An Edit Distance Algorithm with Block Swap

Tian Xia[1,2]
1 Key Laboratory of Data Engineering and Knowledge Engineering (Renmin University of China), MOE, Beijing 100872, China
2 School of Information Resource Management, Renmin University of China. Beijing 100872, China
xiat@ruc.edu.cn

Abstract

The edit distance between two given strings X and Y is the minimum number of edit operations that transform X into Y. In ordinary course, string editing is based on character insert, delete, and substitute operations. It has been suggested that extending this model with block edits would be useful in applications such as DNA sequence comparison and sentence similarity computation. However, the existing algorithms have generally focused on the normalized edit distance, and seldom of them consider the block swap operations at a higher level. In this paper, we introduce an extended edit distance algorithm which permits insertions, deletions, and substitutions at character level, and also permits block swap operations. Experimental results on randomly generated strings verify the algorithm's rationality and efficiency. The main contribution of this paper is that we present an algorithm to compute the lowest edit cost for string transformation with block swap in polynomial time, and propose a breaking points selection algorithm to improve the computation speed.

Keywords: Edit distance, edit operation, block swap, string matching.

1. Introduction

Measuring the similarity of Strings is a basic problem in computer science. It plays an import role in many fields such as information retrieval, natural language processing, computational biology, optical character recognition, image processing, spell checking, pattern recognition and pattern matching[1,2]. A fundamental measure of similarity between strings is the edit distance (a.k.a., Levenshtein distance), which is the minimum number of character insertions, deletions, and substitutions needed to transform one string to another. There are a number of well-known algorithms for computing edit distances, and/or for solving other more-or-less directly related problems, many of these algorithms are very useful in related applications. However, the edit distances, as defined so far, are not suitable for some applications since their basic operations do not concern the large block swap conditions. For instance, sentences with only clause position's difference are very similar by intuition, but they have low similarity when computed by traditional edit distance algorithm.

In this paper, we discuss the common edit distance algorithm and its limitations, and then, we propose an extended edit distance algorithm which permits block swap operations. Furthermore, we iterate extract the most possible substring blocks between two strings according to the longest common substrings, and use suffix trees to do fast string matching, this breaking points selection algorithm by substring blocks can improve the computation speed significantly.

The rest of this paper is organized as follows: In section 2, we give a brief overview of edit distance algorithm and its limitations. In section 3, we present our edit distance algorithm which permits block swap operation, and propose the breaking points selection algorithm to improve the computation speed. Experimental results according to the edit cost ratio and computation time usage are presented in section 4. In section 5, we briefly review related work. And finally, we conclude and discuss the future work.

2. Fundamentals

Let $\sum$ be a finite alphabet and $\sum^*$ be the set of all strings over $\sum$. According to the notation of [3], Let $X=X_1X_2 \ldots X_n$ be a string of $\sum^*$, where X_i is the i^{th} symbol of X. we denote $X_{i..j}$ the substring of X, which includes the symbols from X_i to X_j ($1 \le i, j \le n$). The

978-0-7695-3398-8/08 $25.00 © 2008 IEEE
DOI 10.1109/ICYCS.2008.14

length of such a string is defined as $|X_{i...j}|=j-i+1$. If $i>j$, $X_{i...j}$ is called the null string λ, $|\lambda|=0$.

An elementary edit operation is a pair $(a, b) \neq (\lambda, \lambda)$, where both a and b are strings of lengths 0 or 1. (a, b) is usually written as $a \rightarrow b$. There are three types of edit operations: insertions, deletions, and substitutions, which take the forms $\lambda \rightarrow b$, $a \rightarrow \lambda$, *and* $a \rightarrow b$, respectively. An edit transformation of X into string Y is a sequence S of edit operations that transforms X into Y. Edit operations can be weighted by an arbitrary weight function γ that assigns each operation $a \rightarrow b$ a non-negative real number $\gamma(a \rightarrow b)$. The function γ can be extended to edit transformations $S=S_1S_2...S_m$ by:

$$\gamma(S) = \sum_{i=1}^{m} \gamma(S_i)$$

The edit distance is the total number of operations necessary to transform X into Y. If we do not allow multiple operations in the same position in X, then the problem of computing the edit distance is the shortest path problem in the weighted graph $G_{X,Y}$ between verticals $(0, 0)$ and (n, m) [4]. Given $X, Y \in \sum^*$, the edit distance δ between X and Y is defined as below:

