Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2007)

A Dynamic Programming Algorithm for Name Matching

Philip Top
School of Electrical and Computer Engineering
Purdue University
West Lafayette, Indiana 47906
Email: ptop@purdue.edu

Farid Dowla
Lawrence Livermore National Lab
Livermore, California 94551
Email: dowla1@llnl.gov

Jim Gansemer
Lawrence Livermore National Lab
Livermore, California 94551
Email: gansemer1@llnl.gov

***Abstract*— In many database and data mining applications concerning people, name matching plays a key role. Many algorithms to match names have been proposed. These algorithms must take into account spelling and transcription errors, name abbreviations, nicknames, out of order names, and missing or extra names. The existing algorithms typically fall along the lines of sound based, edit distance based, or token based algorithms which can use other methods in matching each part of the name separately. In this article, we propose a dynamic programming approach that includes a substring matching algorithm. The algorithm's performance is compared against two often used algorithms by testing on a random sample of names from a database. The data used for the testing comes from the DHS US-VISIT Arrival and Departure Information System database, which includes names from all over the world. The performance on this data set was compared with the that of the Damerau-Levenshtein Algorithm and the Jaro-Winkler algorithm. The dynamic programming algorithm with substring matching performed better than both of these algorithms on the data tested.**

I. Introduction

There are many problems, particularly in database searching and matching, where it is necessary for a computer to compare the names or information of two people and make a decision on whether or not the two names represent the same person. The performance of these algorithms is measured by their capability to distinguish between names originating from one person and names originating from distinct people. Many algorithms for accomplishing this task have been proposed over the last several decades. The methodology behind these algorithms typically falls into three categories, though many hybrids have emerged.

The first category is based on sounds. Typically, the name or word is compressed to a sound code using a list of rules. These algorithms often employ the phonetic structure of a language in addition to standard sounds to match names that sound the same. Because of this methodology they are sometimes difficult to use when comparing names from different cultures. The most common method in this category is the soundex algorithm and its variants.

Another class of algorithms is based on an edit distance between two strings. The edits used to match two names can include inclusions, deletions, substitutions, and transpositions or character reversals. Each of these edits is assigned a cost and the final score is based on the minimum number of the edits required to convert one string into another. Determining the minimum number of edits is often done using a dynamic programming approach to compute a matching matrix. Two common algorithms of this type are the Damerau algorithm[1] and the Monge-Elkan distance function [2]. The Monge-Elkan distance function is based on the Gotoh-Smith-Waterman algorithm originally developed for finding matching substrings in DNA sequences [3]. This algorithm uses positive scores for matched characters and penalties for mismatches and gaps. The contribution of Gotoh was to include affine gap costs which differentiate penalties for starting and continuing a gap [4]. The scores are then scaled to between 0 and 1. The Damerau algorithm counts the number of subtractions, deletions, and substitutions required to match strings. Levenshtein extended the Damerau algorithm to include transpositions [5].

A commonly used algorithm that is not based on a dynamic programming matrix is the Jaro algorithm [6]. The Jaro algorithm counts the number of common characters in two strings if they are within half the length of the shortest string of its position in the other string. The algorithm records the number of these common characters which occur in order and those that occur out of order. Winkler extended this algorithm to give extra weight to common prefixes [7]. The Jaro-Winkler works better the shorter the strings it is matching. It therefore works well on names, which are typically short strings.

A third class of algorithms split strings into tokens or words, in the case of personal names the split is made wherever a space or dash occurs. Once each token is separated, it is compared to the tokens in the other string to find the best set of matches for all the tokens. The final score is then based on some function of the individual matches and possibly the order in which these matches occur. Algorithms of this type include the Jaccard similarity and the TFIDF [8]. Often a distance metric of some kind is used for individual token

Thanks to the Department of Homeland Security and the US-VISIT Program Management Office for funding this research.

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48

UCRL-CONF-225678

matches along with one of the token based algorithms in a hybrid scheme. A comparison of some of the algorithms discussed thus far is available in [9].

