A Survey on the Algorithms of the Edit Distance Problem, the Genome Rearrangement Problem and Related Variants[*]

Shian-Liang Lin[a], Chiou-Ting Tseng[b] and Chang-Biau Yang[a†]

[a]Department of Computer Science and Engineering
National Sun Yat-sen University, Kaohsiung, Taiwan
[b]Air Navigation and Weather Services, Civil Aeronautics Administration
Ministry of Transportation and Communications, Taiwan

Abstract

The edit distance problem has been studied for several decades. Given sequences (strings) A and B with length m and n, respectively, $m \leq n$, the edit distance problem is to find the minimum cost of operations required to transform A into B. According to different models of cost functions, operations and input sequences, the problem has several variants. The edit distance on run-length encoding strings and cyclic strings are the variants on the input aspect. The edit distance considering consecutive insertions and deletions is a variant on the cost function. The block edit problem is a variant on the operation aspect. Besides, the genome rearrangement problem can also be viewed as a variant, whose operations include inversions, reversals and transpositions. In this paper, we survey some algorithms for the edit distance problem, its variants and the genome rearrangement problem.

1 Introduction

The sequence similarity has been studied for several decades and many algorithms have been developed for various applications. For example, in biological area, proteins or genomes can be represented by a sequence, and the similarities of sequences can be viewed as the relations between proteins or genomes. In 1970, Needleman and Wunsh [49] first proposed the concept of *sequence similarity* computation of two amino acid sequences, and they presented a primitive algorithm with $O(m^2n)$ time for solving the problem.

Following the same concept, in 1974, Seller [52, 53] presented an improved algorithm with $O(mn)$ time. In the same year, Wangner and Fisher [62] defined a simple version of the *edit distance* problem, including three operations: character *insertion*, *deletion* and *replacement*. They used the *dynamic programming* (DP) approach to solve the problem with $O(mn)$ time, where m and n denote the lengths of two input strings (sequences). Based on the DP approach, many algorithms and variants of this problem have been proposed later. Lowrance and Wagner [42] added a new operation, character exchange, to this problem. Their algorithm is still of $O(mn)$ time. Masek and Peterson [45] proposed an $O(n^2/\log n)$-time algorithm with the four Russians' technique.

With mapping to the *shortest edit script* (SES) problem on the edit graph, equivalent to the *longest common subsequence* (LCS) problem, the diagonal method with $O(nd)$ time was proposed by Myers [48] and $O(np)$ time algorithm by Wu *et al.* [68], where d denotes the edit distance of the two input sequences and the value of p is about a half of d. On the other hand, the variants of the edit distance problem have been studied, such as edit distance for *cyclic strings* [43, 44], edit distance for *run length encoded* (RLE) strings [4, 6, 12, 35, 39] and *block edit distance* [3, 21, 41, 47, 54, 55, 60].

The edit distance originally defined by Wangner and Fisher [62] only considers the cost on single character operations. For example, the cost of two consecutive deletions is twice of a single deletion. The edit distances with considering consecutive insertions/deletions have also been studied by several researchers [20, 24, 46, 57, 64, 65].

Furthermore, in the genome rearrangement problem, the operations are performed on a seg-

[*]This research work was partially supported by the Ministry of Science and Technology of Taiwan under contract MOST 104-2221-E-110-018-MY3.

[†]Corresponding author (Chang-Biau Yang). E-mail: cbyang@cse.nsysu.edu.tw .

ment of sequence (substrings), including reversal (reversing the substring), inverse (reversing the substring, and then substituting each character by its complement in DNA), transposition (exchanging two consecutive substrings). Since the problem with overlapping operations is NP-hard, some approximation algorithms were proposed [7, 17, 33, 63]. Then, with the non-overlapping restriction on operations, some polynomial-time algorithms have also been designed [30, 51, 59].

In this paper, we survey several papers discussing the edit distance problem and the genome rearrangement problem. We use some simple examples to explain the key points or main ideas in these algorithms. Besides, we discuss the evolution of these algorithms, and analyze the difference of these algorithms.

The rest of this paper is organized as follows. In Section 2, we introduce the background knowledge and list the time complexities of the algorithms for a summary of the surveyed papers. In Section 3, we survey some algorithms of the edit distance problem and its variants. In Section 4, we survey some genome rearrangement algorithms with operations performed on a substring. In Section 5, we give the conclusion of this paper.

2 Preliminaries

2.1 Longest Common Subsequence

Given two sequences (strings) $A = a_1a_2a_3\cdots a_m$ and $B = b_1b_2b_3\cdots b_n$, the *longest common subsequence* (LCS) problem (of two sequences) is that of finding the longest common part of A and B by deleting zero or more characters from A and B. For example, suppose that we are given $A =$ `acaagc` and $B =$`atcagtc`. The the answer of the LCS is `acagc`, whose length is 5.

The LCS problem was first presented by Needleman and Wunsch in 1970 [49]. Their purpose is to solve the alignment of biological sequences, composed of DNA, RNA or amino acids of proteins. They proposed a brute-force method to solve the alignment problem with $O(mn(m+n))$ time.

