peterson. Computer programs for detecting and correcting spelling errors. *Comm. Assoc. Comput. Mach.*, 23:676–687, 1980.
19. D. Sankoff and J. B. Kruskal. *Time Warps, String Edits and Macromolecules: The Theory and practice of Sequence Comparison.* Addison-Wesley, 1983.
20. G. A. Stephen. *String Searching Algorithms*, volume 6. Lecture Notes Series on Computing, World Scientific, Sihgapore, NJ, 2000.
21. E. Ukkonen. Algorithm for approximate string matching. *Information and control*, 64:100–118, 1985.
22. R. A. Wagner. Order-n correction for regular languages. *Comm. ACM*, 17:265–268, 1974.
23. R. A. Wagner and M. J. Fisher. The string to string correction problem. *J. Assoc. Comput. Mach.*, 21:168–173, 1974.