$\delta(X, Y)=min\{\gamma(S)|S$ *is an edit transformation from* X *to* $Y\}$

The above expression can also be defined recursively as:

$$\delta(X_{1...i}, Y_{1...j}) = \min\begin{Bmatrix} \delta(X_{1...i-1}, Y_{1...j}) + \gamma(X_i \rightarrow \lambda) \\ \delta(X_{1...i-1}, Y_{1...j-1}) + \gamma(X_i \rightarrow Y_j) \\ \delta(X_{1...i}, Y_{1...j-1}) + \gamma(\lambda \rightarrow Y_j) \end{Bmatrix} \quad (2.1)$$

According to the above formula, the edit distance can be calculated by using dynamic programming, its time and space complexity are both $O(|X| \bullet |Y|)$. Furthermore, the space complexity can be easily decreased to $O(min(|X|, |Y|))$ when the middle results do not need to store anymore.

Standard edit distance allows the relationship between two strings to be expressed graphically by means of an alignment. For instance, how "beautify girl" can be transformed into "beautiful girl" is showing below:

b	*e*	*a*	*u*	*t*	*i*	*f*	*y*	λ		*g*	*i*	*r*	*l*
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
b	*e*	*a*	*u*	*t*	*i*	*f*	*u*	*l*		*g*	*i*	*r*	*l*

As a rule, the arrows in an alignment are not allowed to cross. Furthermore, the character-to-character is determined on an individual basis, with no regard to higher level structure[5]. Considering the alignment comparing the strings X ("beautiful girl") and Y ("girl beautiful"):

λ	λ	λ	λ		*b*	*e*	...			*g*	*i*	*r*	*l*
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
g	*i*	*r*	*l*		*b*	*e*	...			λ	λ	λ	λ

The cost of this alignment is four insertions and four deletions. By overlooking the higher level structure, there is a movement of the word "girl" from the end of the string to the beginning. However, standard edit distance produces an alignment that seems to miss the true relationship between these two similar strings.

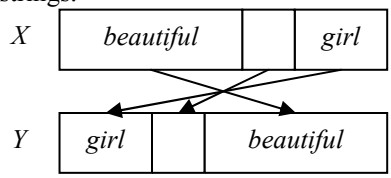

Figure 1. **Matching with block**

To solve this problem, block swap should be considered on the higher level. For above instance, "beautiful girl" can be treated as three blocks: "beautiful", " ", and "girl", and we can do swap as shown in figure 1.

3. Algorithm with block swap

3.1. Definition

Let $X=X_1X_2 \ldots X_n$ be a string of $\sum^*$, $|X| \geq 2$, then X can be divided into three successive blocks: the Head Block (*HB*), the Middle Block (*MB*), and the Tail Block (*TB*) of string *X*., i.e., $X=HB \cup MB \cup TB$. Moreover, we restrict the size range of each block as: $1 \leq |HB| \leq |X|-1$, $0 \leq |MB| \leq |X|-|HB|$, $1 \leq |TB| \leq |X|-|HB|-|MB|$.

It is obvious that the head block cannot be null, while the middle block and the tail block can be null, but they cannot be both null simultaneously. Because of non-null head block, it ensures that each block's size must be less than the original string's.

Lemma 1: Every n-length string X with $n \geq 2$ has $C(m+1,2)$-1combinations.

Proof: This problem can be viewed as inserting two breaking points into the string *X*. Since the size of the head block is greater than 1, $X=X_1 \wedge X_2 \wedge \ldots \wedge X_n$ has n different breaking points, where $\wedge$ represents the breaking point. When the middle block is not null, there are $C(m,2)$ combinations. Otherwise, two breaking points are overlapped together. It is invalid when the middle block and the tail block are both nulls, so, there are m-1 valid divisions under this condition. Therefore, the total valid combinations is $C(m,2)+(m-1)= C(m+1,2)$-1.

The block swap operation is to swap the head block and the tail block of a string, but to keep the middle block unchanged. Block swap operation can also be weighted by an arbitrary weight function γ_{swap}. Now, we define the edit distance denoted by $\delta_S(X, Y)$ as the total number of operations necessary to transform X

into Y, including insertions, deletions, substitutions and block swaps.