The well-known *dynamic programming* (DP) formula for solving the LCS problem was proposed by Hirschberg in 1975. He rewrote the DP method for the edit distance problem, proposed by Wagner and Fischer [62], in Equation 1 [29], where $M[i, j]$ denotes the LCS length of $A_{1..i}$ and $B_{1..j}$. Here,

	-	a	t	c	a	g	t	c
-	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1
c	0	1	1	2	2	2	2	2
a	0	1	1	2	3	3	3	3
a	0	1	1	2	3	3	3	3
g	0	1	1	2	3	4	4	4
c	0	1	1	2	3	4	4	5

Figure 1: The DP lattice for LCS with $A =$ `acaagc` and $B =$`atcagtc`, where the LCS answer is `acagc`.

$A_{i..j}$ denotes the substring of A from position indices i to j.

$$M[i,j] = \max \begin{cases} 0 & \text{if} \quad i=0 \quad \text{or} \quad j=0\ ; \\ M[i-1,j-1]+1 & \text{if} \quad a_i = b_j; \\ M[i-1,j] & \\ M[i,j-1] & \text{if} \quad a_i \neq b_j; \end{cases} \tag{1}$$

For example, the DP lattice for LCS with $A =$ `acaagc` and $B =$`atcagtc` is shown in Figure 1. Obviously, the time complexity is $O(mn)$.

2.2 Edit Distance

The *edit distance* problem is to find a series of edit operations with the minimum cost to transform sequence (string) A into sequence B. It was first defined by Wagner and Fischer in 1974 [62]. The edit operations include character insertion, character deletion and character replacement, with cost $INS(b_j)$, $DEL(a_i)$ and $REP(a_i, b_j)$, respectively. Let $M_{wf}[i, j]$ denote the minimum cost to transform $A_{1..i}$ into $B_{1..j}$. The DP formula proposed by Wagner and Fischer [62] is presented in Equation 2.

$$M_{wf}[i,j] = \min \begin{cases} \text{if} \quad a_i = b_j : \\ \quad M_{wf}[i-1,j-1] \\ \text{if} \quad a_i \neq b_j : \\ \quad M_{wf}[i-1,j-1] + REP(a_i,b_j) \\ \quad M_{wf}[i-1,j] + DEL(a_i) \\ \quad M_{wf}[i,j-1] + INS(b_j) \end{cases} \tag{2}$$

The time complexity of the above edit distance algorithm is $O(mn)$. An example of the process for calculating the edit distance is shown in the DP lattice of Figure 2, with the cost for each character insertion, deletion and replacement being 1, 1 and 2, respectively. The LCS length can be got by Equation 3 with this cost assignment, where L denotes the LCS length and d denotes the edit distance. In the example, the LCS length $5 = \frac{6+7-3}{2}$.

$$L = \frac{m+n-d}{2}. \tag{3}$$

	-	a	t	c	a	g	t	c
-	0	1	2	3	4	5	6	7
a	1	0	1	2	3	4	5	6
c	2	1	2	1	2	3	4	5
a	3	2	3	2	1	2	3	4
a	4	3	4	3	2	3	4	5
g	5	4	5	4	3	2	3	4
c	6	5	6	5	4	3	4	3

Figure 2: The DP lattice for calculating the edit distance of A = `acaagc` and B =`atcagtc` with cost functions $DEL(a_i) = 1$, $INS(b_j) = 1$ and $REP(a_i, b_j) = 2$

	-	a	t	c	a	g	t	c
-	0	1	2	3	4	5	6	7
a	1	0	1	2	3	4	5	6
c	2	1	1	1	2	3	4	5
a	3	2	2	2	1	2	3	4
a	4	3	3	3	2	2	3	4
g	5	4	4	4	3	2	3	4
c	6	5	5	5	4	3	3	3

Figure 3: The DP lattice for calculating the edit distance of A = `acaagc` and B =`atcagtc` with cost functions $DEL(a_i) = 1$, $INS(b_j) = 1$ and $REP(a_i, b_j) = 1$.

One may define variously allowed edit operations and various cost of each operation. For example, suppose the cost for each character insertion, deletion and replacement is defined to be 1, 1 and 1, respectively. Then, the DP lattice for calculating the edit distance with the same example is shown in Figure 3.

In 1974, Sellers [52, 53] also proposed an algorithm for solving the edit distance problem in $O(mn)$ time. His algorithm is based on the concept given by Needleman and Wunsch [49]. Furthermore, Sellers gave a formal definition of *evolutionary distance*, in which the cost of each operation on different characters may be different.

Later on, the edit distance problem with character operations was extended to allow the block operation [3, 21, 41, 47, 54, 55, 60], including block move, block copy, block deletion and block reversal.

