Computational Linguistics and Chinese Language Processing
Vol. 19, No. 1, March 2014, pp. 33-54

Linking Databases using Matched Arabic Names

Tarek El-Shishtawy*

Abstract

In this paper, a new hybrid algorithm that combines both token-based and character-based approaches is presented. The basic Levenshtein approach also has been extended to the token-based distance metric. The distance metric is enhanced to set the proper granularity level behavior of the algorithm. It smoothly maps a threshold of misspelling differences at the character level and the importance of token level errors in terms of token position and frequency.

Using a large Arabic dataset, the experimental results show that the proposed algorithm successfully overcomes many types of errors, such as typographical errors, omission or insertion of middle name components, omission of non-significant popular name, and different writing style character variations. When compared with other classical algorithms, using the same dataset, the proposed algorithm was found to increase the minimum success level of the best tested lower limit algorithm (Soft TFIDF) from 69% to about 80%, while achieving an upper accuracy level of 99.67%.

Keywords: Name Matching, Record Linkage, Data Integration, Arabic NLP, Information Retrieval.

1. Introduction

Information about individuals can be found in a variety of resources, such as population survey databases, national identifier databases, medical records, news articles, tax information, and educational databases. In all of these heterogeneous sources, name matching is a fundamental task for data integration that joins one or more tables to extract information referring to the same person.

Matching personal names has been used in a wide range of applications, such as record linkage or integration, database hardening, removing or cleaning up duplicated records, and searching the Web. Unfortunately, the name may not be known exactly, may be misspelled, or may have spelling variations. Therefore, in these applications, the general word matching

* Prof. Ass. of Computer Engineering, Faculty of Computers and Information, Benha University
E-mail: t.shishtawy@ictp.edu.eg

techniques are insufficient, and optimized techniques have been developed to cope with matching multiple variations of the same personal name.

There are efficient and well-established algorithms that deal with spelling errors, variants for strings, and name matching at a character level. For relatively short names that contain similar yet orthographically distinct tokens, character-based measures are preferable because they can estimate the difference between the strings with higher resolution (Bilenko *et al*., 2003a; Bilenko & Mooney, 2003). In languages where names have very close typographic structure, such as Arabic, character level similarity is not enough to produce high precision matching.

Unfortunately, spelling errors are not the only source of name mismatching. People may report their names inconsistently by removing or inserting additional name tokens, adding initial titles, or writing different punctuation marks and whitespaces. In all of these cases, bag-of-words methods are better suited to the matching problem, since they are more flexible at word level. In addition, token-based approaches are not able to capture the degree of similarity between individual tokens with minor variations in characters (Bilenko *et al*., 2003b).

Experimental results show that hybrid techniques, which take word frequency as well as character-based word similarities into account, increase matching. A first attempt in this direction was introduced by Cohen *et al*. (Cohen *et al*., 2003), in the form of a measure called Soft-TFIDF, which extends the Jaro-Winkler method to combine both the frequency weight of words in a cosine similarity measure and a CLOSE measure at the character level. The soft TFIDF algorithm works as follows: for each token A_i in the first name, find a word B_j in the second one that maximizes the similarity function. Moreau *et al*. (Moreau *et al*., 2008) showed that this may lead to double matching of words and proposed a generic model to enhance the soft TFIDF. Camacho *et al*. (Camacho *et al*., 2008) used a cost function that basically depends on matching all pairs of tokens and summing all edit distances at character level. The distance metric was modified by a frequency measure of the tokens. They also used a permutation factor – Monge-Elkan concept (Monge & Elkan, 1996) – to allow a non-ordered sequence of word tokens to be matched.

In this work, we propose a hybrid sequential algorithm that combines the advantages of token level and character level approaches to improve the name matching quality. The hybrid algorithm is optimized for matching Arabic names. As we will discuss in Section (3), Arabic names have a restricted writing order and close typographic pattern and are subject to middle token omission and omission of common name tokens, even if occurring at the beginning of names. Due to these characteristics, existing algorithms cannot be applied directly to matching Arabic names. Character-based hybrid algorithms may fail due the close typographic pattern, ignoring the sequence order of tokens. Also, allowing permutation of tokens conflicts with the

restricted sequence of writing names. To improve the matching efficiency, most hybrid techniques include weights for token frequency but do not give the same attention to the relative position where the mismatch occurs.

The proposed hybrid algorithm is an extension to the Levenshtein algorithm, with computing 'edit distances' at token level instead of character level in the basic algorithm. The sequential nature of the algorithm keeps the ordering importance of name tokens. The two names to be matched are considered as two bags of tokens, and the algorithm computes the cost of transforming one bag into the other. While the basic Levenshtein algorithm assigns a unity cost to all edit operations, the current algorithm assigns weights that reflect the importance of each edit operation. When matching a pair of tokens, the importance of the edit operation is determined by the relative position of the tokens in names, their frequency measure, and their character level partial similarity.

The remaining parts of this paper are organized as follows. In Section 2, we present some basic techniques for name matching. After that, Section 3 briefly discusses the characteristics of the Arabic naming system considered in our work. The proposed algorithm is described in Section 4. The results of experimental comparisons are discussed in Section 5. Finally, conclusions are discussed in Section 6.

2. Matching Algorithms for Names

Name matching is the process of determining whether two name strings are instances of the same name. Multiple methods have been developed for matching names, which reflects the large number of errors or transformations that may occur in real-life data (Elmagarmid *et al*., 2007). The basic goal of all techniques is to match names (or strings) that are not necessarily required to be identical. Instead of an exact match, a normalized similarity measure usually is calculated to have a value between 1.0 (when the two names are identical) and 0.0 (when the two names are totally different). There are several well-known methods for estimating similarity between strings, which can be roughly separated into two groups: character-based techniques and token-based techniques.