II. Algorithm Description

The algorithm proposed in this article incorporates some of the ideas proposed in previous algorithms. The new algorithm is based on dynamic programming approach using the Smith-Waterman algorithm with a modification to account for transpositions. The scoring is done on the backtracking portion of the dynamic programming algorithm. Each token in a string is then matched with its best matching substring in the other string. A score is computed for each token in a string, shown as W_i. The score for each string is computed as a normalized sum of the scores for the individual tokens,

$$S_n = \frac{1}{WC}\sum_{i=1}^{N_n} W_i, \tag{1}$$

where WC is the normalization constant. An overall matching score is computed as the maximum value in the matching matrix divided by the average string length,

$$S_w = \frac{2 \cdot max(DV)}{length(n1) + length(n2)}. \tag{2}$$

The strings are denoted by $n1$ and $n2$. The final match score is a weighted sum of the individual string scores and an overall match scores;

$$S = w_1 \cdot S_1 + w_2 \cdot S_2 + w_w \cdot S_w, \tag{3}$$

with

$$w_1 + w_2 + w_w = 1. \tag{4}$$

Included in the algorithm are penalties for some of the possible errors. The errors accounted for are mismatches, gaps, and transpositions. The penalty for these errors are denoted by M, G, and T respectively. The matching score for each possible character comparison is specified in a character match matrix, CM. The simplest example of this is

$$CM_{(i,j)} = \begin{cases} 1, & \text{if } i = j, \\ -M, & \text{if } i \neq j. \end{cases} \tag{5}$$

This example results in a score of 1 for matching characters and a penalty of M for mismatches. Depending on the specific situation other more complicated character matching may be appropriate.

If L_1 and L_2 are defined as the lengths of strings $n1$ and $n2$ respectively, The dynamic programming matrix DV has size $L_1 + 1$ by $L_2 + 1$. The character strings are indexed by subscripts; $n1_3$ refers to the third character of string $n1$. The following pseudo-code assumes indexing starts at 0 to alleviate the need to deal with the edges seperately on the backtracking portion of the algorithm. The algorithm for computing the dynamic programming DV is as follows:

for $i = 1$ to L_1
  for $j = 1$ to L_2
    $DV_{(i,j)} = max(DV_{(i-1,j-1)} + CM_{(n1_{j-1},n2_{j-1})}$
      $, 0, DV_{(i-1,j)} - G, DV_{(i,j-1)} - G)$
    if $DV_{(i-1,j)} > DV_{(i-1,j-1)}$
      and $DV_{(i,j-1)} > DV_{(i-1,j-1)}$
      $DV_{(i-1,j-1)} = max(DV_{(i-1,j)}, DV_{(i,j-1)})$
      $DV_{(i,j)} = max(DV_{(i-1,j-1)} - T, DV_{(i,j)})$
    end if
  end for
end for.

The scores for the individual tokens, denoted as W_i, are computed by backtracking through the matrix. An increase or decrease in the score is determined by adding or subtracting scores dependent on the match of the characters in the path as specified by CM. The score is divided by the number of steps taken to get from the end of the token to the beginning. If extra characters are present in the corresponding portion of the other string, the number of steps can be higher than the actual number of characters in the token. At minimum, the number of steps will be equal to the length of the token. As a means of decreasing the impact of errors near the ends of the tokens, the score is never allowed to go negative. The result for each token is a score between 0 and 1.

The starting point for each of the token is determined by searching for local maxima in the appropriate row or column of matrix DV. If multiple local maxima are present, the maxima with the greatest difference in score over the adjacent elements is chosen. The matching then proceeds along the highest scoring path until it reaches the end of the token. The highest scoring path is chosen first by sequentially matching the characters. If the next characters in sequence do not match, then the match that has the maximum score according the the DV matrix is selected. This selection could involve inserting a space in either of the strings or matching mismatched character.