2.3 Alignment

Given two sequences (strings) A and B, the alignment is a way for presenting how to transform A into B. For example, given A = `acaagc` and B = `atcagtc`, one of the possible alignments is shown as follows.

$$\begin{array}{lcccccccc} A: & \texttt{a} & - & \texttt{c} & \texttt{a} & \texttt{a} & \texttt{g} & - & \texttt{c} \\ B: & \texttt{a} & \texttt{t} & \texttt{c} & \texttt{a} & - & \texttt{g} & \texttt{t} & \texttt{c} \end{array}$$

As one can see, the above alignment corresponds to Figures 1 and 2. Eevery two corresponding positions of A and B form an aligned pair, such as (a_1, b_1) = (`a, a`), $(-, b_2)$ = (`-, t`) and $(a_4, -)$ = (`a, -`). Here, each minus sign $-$ represents a gap in an alignment. In an aligned pair $(-, b_j)$, the gap in A represents that character b_j is inserted into A at the position. Similarly, the gap in $(a_i, -)$ represents the deletion of character a_i. The edit distance of each alignment can be easily calculated. Note that different alignments may have the same edit distance. In some biology applications, the gaps is hoped to appear consecutively, which means the minimization of biological mutations. Some variations with linear, concave, convex or general cost functions for finding the best alignment were also studied [20, 24, 46, 64, 65].

2.4 Run-length Encoding

Run-length encoding (RLE) is a simple method to compress data in a lossless way. For a string, RLE compresses the data according to the symbol and the counts of consecutive appearances. For example, A =`aaabbbcc` can be represented as $A = a^3b^3c^2$. In an RLE string, each substring formed by an identical symbol is defined as a *run*, such as a^3, b^3 and c^2 in A. The edit distance problem on RLE strings is a variant of the edit distance problem. Several algorithms for solving this problem usually apply DP based on runs, instead of individual symbols [3, 4, 6, 12, 35, 39].

2.5 Summary of the Surveyed Results

We list the time complexities of the algorithms for solving the edit distance problem and its variants in Table 1. The keyword field in the table contains the key points or techniques for describing the algorithms or problems abstractly. Most algorithms are based on the DP lattice and their time complexities $O(mn)$ depend deeply on the size of the DP lattice.

As shown in Table 1, when the character insertions or deletions are consecutive, the algorithms are still efficient. Some variants can be seen in the table, including cyclic strings, RLE strings, and block edit operations.

Table 1: The algorithms for the edit distance problem. Notations: R_i: cost of each insertion or deletion is 1, each replacement is i; $|S|$: number of candidates considered in each DP cell; s: number of alternations in the mixed cost function of concave or convex; $\alpha(\ )$: inverse Ackermann function; m_a, n_b: numbers of runs in strings A and B, respectively; p_1: number of elements on the bottom boundary of a matched block.

Year	**Author(s)**	**Time complexity**	**Keywords**
	Traditional edit distance (INS,DEL,REP)		
1974	Wagner and Fischer [62]	$O(mn)$	DP
1974	Sellers [52, 53]	$O(mn)$	DP
1975	Lowrance and Wagner [42]	$O(mn)$	DP, interchange
1980	Masek and Peterson [45]	$O(n^2/\log n)$	DP, Four Russians
1986	Myers [48]	$O(nd)$	shortest edit script, R_2
1990	Wu [68]	$O(np)$	shortest edit script, R_2
2002	Jiang *et al.* [31]	$O(mn^3)$	DP, arc, RNA structure
	Consecutive Insertions/Deletions		
1976	Waterman and Smith [65]	$O(mn^2)$	DP
1981	Smith and Waterman [57]	$O(mn^2)$	DP, local alignment
1982	Gotoh [24]	$O(mn)$	DP, linear cost function
1984	Waterman [64]	$O(mn\|S\|), \|S\| \leq n$	DP, concave cost function
1988	Miller and Myers [46]	$O(mn \log n)$	DP, concave cost function, curve line, binary search
1990	Eppstein [20]	$O(n^2 \cdot \alpha(n/s))$	DP, concave, convex and mixed functions, monotone
	Edit distance for cyclic strings		
1990	Maes [43]	$O(mn \log m)$	DP, divide and conquer
2000	Marzal and Barrachina [44]	$O(mn \log m)$	branch and bound, R_1
	Edit distance for RLE strings		
1993	Bunke and Csirik [12, 13]	$O(n_b m + m_a n)$	subdivision, DP, R_2
2002	Arbell *et al.* [6]	$O(n_b m + m_a n)$	subdivision, DP, R_1
2007	Liu *et al.* [39]	$O(\min(n_b m, m_a n))$	subdivision, DP, R_1
2008	Ann *et al.* [4]	$O(m_a n_b + p_1)$	range min/max query R_2, LCS
	Block edit distance		
2010	Ann *et al.* [3]	$O(mn)$, $O(mn \log n)$, $O(mn^2)$	block edit, cost measure, non-overlapping, DP, suffix tree

Table 2 shows the algorithms for genome rearrangement. Since the general genome rearrangement problems with overlapping operations are NP-hard, some restrictions are made, such as non-overlapping operations.

3 Edit Distance

3.1 Algorithm by Lowrance and Wagner

In 1975, Lowrance and Wagner [42] proposed an extension to the edit distance problem with one more operation, interchange, which exchanges two adjacent elements in the same sequence. Under some specific cost assignments, the problem can still be solved in $O(mn)$ time. In the same year, Wagner [61] further analyzed the complexities of other cost assignments. His conclusion is that some are NP-Complete, while some are solvable with polynomial-time algorithms. Furthermore, in 1992, Schoniger and Waterman [51] presented an extension to the edit distance problem with additional operation, inversion (inverting a substring), which can be viewed as a generalization of the interchange operation.