The Levenshtein algorithm and its variants are character-based matching techniques based on edit distance metrics, and the Levenshtein edit distance is defined originally for matching two strings of arbitrary lengths. It counts the minimum differences between strings in terms of the number of insertions, deletions, or substitutions required to transform one string into the other. A score of zero represents a perfect match.

The basic Levenshtein method has been extended in many directions (Hall & Dowling, 1980), for example, having an extension to consider reversals of order (transposition of characters) directly in the edit distance operation. Another direction of generalization is to

allow different weights at character level. The weights for replacing characters can depend on keyboard layout or phonetic similarities (Snae, 2007). In other research (Bilenko *et al*., 2003b), a distance function is produced by a distance function learner and the weights are learned from a training data set to have a combined record-level similarity metric for all fields.

The affine gap distance metric (Waterman *et al*., 1975) offers a smaller cost for gap mismatch; hence, it is more suitable for matching names that are truncated or shortened. Smith and Waterman (1981) described an extension of edit distance and affine gap distance to find the optimal local alignment at the character level. Regions of gaps and mismatches are assigned lower costs. Jaro (1989) introduced a string comparison metric that is dependent on both the number of common characters and the number of non-matching transpositions in the two strings.

Token-based approaches are motivated by the fact that most of the differences between similar named entities often arise because of abbreviations or whole word insertions and deletions. Hence, token models should produce a more sensitive similarity estimate than character-based approaches. Also, experimental results show that including token frequency as a parameter in matching algorithms leads to a significant improvement in matching accuracy. Jaccard-vector space cosine similarity is an example of difference measures that operate on tokens, treating a string as a "bag of words". In these approaches, the two string names to be compared are divided into sets of words (or tokens) before a similarity metric is considered over these sets.

The Jaccard similarity between the word sets A and B is defined as the size of the intersection divided by the size of the union of the sample sets: $J(A,B) = |A \cap B| / |A \cup B|$. The algorithm has been extended to compare bi-grams (paired characters of two string), tri-grams, or n-grams. Strings can be "padded" (Keskustalo *et al.*, 2003) by adding special characters at the beginning and end of strings, Padded n-grams will result in a larger similarity measure for strings that have the same beginning and end but errors in the middle.

TFIDF, or cosine similarity, is another measure that is widely used in the information retrieval community. The basic TFIDF makes uses of the frequency of terms in the entire collections of documents and the inverse frequency of a specific term in a document. The TFIDF weighting method is often used in the Vector Space Model together with Cosine Similarity to determine the similarity between two documents. Similarity between database strings, or between a database string and a query string, is computed via the cosine similarity (inner product) of the corresponding weight vectors, essentially taking the weights of the common tokens into account. The TFIDF similarity of two word sets A and B can be defined as:

$$\mathrm{TFIDF}(A,B) = \sum_{w \in A \cap B} V(w,A).V(w,B) \qquad (1)$$

where V is a weight vector that measures the normalized TFIDF of word w. Like Jaccard, the TFIDF scheme depends on common terms, but the terms are weighted; these weights are larger for words w that are rare in the collection of strings from which A and B are drawn. The basic TFIDF does not account for misspelling mistakes in words. Cohen *et al*. (2003) proposed a soft TFIDF with a heuristic that accounts for certain kinds of typographical errors.

Soft TFIDF is one approach that combines both string-based and token-based distances. In this approach, similarity is affected not only by the tokens w that appear in the sets A and B, but also for those "similar" tokens in A that appear in B.

$$\text{softTFIDF}(A,B) = \sum_{w \in \text{close}(\theta,A,B)} V(w,A).V(w,B)D(w,B) \quad (2)$$

Here, D is the character-based distance of the word w, such that it is greater than a threshold θ. The new set close allows one to integrate a token-based distance and the statistics of a particular corpus in the similarity evaluation of a particular word.

Both TFIDF and Soft TFIDF are insensitive to the location of words, thus allowing natural word moves and swaps (*e.g*., "John Smith" is equivalent to "Smith, John"). Although this is useful in many naming systems, it does not fit the Arabic naming system, which is characterized by restricted component order. Camacho *et al*. (2008) proposed a similar metric that combines both the frequency of words and the edit-based distances of each word pair of the two names. Also, strings may be phonetically similar even if they are not similar at the character or token level. Soundex (Holmes & McCabe, 2002), Phonex (Gadd, 1990), and Phonix (Gadd, 1990) are examples of phonetic-based techniques that convert the name into a sequence of codes that represent how the name is spoken. Phonetic representation of the names is used either for exact or approximate match.

When considering contextual information stored with names (such as address, mail, and other details) to increase the likelihood of a match, the problem is called data or record linkage (Xiao *et al*., 2011). Many techniques have been proposed for record linkage, where not only are pairs of name strings matched, but also many other matching features (Monge & Elkan, 1996). Winkler (2002) demonstrated how machine-learning methods could be applied in record linkage situations where training data are available. Name is time-independent information; therefore, even in feature-based approaches, having an effective name matching is crucial (Winkler, 2006). This work considers only name matching without taking any contextual information into account.

3. Characteristics of Arabic Name Variations

Exact string matching of personal names is problematic for all languages because names are often queried in a different way than they were entered. The proposed algorithm deals with the following problems concerning Arabic names.