3.2. Edit Distance Computation

Since two strings are divided into three different types of blocks, we recursive define the edit distance $\delta_S(X, Y)$ as the minimum value of block edit distance sum for each possible divisions:

$$\delta_S(X,Y) = \min_{\forall division} \begin{Bmatrix} \delta_S(HB_X, HB_Y) \\ +\delta_S(MB_X, MB_Y) \\ +\delta_S(TB_X, TB_Y), \\ \delta_S(HB_X, TB_Y) \\ +\delta_S(MB_X, MB_Y) \\ +\delta_S(TB_X, HB_Y) \end{Bmatrix} \quad (3.1)$$

Suppose all insert operations have the same weight denoted as γ_{ins}, and delete operations denoted as γ_{del}. For the string $X=X_{i1\cdots im}=X_{i1}X_{i2}\ \ldots\ X_{im}$ and the string $Y=Y_{j1\ldots jn}=Y_{j1}Y_{j2}\ \ldots\ Y_{jn}$, we give each of the steps to computing the $\delta_S(X, Y)$ in detail as follows:

(1) If $im<i1$, then $X_{i1\ldots im}$ is null (λ), we need (jn-$j1$+1) insert operations to transform X into Y, and $\delta_S(X, Y)$ can be computed as follows:

$$\begin{aligned} \delta_S(X,Y) &= \sum_{j=j1}^{jn} \gamma(\lambda \to Y_j) \\ &= (jn - j1 + 1) \times \gamma_{ins} \end{aligned} \quad (3.2)$$

(2) If $jn<j1$, then $Y_{j1\ldots jn}$ is null (λ), we need (im-$i1$+1) delete operations to transform X into Y, and $\delta_S(X, Y)$ can be computed as follows:

$$\begin{aligned} \delta_S(X,Y) &= \sum_{i=i1}^{im} \gamma(X_i \to \lambda) \\ &= (im - i1 + 1) \times \gamma_{del} \end{aligned} \quad (3.3)$$

(3) If $i1=im$, then $X_{i1\ldots im}$ is a basic unit(a character of a string), therefore:

$$\delta_S(X,Y) = (jn - j1) \times \gamma_{ins} + \sum_{j=j1}^{jn} \gamma(X_{i1} \to Y_j) \quad (3.4)$$

(4) If $j1=jn$, then $Y_{j1\ldots jn}$ is a basic unit(a character of a string), therefore:

$$\delta_S(X,Y) = (im - i1) \times \gamma_{del} + \sum_{i=i1}^{im} \gamma(X_i \to Y_{j1}) \quad (3.5)$$

(5) Otherwise, both X and Y can be divided into small blocks, then we compute the edit distance for each block pairs. Let $\delta_1(X, Y)$ denote the edit distance without block swap operation, and $\delta_2(X, Y)$ denote the edit distance with block swap operation, then:

$$\delta_1(X,Y) = \min_{\substack{1\le i\le|X|-1 \\ 0\le lx\le|X|-i}} \min_{\substack{1\le j\le|Y|-1 \\ 0\le ly\le|Y|-j}} \begin{Bmatrix} \delta_S(X_{i1\ldots i}, Y_{j1\ldots j}) \\ +\delta_S(X_{i+1\ldots i+lx}, Y_{j+1\ldots j+ly}) \\ +\delta_S(X_{i+lx+1\ldots im}, Y_{j+ly+1\ldots jn}) \end{Bmatrix} \quad (3.6)$$

$$\delta_2(X,Y) = \min_{\substack{1\le i\le|X|-1 \\ 0\le lx\le|X|-i}} \min_{\substack{1\le j\le|Y|-1 \\ 0\le ly\le|Y|-j}} \begin{Bmatrix} \delta_S(X_{i1\ldots i}, Y_{j+ly+1\ldots jn}) \\ +\delta_S(X_{i+1\ldots i+lx}, Y_{j+1\ldots j+ly}) \\ +\delta_S(X_{i+lx+1\ldots i_m}, Y_{j1\ldots j}) \\ +\gamma_{swap} \end{Bmatrix} \quad (3.7)$$

Once we obtain the $\delta_1(X, Y)$ and $\delta_2(X, Y)$, the final edit distance between X and Y is computed:

$$\delta_S(X,Y) = min\{\delta_1(X,Y), \delta_2(X,Y)\} \quad (3.8)$$

According to the above analysis, edit distance algorithm with block swap can be computed via dynamic programming in polynomial time, and time complexity is $O(|X|^4 \bullet |Y|^4)$, it is higher than the algorithm presented by Gregor with $O(|X|^3 \bullet |Y|^3)$ time complexity[6]. However, Gregor's algorithm only supports adjacent block swap, it is not suitable for block movement in large range.