Table 2: The algorithms for genome rearrangement.

Year	Author(s)	Time complexity	Keywords
1992	Schoniger and Waterman [51]	$O(n^6)$	inversion, non-overlapping
1993	Keceioglu and Sankoff [33]	2-approximation $O(n^2)$	reversal, permutation, overlapping, break-point graph, NP-hard
1996	Bafna *et al.* [33]	$\frac{7}{4}$-approximation $O(n^2)$	reversal, permutation, overlapping, break-point graph, NP-hard
2000	Walter *et al.* [63]	$\frac{9}{4}$-approximation $O(n^2)$	transposition, permutation, overlapping, break-point graph, NP-hard
2016	Ta *et al.* [59]	$O(n^3)$	inversion, transposition, non-overlapping, mutation fragment
2017	Hsu *et al.* [30]	$O(n^2)$	inversion, transposition, non-overlapping, repetition, run

3.2 Time Bounds by Wong and Chandra

In 1976, Wong and Chandra [67] proved the bounds on the time complexity for the edit distance problem. Suppose that the cost of each character insertion, deletion and replacement is denoted INS, DEL and REP, respectively. The only decision is equal or unequal between two symbols from A or B. Let $v = REP/(INS + DEL)$. The lower bound of the number of required comparisons is $m(n-m)+vm^2-1/v+1$. When $v = 1$, which is equivalent to the LCS problem, the lower bound becomes mn. This is the same as the result of Aho *et al.* [2]. When $INS = DEL = REP$ and $v = 0.5$, the lower bound is $mn - m^2/2 - 1$. The upper bound for the number of comparisons is $mn - \lfloor m(1-v) \rfloor \times \lfloor m(1-v)+1 \rfloor$. When $v = 1$, which is equivalent to the LCS problem, the upper bound becomes mn. When $INS = DEL = REP$ and $v = 0.5$, the upper bound is $mn-(m^2-4m)/4$.

As a result, when $INS = DEL = REP$, it may be solved with a more efficient algorithm than that with $INS = DEL = 1$ and $REP = 2$ (equivalent to LCS).

3.3 Four Russians' Technique by Masek and Paterson

In 1980, Masek and Paterson [45] proposed an improved algorithm for solving the edit distance problem. The algorithm applies the four Russians' technique to split the lattice matrix into several $k \times k$ submatrices and to store some of the *step* values to speed up the computation. The *step*, computed according to Theorem 1, means the difference between two adjacent cells in the same row or column.

Theorem 1. *[45] Let M_{mp} be the edit lattice matrix. The step (the difference between two adjacent entries) can be computed by*

$$M_{mp}[i,j] - M_{mp}[i-1,j] = \min \begin{cases} REP(a_i,b_j) \\ \quad - (M_{mp}[i-1,j] - M_{mp}[i-1,j-1]), \\ DEL(a_i), \\ INS(b_j) \\ \quad + (M_{mp}[i,j-1] - M_{mp}[i-1,j-1]) \\ \quad - (M_{mp}[i-1,j] - M_{mp}[i-1,j-1]), \end{cases}$$

$$M_{mp}[i,j] - M_{mp}[i,j-1] = \min \begin{cases} REP(a_i,b_j) \\ \quad - (M_{mp}[i,j-1] - M_{mp}[i-1,j-1]), \\ DEL(a_i) \\ \quad + (M_{mp}[i-1,j] - M_{mp}[i-1,j-1]) \\ \quad - (M_{mp}[i,j-1] - M_{mp}[i-1,j-1]), \\ INS(b_j), \end{cases}$$

where INS, DEL and REP denote the cost of each character insertion, deletion and replacement, respectively.

For example, consider A = `acaagc` and B = `atcagtc` with $k = 4$. Figure 4 shows the concept of computation process. Only the cells on the lower boundary and the right boundary of each $k \times k$ submatrix are calculated. And the values of one submatrix boundary can be obtained from its left submatrix and upper submatrix by the table lookup scheme with the two corresponding substrings as the searching index.

To reduce the amount of precomputed lookup tables, the step concept (Theorem 1) is applied to build the step table Sp, as shown in Figure 5. The left of each cell records $M_{mp}[i,j] - M_{mp}[i-1,j]$ and the right records $M_{mp}[i,j] - M_{mp}[i,j-1]$. For example, $Sp[1,4]$ stores the value $M_{mp}[1,4] - M_{mp}[0,4] = -1$, and $Sp[4,2]$ stores $M_{mp}[4,2] - M_{mp}[4,1] = 0$. The edit distance of each cell on

	-	a	t	c	a	g	t	c	ϕ
-	0	1	2	3	4	5	6	7	8
a	1				3				7
c	2				2				6
a	3				1				5
a	4	3	4	3	2	3	4	5	6
g	5				3				5
c	6				4				4
ϕ	7				5				3
ϕ	8	7	8	7	6	5	6	5	4

Figure 4: An example of edit matrix M_{mp} for computing the edit distance, where $A =$ `acaagc`, $B =$ `atcagtc`, $k = 4$ and ϕ denotes the dummy character to make m and n be multiples of k. Here, $DEL(a_i) = 1$, $INS(b_j) = 1$ and $REP(a_i, b_j) = 2$.

the rightmost column and the bottom row can be reconstruced with the step table.