As each token in the string is matched, the region searched for the local maxima is reduced as the match is not allowed to occur in previously matched regions. This technique ensures the best overall match and penalizes repeated tokens if they are not present in both strings being compared. The local maxima are determined using adjacent elements of the column or row being scanned, if two adjacent elements are not available, for instance on the ends of rows or columns, all available adjacent elements along the row or column are used. If a maximum is discovered at the endpoint of a row or column, it is automatically chosen. This automatic selection was put in place because a great majority of names match in order, and this selection speeds up the matching process. A local maxima is defined as an element that is strictly greater then the adjacent elements, so an element which is equal to any of the adjacent elements being tested is not considered a local maximum. If no local maxima are located, scoring proceeds from the previous point to the end of the next token. A limit was put in for a minimum score at the local maxima.

		R	O	B		A	L	T	O	N
	0	0	0	0	0	0	0	0	0	0
R	0	1	0.6	0.2	0	0	0	0	0	0
O	0	0.6	2	1.6	1.2	0.8	0.4	1	1	0.6
B	0	0.2	1.6	3	2.6	2.2	1.8	1.4	1	0.6
E	0	0.8	1.2	2.6	2.6	2.2	1.8	1.4	1	0.6
R	0	1	0.8	2.2	2.2	2.2	1.8	1.4	1	0.6
T	0	0.6	0.6	1.8	1.8	1.8	2.4	2.8	2.4	2
	0	0.2	0.2	1.4	2.8	2.4	3	2.4	2.4	2
A	0	0	0	1	2.4	3.8	3.4	3	2.6	2.2
L	0	0	0	0.6	2.0	3.4	4.8	4.4	4	3.6
T	0	0	0.6	1.2	1.6	3.0	4.4	5.8	5.4	5
O	0	0	1	0.6	1.2	2.6	4.0	5.4	6.8	6.4
N	0	0	0.6	0.6	0.8	2.2	3.6	5.0	6.4	7.8

TABLE I

DYNAMIC PROGRAMMING MATRIX FOR EXAMPLE 1

The limit prevents very short character sequence matches in low matching regions from triggering the maxima detector.

The normalization constant, WC, is initially set as the actual number of tokens in a string, however, to give less emphasis to shorter strings the constant was modified by the following algorithm:

if tokenlength = 3
    $W_i = \frac{1}{2} \cdot W_i$
    $WC = WC - \frac{1}{2}$
elseif tokenlength = 2
    $W_i = \frac{1}{3} \cdot W_i$
    $WC = WC - \frac{2}{3}$
elseif tokenlength = 1
    $W_i = \frac{1}{4} \cdot W_i$
    $WC = WC - \frac{3}{4}$
end if.

Both W_i and WC are modified to preserve a maximum score of 1.0 for an exact match. Less emphasis is given to shorter strings due to the higher liklihood they will match with a random section of a non-matching string. Therefore, shorter strings have less discrimination power and are, hence, given less weight in the final matching score.

III. EXAMPLE

An example will further illustrate the algorithm in action. The names used in the example are purely fictional, created for the purposes of the example. To illustrate how the algorithm matches shortened names, we will compare ”Rob Alton” as Name 1 and ”Robert Alton” as Name 2. The penalties M, T, and G were all set to be 0.4. The weights used were $w_1 = w_2 = 0.35$ and $w_w = 0.3$. The matrix DV is computed for this example in Tab. I

Starting with the final character of Name 1, we search along the last column for local maxima. A local maximum is located in the lower right corner. So, backtracking starts in the corner and progresses along the highlighted path until it reaches the beginning of the next token, which corresponds to the ”B” column and ”T” row. The algorithm then searches for and locates a local maxima in the remaining segment of Name 2. The local maximum is located at the ”B” in ”ROBERT”, hence backtracking starts from that point and proceeds to match every character in ”ROB” generating a score of 3.0 in 3 steps. As ”ROB” is 3 characters or less its weight is reduced to 50% of its original value, and the token count, WC, is also reduced by 0.5 to a value of 1.5. For Name 1, the score is

$$S_1 = \frac{1}{1.5} \cdot \left(\frac{1}{2} \cdot \frac{3}{3} + \frac{6}{6} \right) = 1.0. \tag{6}$$