3.1 Very Close Typographic Structure

Spelling errors normally occur during data entry. This may be due to typographical errors, cognitive errors, or phonetic errors. Whatever the reason for the error, the source and target names are considered strings differing at the character level. According to Jurafsky and Martin (Jurafsky *et al*., 2002), this type of error can be categorized as insertion, deletion or omission, substitution, or transposition.

There are efficient and well-established algorithms that deal with spelling errors variants for a string. When data is represented by relatively short strings that contain similar yet orthographically distinct tokens, character-based measures are preferable since they can estimate the difference between the strings with higher resolution.

The reason that misspelling errors are particularly difficult in Arabic names is the close typographical structure of names. For example, inserting the character "و" to the name "محمد," yields a correct name "محمود". Also, substituting the character "أ" with "م" in "دمحم," gives a correct name "أحمد". If the Levenshtein algorithm is used for matching two names with a length of 20 characters of each name, a single edit distance will show 95% matching similarity of the two names, while they are two different persons. The problem is how to know if the name is written incorrectly or refers to another person (*e.g*., his brother), especially when searching family databases.

3.2 Omission of Name Components

While it is common to have one first, one or more middle, and a surname name for writing a personal name, several variations exist in real free form names. The same problem exists in many other languages, and it has been reported by Borgman and Siegfried (Borgman & Siegfried, 1999) that there are no legal regulations of what constitutes a name. The source of the ambiguity, in many cases, is people themselves as they report their names differently depending upon the organization they are contacting. Examining different Arabic databases shows that name omission is a serious problem that should be handled efficiently in any Arabic name matching algorithm. Name omission is related to both position and frequency as follows.

3.2.1 Name Order

Persons tend to write their names in a restricted correct order. They may omit one or more tokens, while still keeping the correct order.

Examining different writing styles of Arabic names shows that transposition errors occur rarely. Therefore, one wants “Hamed Mohamed Fawzy Ibrahim” to match with “Hamed Mohamed Ibrahim” but not with “Hamed Fawzy Mohamed Ibrahim”. The built in sequential

nature of the proposed algorithm assigns one edit distance to 'omission' and 'two edit distances' to transposition.

3.2.2 Position of Omitted Name

Persons tend to omit one or more middle names, while fewer name omissions typically occur at the beginning or at the end of names.

The analysis shows that a person is keen on writing his first and surname carefully. This raises the position importance of the name variations. For example, one wants “Hamed Mohamed Fawzy Ibrahim” to match with “Hamed Mohamed Ibrahim” but not with “Mohamed Fawzy Ibrahim”. To realize the position relation, the proposed algorithm gives less importance to name omission – or insertion – occurring in the middle of the name, and more importance to first and last names.

3.2.3 Frequency of Omitted Name

Persons tend to omit non-significant components of their names, i.e., omission is likely to occur with common names.

Results of analyzing a sample of 7140 Egyptian full names show that nearly 30% of all name components lie within a set of only 9 common names, as shown in Table (1). In the proposed algorithm, less importance is given to a common name omission or insertion. For example, one wants “Hamed Mohamed Fawzy Ibrahim” to prefer the match with “Hamed Fawzy Ibrahim” over “Hamed Mohamed Ibrahim,” because 'Mohamed' is not as indicative a name as ' Fawzy'.

Table 1. Arabic Common Name Frequency

Arabic Name	English name	TF
محمد	Mohamed	11.38%
احمد	Ahmed	5.98%
محمود	Mahmoud	2.39%
على	Ali	2.28%
ابراهيم	Ibrahim	2.07%
حسن	Hassan	1.84%
السيد	Alsayed	1.54%
مصطفى	Mostafa	1.33%
حسين	Hossien	0.87%
Total percentage of top common Arabic name tokens		29.7%

Common names have another impact when searching the Web for famous persons. When searching for the former president of Egypt, many people do not know that his first name is 'Mohamed,' and search the web only for 'Hossny Mubark". The search engine should be smart enough to return also matches with his full name as top hits, because 'Mohamed' is a common name and is expected to be omitted. This is different from returning 'Gamal Hossny Mubark' – his son – since 'Gamal' is not a common name.

The previous discussion shows that the frequency distributions of name values can be used to improve the quality of name matching. They can be calculated either from the data set containing the names to be matched or from a more complete population-based database, like a telephone directory or an electoral roll.

3.3 Writing Styles Character Variations

One important component of the proposed work is name standardization. Standardization eliminates writing style character variations; hence, it makes the data comparable and more usable. To produce a uniform representation, the algorithm runs SQL script to replace various spellings of words with a single spelling. For instance, different prefixes, spacing, punctuations, and character variations are replaced with a single standardized spelling.

A name standardization (or character variation elimination) module is common module in name matching algorithms (PAtman & Thompson, 2003). In the current work, the standardization concept is optimized for Arabic names. For instance, the module trims certain prefixes such as (د. أ.د. م. دكتـــور, د/ ، السـيد/), replaces multiple blanks with a single blank, replaces the characters (أ، إ ، آ with ا), and replaces the ending character (ي) with (ى). There are some cases where the Arabic name component is composed of two tokens. For example, a prefix name component (عبـد) and a postfix name component (الـــدين) cannot be standalone names. There is no standard style for writing composite names, as it is not always necessary to have a distance space between them. To standardize composite names, either leading or trailing spaces are removed, whenever a pre/post tokens are detected. Name style standardization is an inexpensive step, but it improves the overall performance for name matching.