For a submatrix, given the two substrings of length k with steps in the leftmost column and top row, the algorithm builds the resulting steps of the rightmost column and the bottom row. All possible $k \times k$ submatrices can be precomputed and the resulting steps are stored. As a result, after $O(|\Sigma|^k k^2 \log k)$-time preprocessing, the algorithm needs only $O((m/k) \times (n/k) \times (\log n))$ time to split the $m \times n$ edit matrix into mn/k^2 submatrices and needs $O(k + \log n)$ time to fetch the precomputed steps to compute the edit distance. In addition to the preprocessing time $O(|\Sigma|^k k^2 \log k)$, the algorithm requires $O(mn/\log n)$ time when $k = \lfloor \log n \rfloor$, and requires $O(n)$ time when $k > m$.

3.4 The Diagonal Method by Myers

In 1986, Myers [48] proposed a *diagonal method* for solving the edit distance problem with time complexity $O(nd)$, where d denotes the edit distance between the two input sequences and $m \le n$. Here, the costs of each insertion, deletion and replacement are assumed to be 1, 1, and 2, respectively. It is very efficient if the distance is very small, that is, the two input sequences are very similar.

The main concept is to calculate the furthest contours of distance $0, 1, 2, \cdots, d$, sequentially. On each diagonal line k, consisting of all cells (i, j) in the DP lattice with $k = j - i$, the furthest cell achieving distance d', $0 \le d' \le d$, is maintained.

An example of the calculated lattice is shown in Table 3. For round 0, the cells with distance 0 are computed. For round 1, the cells of distance 0 are extended to compute distance 1. The furthest contour with distance 1 consists of $(3, 4)$ on diagonal line 1 and $(4, 3)$ on diagonal line -1. For round 2, only $(3, 4)$ and $(4, 3)$ are extended to compute distance 2. Some cells with distance 2 are not calculated, such as $(2, 0)$, $(3, 1)$ and $(4, 2)$. For round 3, when diagonal -1 is to be extended, there are two possible starting cells, $(6, 5)$ from $(6, 4)$ on diagonal -2, or $(5, 4)$ from $(4, 4)$ on diagonal 0. $(6, 5)$ is selected as the starting cell, since $(6, 5)$ is further than $(5, 4)$.

Since the minimum distance on diagonal k is $|k|$, the range of diagonals required to be searched is $\{-d, -d + 1, \cdots, d - 1, d\}$. When the calculation process touches cell (m, n), the algorithm terminates and the edit distance is obtained.

Table 3: The calculation lattice for Myers' algorithm with $A =$ `cadaadaccb` and $B =$`cdaddcabccbd`. Here, the numbers with underlines are the furthest (lowest right) cells on each diagonal k with the same distance and the cells with Italic and bold are really traced in the algorithm.

		0	1	2	3	4	5	6	7	8	9	10	11	12
		-	*c*	*d*	*a*	*d*	*d*	*c*	*a*	*b*	*c*	*c*	*b*	*d*
0	-	$\boldsymbol{0}$	1	2	3	4	5							
1	*c*	1	$\underline{\boldsymbol{0}}$	$\boldsymbol{1}$	2	3	4	5						
2	*a*	2	$\boldsymbol{1}$	2	$\boldsymbol{1}$	2	3	4	5					
3	*d*	3	2	$\boldsymbol{1}$	2	$\underline{\boldsymbol{1}}$	$\underline{\boldsymbol{2}}$	$\boldsymbol{3}$	4	5				
4	*a*	4	3	2	$\underline{\boldsymbol{1}}$	$\underline{\boldsymbol{2}}$	$\underline{\boldsymbol{3}}$	4	$\underline{\boldsymbol{3}}$	$\underline{\boldsymbol{4}}$	$\underline{\boldsymbol{5}}$			
5	*a*	5	4	3	$\boldsymbol{2}$	3	4	5	$\underline{\boldsymbol{4}}$	$\underline{\boldsymbol{5}}$				
6	*d*	6	5	4	3	$\underline{\boldsymbol{2}}$	$\underline{\boldsymbol{3}}$	$\boldsymbol{4}$	5	6				
7	*a*		6	5	4	$\underline{\boldsymbol{3}}$	$\boldsymbol{4}$	5	$\underline{\boldsymbol{4}}$	$\boldsymbol{5}$	6			
8	*c*			6	5	$\underline{\boldsymbol{4}}$	5	$\underline{\boldsymbol{4}}$	$\underline{\boldsymbol{5}}$	6	$\boldsymbol{5}$	6		
9	*c*				6	$\underline{\boldsymbol{5}}$	6	$\underline{\boldsymbol{5}}$	$\boldsymbol{6}$		6	$\boldsymbol{5}$	6	
10	*b*					$\underline{\boldsymbol{6}}$		$\underline{\boldsymbol{6}}$		$\underline{\boldsymbol{6}}$		$\underline{\boldsymbol{6}}$	$\underline{\boldsymbol{5}}$	$\underline{\boldsymbol{6}}$

3.5 The Diagonal Method by Wu *et al.*

In 1990, Wu *et al.* [68] proposed another diagonal method with time complexity $O(np)$, where p is the number of deletions in the edit operations. Their algorithm is nearly twice as fast as Myers' algorithm [48].