In the same way backtracking through Name 2 starts in the lower right corner and tracks until it gets to the B in ”ROB” of Name 1. For scoring of second token in Name 2, a score of 6.0 was obtained in 6 steps. For the first token, no local maxima is discovered along the ”T” row in the remaining characters of Name 1, so backtracking continues where it left off. The backtracking continues along the ”B” column until it gets the ”B” in ”ROBERT” from Name 2, then the algorithm matches all the remaining characters. The backtracking in this case produces a score of 3.0 in 6 steps. The missing characters in the first token would ordinarily produce a penalty and reduce the score, however, since they occur at the end of a token, and the score is never allowed to go negative, the penalty is reduced. Thus, in total for Name 2 we have

$$S_2 = \frac{1}{2} \cdot \left(\frac{3}{6} + \frac{6}{6} \right) = 0.75. \tag{7}$$

The maximum from the matching matrix is 7.8, so the string whole matching score is then

$$S_w = \frac{2.0}{9 + 12} \cdot 7.8 = 0.742. \tag{8}$$

Finally, we can compute the total score

$$S = 0.35 \cdot 1.0 + 0.35 \cdot 0.75 + 0.3 \cdot 0.742 = 0.835. \tag{9}$$

We can compare this score with a score of 0.73, which would be obtained if a simple edit distance algorithm was used. The matching would require 3 edits in 11 characters. While the score is not directly comparable, a higher score provides a better possibility of discrimination. The value of a matching algorithm lies in its ability to distinguish matching names from non-matching names.

IV. TEST SETUP

The proposed algorithm was compared with two other commonly used algorithms. The two algorithms tested in addition to the new algorithm were the Jaro-Winkler algorithm and a token based algorithm using the Damerau-Levenshtein algorithm. The records used in the algorithm come as first and last names, though each may contain multiple tokens. In the Jaro-Winkler algorithm these names are compared separately and each comparison is given a weighting of 0.5. The Damerau algorithm splits the names into tokens, furthermore, it also attempts to find and split run-on names

where spaces have been left out in one of the names but not the other. An additional component of the final score in the Damerau algorithm is computed using the order of the matching tokens. The dynamic programming algorithm uses penalties set at 0.4 and weights set at 0.35 for each name and 0.3 for the overall score. These parameters are the same parameters as were used in the example.

The data set used in the testing was a random sample from the Department of Homeland Security's US-VISIT Arival and Departure Information System (ADIS) database. The sample contained all entries for a specific period of time, and included names from all over the world. From the data sample available we computed probability distributions of the tested algorithms for matching and non-matching names. The matching names set was extracted from the availabe sample of the database. All the samples were matched through other means and were all instances of persons in which the names in the record did not match exactly, either by errors or differences in recording the name. A majority of the matches in the database had names which matched exactly. However, since detecting matches when the names match exactly is trivial these matches were excluded. In total, roughly 64000 record pairs were scored using all three algorithms. The scores from 0 to 1 were placed into bins with a resolution of 0.01. Once all the data was recorded, the results were normalized to approximate a probability distribution.

To test mismatched names, 10000 names were selected at random from the database, all possible comparisons were made from these 10000 names resulting in nearly 50 million individual comparisons. A probability distribution was produced in the same manner as the matching names for all the algorithms tested.

The algorithms were all written and tested in Perl to simplify the database access. Additional versions of the algorithms were also written in C for testing and comparison purposes. To simplify the process, all the name comparisons were made using only uppercase letters. The strings were converted to numerical arrays based on ascii codes. The characters A-Z were converted to the numbers 0-25 and the space and other characters were converted to the code 26. This conversion allowed rapid indexing into the character matching matrix and easy comparison.