4. The Proposed Algorithm

We started with the Levenshtein edit distance similarity metric and extended it to handle name matching at a token level. The sequential nature of the Levenshtein method ensures that the sequential name order of tokens is considered. Following the variant of Needleman-Wunch (gap cost), the current algorithm replaces the fixed unity cost of the simple Levenshtein form with a cost function that is dependent on frequency and position of tokens to be matched.

Specifically, the implementation of the proposed algorithm applies three modifications to the basic Levenshtein distance metric. The first modification is the application of the same dynamic programming technique at the token level instead of the character level in basic Levenshtein. For example, the distance between the two names a = ('Mohamed,' 'Ahmed,' 'Hassan,' 'Ali') and b = ('Mohamed,' 'Hassan,' 'Ali,' 'Ibrahim') is two. This is because (a) requires two edit operations (deletion of the token 'Ahmed', and insertion of the token 'Ibrahim' at the end of a).

The second modification is the mapping of the frequency and position importance of name tokens – discussed in Sections 3.2 and 3.3 – with a cost function C, instead of assigning fixed unity cost for all edit operations. The role of the cost function C is to lighten (or strengthen) the effect of token mismatch according to word position and frequency.

The third modification is the implementation of partial matching of individual token pairs at the character level. This fine grained level ensures that pairs with slightly different misspellings are not ignored.

For two tokens (a_k, b_l), where $1 \le l \le L$ and $1 \le k \le K$

$$H(k,l) = min\begin{cases} H[k-1,l] + C_{k,l} \\ H[k,l-1] + C_{k,l} \\ H[k-1,l-1] + \text{TokenCost}(a_k, b_l) C_{k,l} \end{cases} \tag{3}$$

where $C_{k,l}$ is given by

$$C_{k,l} = P_{k,l}\ F_{k,l} \tag{4}$$

$P_{k,l}$ $F_{k,l}$ are the position and frequency costs defined in Sections (4-2) and (4-3), respectively.

TokenCost is the token-pair similarity cost at a character level. The final similarity percentage between the two name strings then is given by:

$$\text{sim}[a,b] = 1 - \frac{H[K,L]}{\max(K,L)} \tag{5}$$

4.1 Token Pair Mismatch Cost

The 'TokenCost' measure is the cost of partial match between tokens a[k] and b[l], and it captures word spelling errors at the character level. Pairs of tokens that are not necessarily identical are also considered in the edit operation but with a cost that depends on their similarity. The proposed 'TokenCost' measure combines the token level edit operations with approximate token matches. The concept is similar to the 'close' function in soft TFIDF. In our implementation, the 'TokenCost' function has a value that ranges from zero (for exact token match, thus having 0 required edit actions), to 1 (for completely mismatch, thus having 1

complete edit action). When similarity is below a certain threshold, the pairs are considered dissimilar and the distance function is set to one. It is computed with Levenshtein edit distance, and clipped at a threshold value θ. The threshold limit is implemented with the threshold rule:

$$\begin{aligned} &\text{TokenCost}(a_k, b_l) = \text{Levenshtein}(a_k, b_l) \\ &\text{IF Levenshtein}(a_k, b_l) \geq \theta \text{ then TokenCost}(a_k, b_l) = 1 \end{aligned} \tag{6}$$

For two names a and b, the current algorithm defines a Levenshtein cost of the two token words $a_k \in a$ and $b_l \in b$ as $0 \leq \text{Levenshtein}(a_k, b_l) \leq 1$. The distance cost ranges from zero for a complete match to one for a complete mismatch. The distance depends on Levenshtein edit metric. The basic Levenshtein algorithm is a character-based approach, which takes two strings, a_k and b_l of lengths m and n characters, and returns the Levenshtein distance between them. For all i and j, d[i,j] will hold the Levenshtein distance between the first i characters of a_k and the first j characters of b_l. The elements of d[i,j] are computed according to the following:

$$\begin{aligned} &\text{If } a_k[i] = b_l[j], \text{then } d[i,j] = d[i-1, j-1] \\ &\textit{Else } d[i,j] = min \begin{cases} d[i-1, j] + 1 & \textit{//a deletion} \\ d[i, j-1] + 1 & \textit{//an insertion} \\ d[i-1, j-1] + 1 & \textit{//a Substitution} \end{cases} \end{aligned} \tag{7}$$

The similarity measure then is given by:

$$\text{Levenshtein}[a_k, b_l] = \frac{d[m,n]}{\max(M,N)} \tag{8}$$

4.2 Position Mismatch Cost of Tokens

As in Western naming systems, an Arabic name's token order is very important. Family name usually appears as the last token of the name. Therefore, any successful name matching algorithm should allow for gaps of unmatched characters (*e.g.*, Smith-Waterman algorithm) and the problems of out of order of tokens (*e.g.* Monge-Elkan method). The proposed algorithm satisfies both constraints with more flexibility for adjusting the relative importance of token position.

Instead of using the number of edit operations as a distance metric, the proposed algorithm uses the cost of the edit operations required to transform one string to another. When matching complete names, inserting or deleting a token (name) at the beginning (first name) or at the end (family name) of a complete name has a different cost from doing the same edit actions for middle names. We call this position cost, which is implemented using a position weight cost $0 \leq P \leq 1$.

$P_{k,l}$ is the position weight for a matching token k with a token j. In general, the proposed position cost is flexible and can have different values at each token position. In our implementation, persons are keen to write their first and last names. For this reason, more importance is given when matching first and last names. Position weight is assigned a complete unity edit cost when matching either the two first or the two last tokens in both strings.