Table 4 shows the p values in the calculation lattice of Wu *et al.* with the same inputs of Table 3. Let $\triangle$ denote the diagonal passing through cell (m, n), where $\triangle = n - m$ and it is assumed $m \le n$. Wu *et al.* found the following equality

$$d = \triangle + 2p. \tag{4}$$

	-	a	t	c	a	g	t	c	ϕ
-	-,-	-,1	-,1	-,1	-,1	-,1	-,1	-,1	-,1
a	1,-				-1,-				-1,-
c	1,-				-1,-				-1,-
a	1,-				-1,-				-1,-
a	1,-	-,-1	-,1	-,-1	1,-1	-,1	-,1	-,1	1,1
g	1,-				1,-				-1,-
c	1,-				1,-				-1,-
ϕ	1,-				1,-				-1,-
ϕ	1,-	-,-1	-,1	-,-1	1,-1	-,-1	-,1	-,-1	1,-1

Figure 5: The step table Sp, where in each cell, the left is $M_{mp}[i,j] - M_{mp}[i-1,j]$ and the right is $M_{mp}[i,j] - M_{mp}[i,j-1]$. The symbol '$-$' means that it is not calculated.

Table 4: The p values (numbers of deletions) for the algorithm of Wu *et al.* with $A =$ `cadaadaccb` and $B =$ `cdaddcabccbd`. Here, the numbers with underline are the furthest (lowest right) cells on each diagonal with the same p value and the cells with Italic and bold are really traced in the algorithm.

		0	1	2	3	4	5	6	7	8	9	10	11	12
		-	c	d	a	d	d	c	a	b	c	c	b	d
0	-	***0***	0	0	1	2								
1	c	1	$\underline{\textbf{\textit{0}}}$	***0***	0	1	2							
2	a	2	***1***	1	***0***	0	1	2						
3	d		2	***1***	1	$\underline{\textbf{\textit{0}}}$	$\underline{\textbf{\textit{0}}}$	***1***	2					
4	a			2	$\underline{\textbf{\textit{1}}}$	$\underline{\textbf{\textit{1}}}$	$\underline{\textbf{\textit{1}}}$	1	$\underline{\textbf{\textit{1}}}$	$\underline{\textbf{\textit{2}}}$				
5	a				***2***	2	2	2	$\underline{\textbf{\textit{1}}}$	$\underline{\textbf{\textit{2}}}$				
6	d					$\underline{\textbf{\textit{2}}}$	$\underline{\textbf{\textit{2}}}$	***2***	2	2				
7	a								$\underline{\textbf{\textit{2}}}$	***2***	2			
8	c										***2***	2		
9	c											***2***	2	
10	b												$\underline{\textbf{\textit{2}}}$	$\underline{\textbf{\textit{2}}}$

For example, in Table 4, $p = 2$ and $\triangle = n - m = 12 - 10 = 2$. In Table 3, $d = 6$. Therfore, $d = \triangle + 2p = 6 = 2 + 2 \times 2$. In other words, d can be calculated from p.

When the algorithm is executed, only diagonals $\{-p, -p+1, \cdots, \triangle, \triangle+1, \cdots, \triangle+p\}$ (totally $\triangle + 2p+1$ diagonals) are searched, instead of $\{-d, -d+1, \cdots, d-1, d\}$ (totally $2d+1$ diagonals). Thus, the required time of Wu *et al.* is about a half of of Myers asymptotically.

The main spirit of the algorithm of Wu *et al.* is to count only character deletions (p values), not character insertions. The DP formula for the algorithm of Wu *et al.* can be rewritten as follows when (i, j) is on diagonal $k = j - i \leq \triangle - 1$.

$$M_{wu}[i,j] = \min \begin{cases} M_{wu}[i-1,j-1] & \text{if } a_i = b_j, \\ M_{wu}[i,j-1], & \text{// insertion} \\ M_{wu}[i-1,j]+1. & \text{// deletion} \end{cases} \tag{5}$$

When (i, j) is on diagonal $k = j - i \geq \triangle$, the DP formula is rewritten as follows.

$$M_{wu}[i,j] = \min \begin{cases} M_{wu}[i-1,j-1] \ \text{ if } a_i = b_j, \\ M_{wu}[i,j-1]+1, \ \text{// insertion,} \\ \quad \text{// but need one more deletion} \\ M_{wu}[i-1,j]. \ \text{// deletion,} \\ \quad \text{// number of deletion has} \\ \quad \text{// been counted on insertion} \end{cases} \tag{6}$$

Applying the concept of Myers' algorithm, the algorithm incrementally test the p value by finding the furthest reaching cell on diagonal k. For round p, diagonals $-p, -p+1, \cdots, \triangle - 1$ are first updated sequentially. Then diagonals $\triangle + p, \triangle + p - 1, \cdots, \triangle$ are updated sequentially. The algorithm stops when cell (m, n) is reached.