3.3. Algorithm Improvement

The algorithm we proposed above is to enumerate all possible divisions between two strings. However, once two strings X and Y were given, we do not need to enumerate all their divisions. In most cases, the common substring between X and Y should not be divided again. Therefore, the strings "I like this book" and "this book I like" only have three break points separately: "I like／ ／this book／", "this book／ ／I like／".

In order to select a small set of the breaking points, we need to divide X and Y into the most possible substring blocks. Suppose all the blocks of X are stored into a double linked list L_X, and the blocks of Y are stored into L_Y. Each block node has five basic attributes: previous pointer, next pointer, start index and end index of the original string, and a divided flag which indicates the current block has been divided or not. For the string $X=X_1\ldots X_i$ and $Y=Y_1\ldots Y_j$, the breaking points selection algorithm (BPSA) we proposed is described as follows.

Step 1: Initialize

L_X=[(*NULL, NULL, 1, i, false*)]

L_Y=[(*NULL, NULL, 1, j, false*)]

Step 2: Split

For the current block node b_x retrieved from L_X, we compute the longest common substring (LCS) between b_x and each undivided block node b_y of L_Y. Once the maximum length of the LCS is found, current b_x and b_y are both divided into three block nodes: the undivided left node represents the substring before LCS, the undivided right node represents the substring after LCS, and the divided node represents the current LCS. After, we divide the left node and the right node of b_x with L_Y recursively.

The LCS[7] of b_x and b_y is the longest string z which occurs in both the part string represented by b_x and the part string represented by b_y. To extract the LCS efficiently, we build a compacted trie tree of all suffixes of b_x and b_y, such that each suffix of b_x and b_y corresponds to unique root-to-leaf paths, and then, the LCS is the path-label of the deepest node with suffixes from both b_x and b_y as leaves in its sub-tree, and its time complexity is $O(m+n)$.

Step 3: Merge

In some cases, there are many 1-length successive blocks in the block list, and it can slow down the processing speed significantly. Furthermore, different edit operations for 1-length block sometimes may have the same cost. Therefore, we merge all successive blocks with 1-length to one large block after step 2.

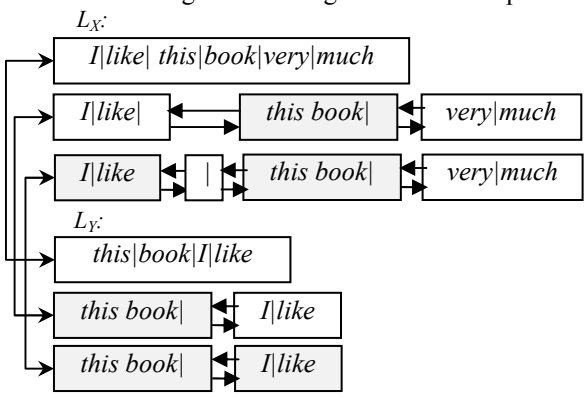

Figure 2. **An Illustration of BPSA**

Figure 2 illustrates the procedures of BPSA, where "|" represents the space character. For this instance, all possible break points number of X is 26, and Y is 16, but the reserved break points number of X is only 4, and Y is 2.

Since only a small set of divisions are reserved, therefore the step 5 in section 3.2 should also be revised. During each recursion, if the current string X or Y is represented by a single block in the list, we compute the edit distance by formula 2.1 directly. Otherwise, we only use the reserved break points to make the block divisions in the formula 3.6 and 3.7 correspondingly.

4. Experimental results

The algorithm presented in this paper has been implemented in java language (JDK1.6.0), and executed on a dual 1.83G Intel Core processor with 2G RAM. All the tested algorithms in our experiments are shown below:

A1: Traditional Edit Distance Algorithm.

A2: Gregor's Algorithm [6].

A3: Our Algorithm without BPSA Processing.