In our implementation, for two tokens (a_k, b_l), where $1 \leq l \leq L$ and $1 \leq k \leq K$, the position cost rule is given by:

$$P_{k,l} = \begin{cases} 1, & (k=1 \text{ and } l=1) \\ & \text{or}(k=K \text{ and } l=L) \\ \beta & \text{Otherwise} \end{cases} \quad (9)$$

The position rule is used to initialize the zero rows and columns, as shown in Table 1. Note that this initialization is different from Smith and Waterman (1981), which assigns zeroes to all zero rows and columns. Table (2) illustrates an example of computing H(l,k), defined in Equation (4), when considering only position weights to match two strings a [L]= (W1, W2, W3, W4, W5) and name b[K]= (W1, W3, W4, W6).

Table 2. Effect of position weight

		W1	W3	W4	W6
	0	1	1+β	1+2β	2+2β
W1	1	0	β	2β	3β
W2	1+β	β	β	2β	3β
W3	1+2β	2β	β	2β	3β
W4	1+3β	3β	2β	β	2β
W5	2+3β	4β	3β	2β	1+β

As given in Equation (9), the position weight has either a value of 1 or β. In Table (2), and according to Equation (9), the step increase of position weight (in zero rows and columns) is β, except for the first and last cells, which have a value of one. The remaining cells are computed based on Equation (4) while considering only position weight. For example, the entry at a cell (W2, W3) is the minimum of its up, left, and left-up cells plus β, which gives H(W2,W3)=β. Another example, H(W3.W3) is β, since the two words are matched and the IF-part of Equation (4) is applied.

The total distance cost then is H(W5,W6) divided by the maximum length of the two tokens, which is (1+ β) /5, Note that, in the original edit distance algorithm, a complete edit distance (β =1) is assigned to each position. By varying the value of (β≤1), one can control the

position importance of the token.

4.3 Frequent Name Mismatch Cost

Motivated by the assumption that 'persons tend to omit their common names,' we allow the cost of edit distances for a common name to have a different cost than a rare name. This concept is called a term frequency weight $0 \le F \le 1$, For two tokens (a_k, b_l), where $1 \le l \le L$ and $1 \le k \le K$, the term frequency is given by:

$$F_{k,l} = 1 - \alpha \frac{TF(a_k)TF(b_l)}{MTF^2} \tag{10}$$

where $TF(a_k)$ is the Term Frequency of token a_k, $TF(b_l)$ is the Term Frequency of token b_l, MTF is the maximum term frequency of names, and α is a frequency weighting factor. To allow the algorithm to have a maximum frequency effect, α is set to 1. In this case, the common name frequency cost and the overall cost of the edit operation are nearly 0.

5. Experimental Results

5.1 sets

We used two types of datasets: the base set and test sets. The base dataset contains a sample of 7140 names extracted from MIS of Egyptian university staff members. To make the base dataset usable, all names were standardized to eliminate writing style character variations, as explained in Section 3.3. The extracted sample contains two fields: 1) a Base name IDentifier B_ID, and the Base Name (BName).

The Arabic names dataset characteristics are shown in Table (3) and can be downloaded from http://www.scribd.com/doc/143493637/Arabic-Name-Dataset. The frequency distribution of the number of tokens in the sample indicates that 97.4% of the Arabic names were written in 3 to 5 tokens. The preliminary analysis of the sample shows that the sample contains 30 duplicated names (a percentage of 0.41%). Also, as shown in Table (3), the duplication of names is only 0.76% (actually 0.35% when subtracting full name duplications) when the Arabic name is written in four tokens.

The experiment used six different test datasets, with each containing 300 names extracted randomly from the base dataset. Each test set was subjected to a noise to induce one of the following errors: 1) deletion of random single character, 2) deletion of random two characters, 3) omitting the first token, 4) omitting the second token, 5) omitting the third token, or 6) omitting both the second and third tokens. Each test set had one type of distortion, but all test sets had a similar field structure: 1) Test name Identifier (T_ID), 2) the distorted name (DName), and 3) a Reference to the original base name Ref_B_ID. Then, we had six test sets

for which the true match status was known and each carrying one type of errors.

Table 3. Characteristics of the Arabic name dataset

No. of Words	Frequency	Duplication				
1	0	88.34%	35.64%	5.74%	0.76%	0.41%
2	0.1%					
3	14.4%					
4	62.6%					
5	20.4%					
6	2.2%					
7	0.2%					

5.2 Experimental Methodology

The purpose of all experiments was to measure the degree to which the proposed algorithm could overcome each type of inserted noise. The 300 distorted names of each test set were matched against the original 7140 base set. Since we are interested in automating the matching process, we defined 'success' as a direct measure of the algorithm performance. The testing environment was adopted to accept only the top-scoring name from the tested algorithm as a match. The match was counted as true if it corresponded to its original name in the base set. The success percentage was calculated as the count of true matches divided by the test set size.

For a single distorted name (DNname, Ref_B_ID), the test methodology consults the tested algorithm (the proposed algorithm or other algorithms), to compute the similarity against the base set. The test environment keeps the running maximum similarity and the Base name ID (Sim, B_ID) as a candidate matched name. This match is true when the final maximum similarity name is the same as the original base name. The algorithm is given as follows.