See Table 4. For example, in round $p = 2$, the starting cell on diagonal -2 is $(5, 3)$. Then, it can be extended to $(6, 4)$ on the same diagonal. However it cannot be extended any more. Thus, it is directed to $(6, 5)$ on diagonal -1, and then to $(6, 6)$, $(7, 7)$ on diagonal 0. Finally, it reaches the furthest cell $(10, 11)$ on diagonal 1.

3.6 Consecutive Insertions and Deletions by Waterman *et al.*

In 1976, Waterman *et al.* [65] presented a more general form for edit distance. They not only considered the number of used operations but also the state of alignment. For example, given $A =$ `aacc` and $B =$ `ac`, the edit distance of the alignment of $A =$ `aacc` to $B =$ `-ac-` is different from that of $A =$ `aacc` to $B =$ `a--c`. The former alignment

	-	a	c
-	0	1	2
a	1	0	1
a	2	1	1
c	3	2	1
c	4	3	2

(a)

	-	a	c
-	0	1	1.1
a	1	0	1
a	1.1	1	1
c	2.1	1.1	1
c	2.2	2.1	1.1

(b)

Figure 6: An example of the DP lattice M_{wa} for the edit distance with two different cost functions, where A = aacc and B = ac. (a) The lattice that the cost of each single insertion, deletion or replacement is 1. (b) The lattice that the cost of each single insertion, deletion or replacement is 1, and the cost of each double-insertion or double-deletion is 1.1.

contains two deletions of length 1, while the latter involves only a deletion of length 2.

They considered the length of consecutive insertions/deletions as a factor of cost. For example, see Figures 6. In the first cost function, the cost of each single insertion, deletion or replacement is 1; in the second cost function, the cost of each single insertion, deletion or replacement is 1, and the cost of each double-insertion (two consecutive insertions) or each double-deletion (two consecutive deletions) is 1.1.

Different cost functions of insertions/deletions with different lengths may be more accurate to get the desired alignment. If we want to get more consecutive insertions/deletions, we can decrease the cost per insertion/deletion as the length increases. The DP formula for considering consecutive insertions/deletions is given in Equation 7 [65], where $\delta(k)$ is the cost of a consecutive insertion/deletion of length k.

$$M_{wa}[i,j] = \min \begin{cases} M_{wa}[i-1,j-1] + REP(a_i,b_j) \\ Ins[i,j] \\ Del[i,j], \end{cases}$$

where

$$Ins[i,j] = \min_{1\le k\le j}\{M_{wa}[i,j-k] + \delta(k)\},$$
$$Del[i,j] = \min_{1\le k\le i}\{M_{wa}[i-k,j] + \delta(k)\}. \tag{7}$$

The time complexity of the above algorithm is $O(mn^2)$, since each cell should considers $O(n)$ cases of consecutive insertions and consecutive deletions, and the size of the DP lattice is $O(mn)$.

3.7 Relation between Edit Distance and Similarity by Smith *et al.*

In 1981, Smith *et al.* [56–58] formally defined the equation for the similarity and distance of two given sequences. An alignment Λ can be represented by recording the aligned pairs. For example, suppose $A = a_1a_2\dots a_8$ and $B = b_1b_2\dots b_6$. An alignment $\Lambda = \{(a_1,b_1),(a_2,b_4),(a_6,b_5),(a_7,b_6)\}$ mean that $A_{3..5}$, a_8 and $B_{2..3}$ are unmatched. The unmatched substrings are called *gaps* in an alignment. The unmatched substrings of A are called deletions and the unmatched substrings of B are called insertions.

Let $s(a_i,b_j)$ be the score of aligning a_i with b_j, $REP(a_i,b_j)$ be the distance between a_i and b_j, $\delta(k)$ be the cost for a consecutive insertion/deletion of length k. The total score of an alignment Λ is the sum of $s(a_i,b_j)$, $(a_i,b_j)\in\Lambda$, minus the sum of the gap penalties. The similarity between two given sequences is the maximum score among all possible alignments. On the other hand, the distance measure of an alignment was proposed by Sellers [52] and generalized by Waterman *et al.* [65]. The cost of an alignment Λ is the sum of $REP(a_i,b_j)$, $(a_i,b_j)\in\Lambda$, plus the sum of the gap penalties. The distance between two given sequences is the minimum cost among all possible alignments.

Smith *et al.* analyzed the relation of the distance measured by Sellers [53] and the similarity measured by Needleman and Wunsch [49]. Let $\delta_{Sellers}(k)$ and $\delta_{Needleman}(k)$ be the gap penalty with length k by Sellers and Needleman *et al.*, respectively. If we set $\delta_{Sellers}(k) = \delta_{Needleman}(k) + k/2$, the two measurements become equivalent. They also proposed an algorithm for the *maximum similarity segment*, also called *local alignment*. Instead of finding the similarity of the two whole given sequences (strings), this problem tries to find the maximum similarity pair of substrings in the two given sequences. Their algorithm is given in Equation 8 [57].

$$M_{sw1}[i,0] = 0$$
$$M_{sw1}[0,j] = 0$$
$$M_{sw1}[i,j] = \max \begin{cases} M_{sw1}[i-1,j-1] + s(a_i,b_j), \\ \max_{1\le k<i}\{M_{sw1}[i-k,j] - \delta(k)\}, \\ \max_{1\le k<j}\{M_{sw1}[i,j-k] - \delta(k)\}, \\ 0. \end{cases} \tag{8}$$