A4: Our Algorithm with BPSA Processing (No Merge).

A5: Our Algorithm with BPSA Processing (Merging 1-length successive blocks)

To build the test dataset, we generate 1,000 <X, Y> string pairs by randomly generated strings over a finite alphabet $\sum$= {0, 1} where $|X|=|Y|=n$. The simple quality measure we use in the experiments is the edit cost ratio, which is the quotient between the edit distance and the string length n, and all the values are obtained as an average over 10 runs.

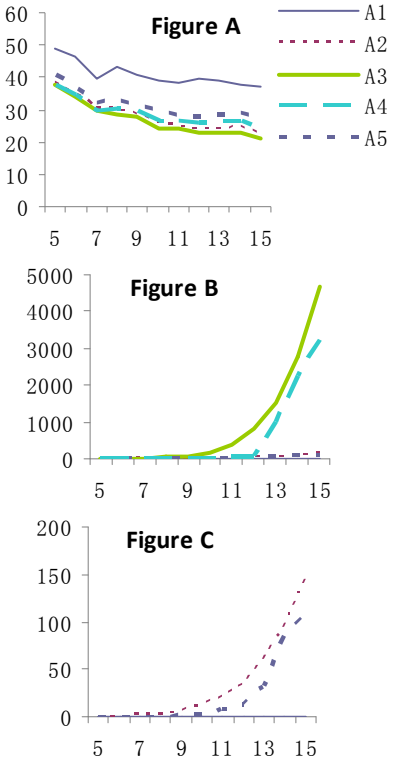

Figure 3. **The edit cost ratio and the time usage for different algorithms ($n \in$ [5, 15])**

The experimental results are illustrated in figure 3 with three small figures marked as A, B and C. In each small figure, the horizontal axis represents the different string length ranged from 5 to 15, the vertical axis in figure A represents the percentage of the edit cost ratio, and it indicates the millisecond time usage both in

figure B and C. It is obvious that the cost of transforming one string to another by algorithm A1 is very high, while algorithm A3 has the lowest cost for the same test dataset. For algorithm A3 and A4, they have lower cost compared with A1, but a little higher than A2 as shown in table 1.

Table 1. **The edit cost ratio on average.**

A1	A2	A3	A4	A5
40.89%	28.18%	26.84%	28.98%	30.47%

Though algorithm A3 has the lowest cost for transformation, its speed is the slowest. A4 is also very slow as illustrated in figure B. In figure C, we only reserve the lines of A2 and A5, so we can see their difference in more detail. Taking the edit cost and speed into account, algorithm A5 we proposed in this paper is more reasonable for string transformation.

Since only a subset of the break points are reserved in A4 and A5, the cost is higher than A2 or A1 in some cases. For instance, Let X="000101" and Y="000001", then, L_X="0001/\01", and L_Y="00/\0001 ", the edit distance computed by A4 or A5 is 2, which is higher than the traditional edit distance with only one substitute operation. Therefore, how to divide the strings into blocks more reasonable and efficient is still need a lot of work to do in our future work.

5. Related work

String matching is one of the most widely studied problems in computer science. Edit distance, proposed by Levenshtein[8], is the earliest and the most classic metrics for approximate string matching. Previous work has generally focused on normalized edit distance (NED) [3, 9, 10]. The NED between two strings X and Y is defined as the minimum quotient between the sum of weights of the edit operations required to transform X into Y and the length of the editing path corresponding to these operations, and an algorithm for computing the NED has been introduced by Marzal and Vidal that exhibits $O(|X|\bullet|Y|^2)$ computing complexity [3].

Traditional edit distance algorithm has only three operations, to extend the model more suitable in some real-world applications such as molecular biology and sentence similarity computation, some new edit operations have been introduced. Lowrance and Wagner added the swap operation to the set of operations defining the distance metric[11]. The swap, where the order of two consecutive symbols is reversed, is one of the most typical typing errors, and Amir presented an algorithm that solves the pattern matching with swaps problem in time $O(nm)$ [12]. Furthermore, Amir discussed the time complexity of the approximate string matching problem with swap and mismatch as the edit operations[13]. Graham provided a deterministic near-line algorithm to the string edit distance matching with moves [14], it is the first algorithm for any string edit distance matching problem with nontrivial alignment. The swap operation mentioned above is mostly limited to the adjacent characters, and how to improve the computation speed based on these extended edit operations is the major research question.