V. Results

The results of the tests indicate that the dynamic programming algorithm performs better than both other algorithms tested at higher probability of match levels, and performs equivalently to the Jaro-Winkler algorithm at low probability of false positive levels. The distributions of matching and non-matching comparisons for the dynamic programming algorithm are shown in Fig. 1. The distributions are plotted on a log scale on the y-axis for visibility purposes. From the distributions, receiver operating characteristic (ROC) curves can be plotted and compared for the three name matching algorithms. The ROC curve is a plot of the probability of false positive matches versus the probability of detecting

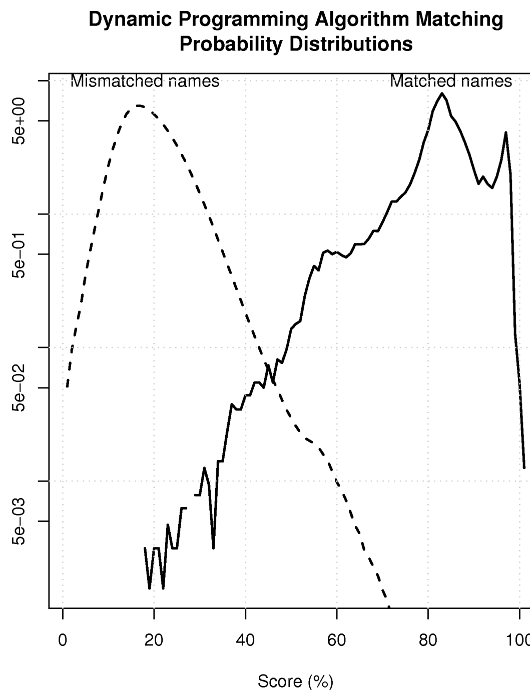

Fig. 1. Dynamic Programming algorithm probability distributions

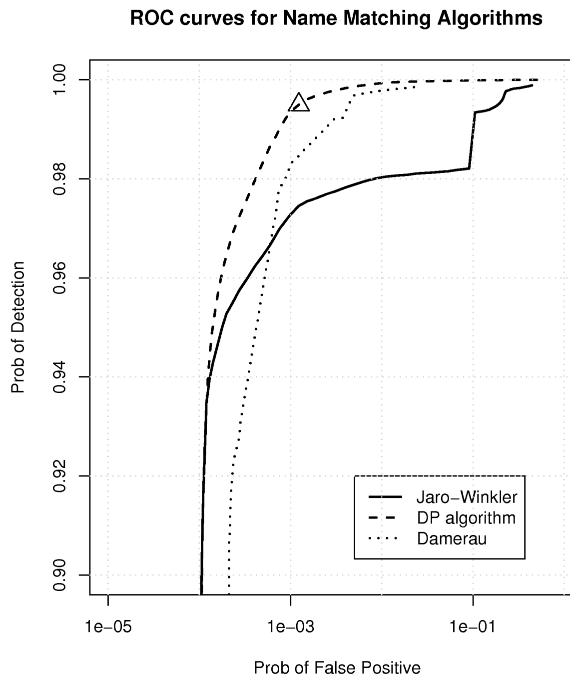

Fig. 2. ROC curves for name matching algorithms

correct matches. The ROC curves for the three algorithms tested are shown in Fig. 2. From the plot, it is clear that the dynamic programming algorithm performs better than both the other two algorithms and also shows the different circumstances when the other two algorithms perform better than the other. The triangle on the ROC curve for the dynamic programming algorithm marks the performance of a threshold of 0.5. The ROC curve is generated under the assumption that the probability of matching and non-matching names is equal. This assumption is likely not valid in many situations, which means that the specific probabilities on the ROC curve are not valid, but the comparison of the different algorithms is still valid.

VI. Discussion

The algorithm contains several parameters that would allow it to be customized to a particular data set. The use of the comparison matrix, CM, allows different character comparisons to be given different weights. In the data set used for testing, the probability of mistaking one character for another was small enough that it was of no advantage to change the penalty for any of the errors, so this was not done. Another possibility, is changing the positive score for the matching characters, giving more weight to less common characters. This modification would make normalization more difficult, but may improve matching performance. Further adjustments to the weighting or penalties may also improve matching performance in other applications. A series of tests were run to determine the best weights and penalties for this data set, though the tests showed the actual performance was not sensitive to changes in the parameters.