Algorithm Success-Match-Percentage
Input : Specific values of θ, α and β
Specific Distorted Data Set DDS_i (DNname, Ref_B_ID) , $1 \leq i \leq 300$
Base Data Set BDS_j (BName, B_ID), , $1 \leq j \leq 7140$

```
Begin
  True Match Count = 0;
  For (I = 1 to 300) {
       Max Running score = 0;
       Ref record = 0;
       For (j=1 to 7140) {
                    Apply Equation (5) to get Sim (DName(i), BName(j)
                          If Sim> Running max Score {
                    Set Max Running Score = Sim;
```

```
                    Set Ref Record = B_ID (j); }
            } End loop j
    If   Ref record = Ref_B_ID (i) {      /* success */
            Increment True Match Count by 1 ; }

  } End loop i
 Success percentage = True Match Count / 300;
End;
```

In summary, we used artificially generated files, with each carrying one type of error. Although the generated data sets do not approximate the types of errors that occur in real data, they are very useful in 1) analyzing the effect of different parameters on each type of error and 2) determining the upper and lower success limits of each algorithm.

5.3 Results

We ran our experiments on data sets, with each carrying only one type of error. Results of the proposed algorithm are summarized in Tables 4 to 9. Each cell entry is the percentage of success, which is calculated as the number of true matches of the test set names divided by the total number of names in the set. To illustrate the role of the position weight and individual token match threshold in evaluating a name matching, the experiment was repeated for different values of β and θ.

Table 4. One character omission success percentages

θ	β					
	0	0.1	0.3	0.5	07	1
0	50.33%	70.33%	91.00%	93.67%	94.67%	94.00%
0.1	50.33%	70.33%	91.00%	93.67%	95.67%	94.00%
0.3	82.33%	91.00%	95.33%	95.33%	97.67%	95.33%
0.5	87.67%	97.00%	97.33%	99.00%	99.00%	99.33%
0.7	88.00%	97.33%	98.67%	98.67%	98.00%	98.33%
1.0	87.33%	97.00%	98.67%	98.67%	98.00%	98.00%

Table 5. Two characters omission success percentages

θ	β					
	0	0.1	0.3	0.5	07	1
0	29.33%	39.00%	64.00%	70.67%	71.33%	72.00%
0.1	29.33%	39.00%	64.00%	79.67%	90.67%	82.00%
0.3	67.33%	78.33%	88.67%	90.33%	94.67%	91.00%

0.5	79.67%	92.00%	96.00%	96.33%	97.33%	96.67%
0.7	78.67%	94.33%	96.67%	96.67%	96.33%	95.33%
1.0	78.67%	94.33%	96.67%	96.67%	95.67%	95.00%

Table 6. First token omission success percentages

θ	β					
	0	0.1	0.3	0.5	07	1
0	2.33%	34.00%	74.33%	82.33%	87.00%	89.00%
0.1	2.33%	34.00%	74.33%	82.33%	87.00%	89.00%
0.3	2.67%	28.67%	66.33%	76.67%	85.33%	88.67%
0.5	4.00%	16.33%	53.00%	67.33%	81.33%	84.67%
0.7	4.00%	10.00%	41.33%	63.67%	78.67%	82.00%
1.0	4.33%	8.33%	37.00%	62.33%	76.67%	79.33%

Table 7. Second token omission success percentages

θ	β					
	0	0.1	0.3	0.5	07	1
0	92.33%	99.33%	99.33%	99.67%	99.67%	99.00%
0.1	92.33%	99.33%	99.33%	99.67%	99.67%	99.00%
0.3	92.33%	99.00%	98.67%	98.00%	97.33%	96.67%
0.5	92.33%	98.67%	98.33%	97.33%	96.00%	93.33%
0.7	92.33%	98.00%	97.67%	97.00%	95.67%	90.00%
1.0	92.33%	97.33%	97.00%	96.67%	95.33%	87.67%

Table 8. Third token omission

θ	β					
	0	0.1	0.3	0.5	07	1
0	77.00%	83.67%	91.00%	94.33%	95.00%	71.00%
0.1	77.00%	83.67%	91.00%	94.33%	95.00%	71.00%
0.3	77.00%	83.33%	89.00%	90.67%	93.67%	65.67%
0.5	77.00%	83.33%	85.67%	89.33%	93.00%	60.67%
0.7	77.00%	83.33%	84.67%	89.00%	92.67%	55.00%
1.0	77.00%	82.33%	83.00%	88.00%	92.00%	51.67%

Table 9. Second and third tokens omission success percentages

θ	β					
	0	0.1	0.3	0.5	07	1
0	76.67%	76.33%	79.67%	79.67%	80.33%	71.00%
0.1	76.67%	76.33%	79.67%	79.67%	80.33%	71.00%
0.3	76.67%	76.00%	78.00%	78.00%	77.00%	65.67%
0.5	76.67%	76.00%	78.00%	78.00%	74.00%	60.67%
0.7	76.67%	76.00%	78.00%	78.00%	73.67%	55.00%
1.0	76.67%	75.67%	77.33%	77.33%	72.00%	51.67%

In general, the results show that the algorithm succeeds in overcoming single character and second token omissions with nearly 100% accuracy when properly setting the parameter values. Names with two character omission are matched successfully with 97.33% accuracy. Next is the third token omission with a success percentage of 95.00%. Table (6) shows that the first token omission is a serious error, and the algorithm succeeds only with 89% accuracy in matching correct names. The worst accuracy of 79.67% is achieved when a person omits both his/her second and third names.