For block operation, Tomkins examined string edit block edit distance, in which two strings are compared by extracting collections of substrings and place them into correspondence[5], and showed that several variants to solve this problem is NP-complete. [15] proposed an efficient algorithm for sequence comparison with block reversals. Gregor also introduce a distance measure which extends the edit distance by block transpositions as constant cost edit operation, and used this measure as an evaluation criterion in machine translation [6], but this algorithm can only do swap operations between any arbitrary consecutive blocks, and have no optimizations according to common substring blocks.

The algorithm proposed in this paper has no limits for only consecutive symbols or blocks swap. Thereby, the transforming cost is the lowest for any two given strings compared with existing edit distance related algorithms.

6. Conclusions and future work

In this paper, we propose an extended edit distance algorithm which permits insertions, deletions, and substitutions at character level, and also permits block swap operations. Furthermore, we iterate extract the most possible substring blocks between two strings, and then, only block breaking points are reserved to improve the computation speed. Experimental results show that our algorithm can do string transformation more reasonable and efficient. Several research directions to extend and improve this work in the future include:

(1) Solve the ambiguous block extraction problem, and add block analysis phase between every string pair, for string transformation, block analyzer should determine whether block swap operation is needed or not.

(2) Find a way to decrease the time complexity of current algorithm.

7. Acknowledgements

This work has been supported by the open project of Key Labs of Data Engineering and Knowledge Engineering, Ministry of Education (Project Name: Study on the Knowledge platform construction on the Web). The helpful comments from anonymous reviewers are also gratefully acknowledged.

References

[1] D. Lopresti, G.W.: 'A fast technique for comparing graph representations with applications to performance evaluation', *International Journal on Document Analysis and Recognition*, 2003, 6, (4), pp. 219-229

[2] Wei, J.: 'Markov Edit Distance', *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2004, 26, (3), pp. 311-321

[3] Andres Marzal, E.V.: 'Computation of Normalized Edit Distance and Applications', *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1993, 15, (9), pp. 926-932

[4] Arslan, A.N.: 'An algorithm for string edit distance allowing substring reversals'. *Proc. Sixth IEEE Symposium on BioInformatics and BioEngineering*, 2006, Washington D.C., USA, 2006-10 2006

[5] Tomkins, D.L.A.: 'Block Edit Models for Approximate String Matching', *Theoretical Computer Science*, 1997, 181, (1), pp. 159-179

[6] Gregor Leusch, N.U., Hermann Ney: ' A novel string-to-string distance measure with applications to machine translation evaluation' . *Proc. Machine Translation Summit IX*, New Orleans, Louisiana, USA, 2003-9-23 2003 pp. Pages

[7] Gusfield, D.: 'Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology' (*USA: Cambridge University Press*, 1999. 1999)

[8] Levenshtein, V.I.: 'Binary Codes Capable of Correcting Deletions, Insertions and Reversals', *Cybernetics and Control Theory*, 1966, (10), pp. 707-710

[9] PEnrique Vidal, A.M., Pablo Aibar: 'Fast Computation of Normalized Edit Distances', *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1995, 17, (9), pp. 899-902

[10] Li Yujian, P.B.: 'A Normalized Levenshtein Distance Metric', *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2007, 29, (6), pp. 1091-1095

[11] Robert A. Wagner, R.L.: 'An Extension of the String-to-String Correction Problem', *Journal of the ACM*, 1975, 22, (2), pp. 177-183

[12] Amihood Amir, Y.A., Gad M. Landau, Moshe Lewenstein, Noa Lewenstein: 'Pattern Matching with Swaps', *Journal of Algorithms*, 2000, 37, (2), pp. 247-266

[13] A. Amir, E.E., E. Porat: ' Swap and Mismatch Edit Distance', *Algorithmica*, 2006, 45, (1), pp. 109-120

[14] Graham Cormode, S.M.: 'The string edit distance matching problem with moves', *ACM Transactions on Algorithms*, 2007, 3, (1)

[15] S. Muthukrishnan, S.C.S.: 'An Efficient Algorithm for Sequence Comparison with Block Reversals', *Theoretical Computer Science*, 2004, 321, (1), pp. 95-101