The speed of the dynamic programming algorithm is slower than the other algorithms tested, the Jaro-Winkler algorithm is significantly faster than either of the other two. Both the Damerau-Levenshtein and the dynamic programming algorithm have the same complexity in computing the matrix; both are $O(mn)$ where m and n are the lengths of the strings being compared. However, the dynamic programming algorithm has a much more complicated backtracking procedure. This extra computation makes it noticeably slower than the Damerau-Levenstein algorithm, as much as a factor of 3 for short strings. In practice, the actual increase in time of the algorithm will depend on the language the algorithm is implemented in and the size of strings being compared. When tested on a $2\,$GHz computer, the Jaro-Winkler algorithm ran 100000 comparisons with an average length of 12 characters in $1.7\,$seconds. The same test took 27.1 seconds, and 1 minute 25 seconds for the Damerau-Levenshtein and the dynamic programming algorithms, repectively. With longer strings, these times increase significantly due the computation of dynamic programming matrix. Various means could be used to speed up the algorithm such as implementing a short circuit that would break off computation for very low scoring strings, or some fast algorithm for filling the dynamic programming matrix.

Ideally, the algorithm would be used in situations where some other analysis can be used to screen out all but a few candidates. In the case of pure name matching problem, an ideal presreen would be a fast comparison algorithm with a low false negative rate. This fast algorithm would remove a majority of the possibilities and leave the final matching to the dynamic programming algorithm. In database matching situations, there is typically additional information that can be used to narrow down the number of names that could be considered to a mere handful, in which case the algorithm proposed here can be used to pick from the remaining matches or reject all of them.

The dyanmic programming algorithm is slower than other algorithms, but in cases where some speed can be traded for improved matching performance this algorithm would be good fit. The algorithm is designed for names but it could be used for matching any string, though computation time becomes a key issue for longer strings. The algorithm should work on things such as addresses, place names, and company names. It is particulary effective in situations where abbreviations, inconsistent token seperations, and non-uniform ordering are common.

VII. Conclusion

In this article we have described a new algorithm for matching strings. The algorithm uses dynamic programming methods to create a matching algorithm that combines features of several other algorithms into a single method. Tests indicate that the algorithm shows improved matching performance over two commonly used algorithms. The improvement comes at the expense of increased computational time in a backtracking routine. Adjustable parameters in the algorithm will allow it to be used in many circumstances for improved matching performance.

References

[1] F. J. Damerau, “A technique for computer detection and correction of spelling errors,” *Communications of the ACM*, 1964.

[2] A. Monge and C. Elkan, “The field matching problem: Algorithm and applications,” in *Proceedings of the 2nd ACM SIGKDD International Conference on Discovery and Data Mining*, pp. 267–270, AAAI Press, 1997.

[3] T. F. Smith and M. S. Waterman, “Identification of common molecular subsequences,” *Journal of Molecular Biology*, vol. 147, pp. 195–197, 1981.

[4] O. Gotoh, “An improved algorithm for matching biological sequences,” *Journal of Molecular Biology*, vol. 162, pp. 705–708.

[5] V. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” *Soviet Physics Doklady*, 1966.

[6] M. A. Jaro, “Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida,” *Journal of American Statistical Association*, vol. 84, no. 406, pp. 414–420, 1989.

[7] W. E. Winkler, “The state of record linkage and current research problems,” Publication R99/04, Statistics of Income Division, Internal Revenue Service, 1999.
Available from http://www.census.gov/srd/www/byname.html.

[8] W. W. Cohen, “Data integration using similarity joins and a word-based information representation language,” *ACM Transactions on Information Systems*, vol. 18, no. 3, pp. 288–321.

[9] W. W. Cohen, P. RaviKumar, and S. E. Fienberg, “A comparison of string distance metrics for name-matching tasks,” in *Proceedings of the IJCAI*, 2003.