5.3.1 Effect of Threshold and Position Weights

The results show that there is a conflicting need for the value of θ required for obtaining optimum results for all types of errors. Character omission errors require a moderate value (θ = 0.5), while overcoming token omission requires lower values of θ ranging from 0 to 0.1. It is clear that, for lower thresholds, the similarity of token cost at character level (the third term of the minimum relation of Equation 3), always dominates, and the system behavior is mainly a character level operation. On the other hand, setting the threshold θ = 1, a unity edit cost is assumed and the algorithm neglects all character variations at the token level.

The position weight β required to obtain best results has a fixed pattern for both characters and token omission errors. Better results usually are found when β ranges from 0.5 to 0.7. The only exception is the first token omission error, which needs higher values of β. To explain the effect of β values, let us return to Equation (3). If β has a maximum value of 1, no positional information will be used, since a unity edit cost is assumed whatever the position is and the algorithm behavior will give equal importance to all token variations, which corresponds to a static string distance. Also, for very low values of β, the algorithm neglects nearly all token level variations other than first and last tokens.

5.3.2 Comparing Results with other Algorithms

We now present an experiment comparing our hybrid weighting technique (run at α =1, β=0.7,θ =0.1) with different state-of-the-art systems using the same Arabic data sets and the same experimental methodology. The tested algorithms are: Levenshtein, Monge-Elkan, Jaro-Winkler (JW), and Soft-TFIDF (run at JW, θ =0.9). The Java open-source toolkit of tested algorithms is available at http://secondstring.sourceforge.net/) (Cohen *et al*., 2003).

When considering only character misspelling errors, the basic Levenshtein algorithm did the best, followed by Jaro-Winkler with a success of 95%. Both the proposed algorithm and Soft TFIDF had the same average success rate of 93% for character level errors, while the Monge-Elkan algorithm returned the worst accuracy result, where nearly 45% of the distorted names were matched incorrectly.

For token omission errors, as shown in Table 10, our proposed algorithm generally seems the best when considering either the success boundary or individual token's errors. It had a better success range (from 80.3% to 99.7%) than Soft TFIDF (from 69.0% to 94.3%). Part of this better performance may be due to missing prior Arabic frequency knowledge for the Soft TFIDF algorithm. The next best performance for the tested algorithms is the Monge-Elkan method, which had a success boundary ranging from 23.0% to 89.3%. The worst result comes from Jaro-Winkler, with success ranging from 8.3% to 72.3%.

Table 10. Comparison of success percentages of the proposed algorithm with existing matching algorithms using Arabic dataset.

Omission Error Type	Proposed Average β=0.7 θ =0.1	Basic Lev.	Monge Elkan	Jaro Winkler (JW)	Soft TFIDF JW θ =0.9
One character	95.7%	100%	66.7%	96.0%	95.7%
Two characters	90. 7%	100%	44.3%	94.0%	90.3%
First token	87.0%	90.7%	89.3%	72.3%	86.0%
Second token	99.7%	67.7%	40.3%	68.7%	94.3%
Third token	95.0%	65.3%	35.0%	51.3%	91.3%
Second & third	80.3%	16.0%	23.0%	8.3%	69.0%

It is important to note that the actual success percentage of all presented algorithms will be at a point between their upper and lower boundary limits in real data sets. The actual success operation of any algorithm depends also on the mixture amount of different types of errors in the real sets.

5.3.3 Practical Implementation of the Algorithm

The proposed algorithm was built as a tool for integrating cluttered and heterogeneous databases in universities as a part of activities of the Egyptian Universities Portals Project EUP. The project was funded by Ministry of Higher Education MOHE in Egypt. The official data of staff members are stored in SQL server databases – across universities – and include many fields, such as primary key fields, staff national ID, staff name, and address. Also, there exist many simple databases (in many cases only Excel sheets), that hold other activities within each university, such as Training File Records. Training records include staff name, course title, completion date, and other attributes. For each new course, new records are added to the file describing attendance with repeated staff member names. Therefore, the only available field for integrating both databases to get all courses for a staff member is matching the names in both databases.

The proposed algorithm was adopted to copy primary key fields from staff MIS table as foreign key fields in training database tables when names in both tables are matched. In Benha University, the staff table holds data of nearly 3500 staff members, while the number of records in the training database was 2200. When trying to link both databases using exact name matching, only 14% of names stored in the training database were identified. In implementing the proposed algorithm, the success in linking both tables reached 98.5% without any manual intervention. It is important to note here that the success factor greatly exceeds the minimum success level described previously through testing experiments. This is because the implemented heavy distortion scenarios were the worst cases that any matching algorithm should overcome. For example, real names to be matched cannot all have two missed tokens. Nevertheless, this high linking success level cannot be generalized since it is strongly dependent on the nature of errors existing in the names to be matched. Therefore, we highlight the importance of setting referenced criterion based on error types when comparing the success percentage of matching algorithms.

6. Conclusions

In this work, the basic character-based Levenshtein approach has been extended to a token-based distance metric. The algorithm is enhanced to combine the minor misspelling differences at the character level and both position and frequency parameters at the token level.

The experimental results demonstrate that taking position weight β and character threshold θ into account to weight distance metrics improves the performance of name matching. The proper value for θ is critical since it determines the granularity level behavior of the algorithm. For example, if θ is set to one, the algorithm ignores all individual token spelling errors and tends to be a token level algorithm. On the other hand, setting θ to zero, the

algorithm considers all token pairs to be matched with a similarity score; hence, the operation of the algorithm approaches character level algorithms. Sets that are characterized by more spelling errors need higher thresholds, while those that are characterized with token omissions need lower thresholds. Nevertheless, our presented hybrid name matching algorithm is able to work efficiently under different types of errors. Due to flexibility in changing the algorithm parameters, it can be optimized according to dataset characteristics to obtain better results.

Comparing the performance of different name matching algorithms is still a problematic issue. It was agreed that even metrics that demonstrate high performance for some data sets can perform poorly on others (Bilenko & Mooney, 2003). In a real environment, one cannot expect the matched set characterization. Therefore, in this work, we follow another comparison methodology by setting upper and lower limits of success of tested algorithms based on a variety of error schemes they may face in real sets. Clearly, highly spanned success algorithms are better and will have robust performance for different datasets. Also, since we intend to automate the whole matching process, we defined a 'success' of a tested algorithm as its ability to produce the correct matched name as a top scoring name. Using this methodology with a large Arabic dataset, the average performance of the proposed algorithm was compared with other classical algorithms. It was found that it raises the minimum success level from 69% – of best tested Soft TFIDF algorithm – to about 80%, (for second and third token omission), while achieving an upper accuracy limit of 99.67%. The best accuracy is lower than the basic Levenshtein algorithm, which achieves 100% upper accuracy levels for single and two character omission. Nevertheless, basic Levenshtein achieves only 16% as a lower accuracy level.

The concepts used in this algorithm are general and can be applied to name matching in many other languages in which the naming system is characterized by writing names in a restricted correct order, omission of one or more middle names, omission of non-significant components of names, and rare use of abbreviations for tokens. The only modification required to apply the proposed algorithm to other languages is the replacement of the Arabic name frequency table with the specific language one.

As a future work, we are working to upgrade the algorithm to deal with cross-language name matching. Name matching across languages requires approximate mapping of a name from one language into the other before applying a name matching algorithm. Nevertheless, the problem is not a one-to-one character (or phoneme) replacement of name components in both languages, since different writing styles should be considered. For example, the current algorithm has to be modified to deal with name abbreviations and a change of token order in other languages.

References

Bilenko, M., Mooney, R., Cohen, W., Ravikumar, P., & Fienberg, S. (2003). Adaptive name matching in information integration. *Intelligent Systems, IEEE*, *18*(5), 16-23.

Bilenko, M., & Mooney, R. (2003). Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining (2003)*, 39-48.

Borgman, C. L., & Siegfried, S. L. (1999). Getty's Synoname™ and its cousins: A survey of applications of personal name-matching algorithms. *Journal of the American Society for Information Science*, *43*(7), 459-476

Camacho, D., Huerta, R., & Elkan, C. (2008). An Evolutionary Hybrid Distance for Duplicate String Matching. http://arantxa.ii.uam.es/~dcamacho/ StringDistance/ hybrid-distance.pdf .

Cohen, W., Ravikumar, P., & Fienberg, S. (2003). A comparison of string distance metrics for name-matching tasks. In *Proceedings of the IJCAI-2003 Workshop on Information Integration on the Web (IIWeb-03)*, 73-78.

Elmagarmid, A., Ipeirotis, P., & Verykios, V. (2007). Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering*, *19*(1), 1-16.

Gadd, T. N. (1990). PHONIX: The algorithm. *Program: electronic library and information systems*, *24*(4), 363-366.

Hall, P., & Dowling, G. (1980). Approximate string matching. *ACM Computing Surveys (CSUR)*, *12*(4), 381-402.

Holmes, D., & McCabe, M. (2002). Improving precision and recall for soundex retrieval. In *Information Technology: Proceedings. International Conference on Coding and Computing 2002*, 22-26.

Jaro, M. (1989). Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida. *Journal of the American Statistical Association*, *84*(406), 414-420.

Jurafsky, D., Martin, J., & Kehler, A. (2002). *Speech and language processing: an introduction to natural language processing, computational linguistics and speech recognition* (Vol. 2). Prentice Hall.

Keskustalo, H., Pirkola, A., Visala, K., Leppänen, E., & Järvelin, K. (2003). Non-adjacent digrams improve matching of cross-lingual spelling variants. In *String Processing and Information Retrieval, Lecture Notes in Computer Science* Volume 2857, 252-265. Springer Berlin/Heidelberg.

Monge, A., & Elkan, C. (1996). The field matching problem: Algorithms and applications. In *Proceedings of the second international Conference on Knowledge Discovery and Data Mining*, 267-270.

Moreau, E., Yvon, F., & Cappé, O. (2008). Robust similarity measures for named entities matching. In *Proceedings of the 22nd International Conference on Computational Linguistics*, *1*, 593-600.

Patman, F., & Thompson, P. (2003). Names: A new frontier in text mining. *Intelligence and Security Informatics, Lecture Notes in Computer Science* Volume 2665, 960-960.

Smith, T., & Waterman, M. (1981). [a]Identification of Common Molecular Subsequences. [o] *J. Molecular Biology*, *147*, 195-197.

Snae, C. (2007). A comparison and analysis of name matching algorithms. *International Journal of Applied Science, Engineering and Technology*, *4*(1), 252-257.

Waterman, M., Smith, T., & Beyer, W. (1976). Some biological sequence metrics. *Advances in Mathematics*, *20*(3), 367-387.

Winkler, W. (2002). Methods for record linkage and Bayesian networks. Technical report, *Statistical Research Division*, (2002). US Census Bureau, Washington, DC.

Winkler, W. (2006). Overview of record linkage and current research directions. In Bureau of the Census. (2006).

Xiao, C., Wang, W., Lin, X., Yu, J., & Wang, G. (2011). Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems (TODS)*, *36*(3), 15.