3.8 Consecutive Insertions and Deletions with the Linear Cost Function by Gotoh

In 1982, Gotoh [24] presented an algorithm for calculating the edit distance in $O(mn)$ time with a linear cost function $\delta(k) = \alpha+\beta k$, where $\alpha, \beta \geq 0$, for a consecutive insertion/deletion of length k. Observing the original DP formula of Waterman *et al.* [65] in Equation 7 with $\delta(k)$, Gotoh presented a more efficient way to calculate the cost of consecutive insertions as follows.

$$\begin{aligned} Ins[i,j] &= \min_{1\leq k\leq j}\{M_{wa}[i,j-k]+\delta(k)\} \\ &= \min\{M_{wa}[i,j-1]+\delta(1), \\ &\quad \min_{2\leq k\leq j}\{M_{wa}[i,j-k]+\delta(k)\}\} \\ &= \min\{M_{wa}[i,j-1]+\delta(1), \\ &\quad \min_{1\leq k'\leq j-1}\{M_{wa}[i,j-1-k']+\delta(k')\}+\beta\} \\ &= \min\{M_{wa}[i,j-1]+\delta(1), Ins[i,j-1] \\ &\quad +\beta\}. \end{aligned} \tag{9}$$

In other words, each $Ins[i,j]$ needs only to check two possible candidates, $M_{wa}[i,j-1]+\delta(1)$ and $Ins[i,j-1]+\beta$, instead of the original $\min_{1\leq k\leq j}\{M_{wa}[i,j-k]+\delta(k)\}$. The formula for the deletion case can be derived similarly.

Based upon the above observation, Gotoh gave a DP formula, shown in Equation 10 [24], for the linear cost function. As a result, with the linear cost function for consecutive insertions and consecutive deletions, the time complexity can be reduced to $O(mn)$.

$$M_{go}[i,j] = \min\begin{cases} M_{go}[i-1,j-1]+REP(a_i,b_j) \\ Ins[i,j] \\ Del[i,j], \end{cases}$$

where

$$\begin{aligned} Ins[i,j] &= \min\{M_{go}[i,j-1]+\delta(1), Ins[i,j-1]+\beta\}, \\ Del[i,j] &= \min\{M_{go}[i-1,j]+\delta(1), Del[i-1,j]+\beta\}. \end{aligned} \tag{10}$$

3.9 Consecutive Insertions and Deletions with the Concave Cost Function by Waterman

In 1984, Waterman presented an algorithm for edit distance by considering consecutive insertions and deletions with the concave cost function [64]. A *concave* function, such as $\delta(k) = \alpha+\beta\log(k), \alpha, \beta > 0$, satisfies the general inequality

$$\delta(\ell_a+\ell_b) \leq \delta(\ell_a)+\delta(\ell_b), \quad \ell_a, \ell_b \geq 1, \tag{11}$$

or equivalently,

$$\delta((1-\alpha)x+\alpha y) \geq (1-\alpha)\delta(x)+\alpha\delta(y), \quad 0 \leq \alpha \leq 1. \tag{12}$$

The inequality in Equation 11 shows that the cost of a consecutive insertion/deletion with length $\ell_a+\ell_b$ is less than or equal to that of two consecutive insertions/deletions with lengths ℓ_a and ℓ_b. By combining the two inequalities in Equation 11 with $\delta(\ell_a+\ell_b+\ell_c)$ and $\delta(\ell_a+\ell_c)$, Waterman got the inequality

$$\delta(\ell_a+\ell_b+\ell_c)-\delta(\ell_a+\ell_c) \leq \delta(\ell_a+\ell_b)-\delta(\ell_a), \quad \ell_a, \ell_b, \ell_c \geq 1. \tag{13}$$

Equation 13 is the key of Waterman's algorithm. First, Waterman analyzed the insertion case of the general formula in Equation 7

$$Ins[i,j] = \min_{0\leq k<j}\{M_{wa}[i,k]+\delta(j-k)\}. \tag{14}$$

Assume $Ins[i,j] = M_{wa}[i,l]+\delta(j-l)$ for some $0 \leq l < j$ has the minimum value, we have

$$M_{wa}[i,l]+\delta(j-l) \leq M_{wa}[i,k]+\delta(j-k), \quad 0 \leq k < j. \tag{15}$$

If $l \leq k$, Equation 13 can be applied by letting $\ell_a = j-k, \ell_b = 1, \ell_c = k-l$, we get

$$\delta(j-l+1)-\delta(j-l) \leq \delta(j-k+1)-\delta(j-k), \quad 0 < l \leq k < j. \tag{16}$$

Combining Equations 15 with 16, we have

$$M_{wa}[i,l]+\delta(j-l+1) \leq M_{wa}[i,k]+\delta(j-k+1), \quad 0 < l \leq k < j. \tag{17}$$

Thus,

$$\begin{aligned} &Ins[i,j+1] = \min\{M_{wa}[i,j]+\delta(1), \\ &\min_{0\leq k'\leq l}\{M_{wa}[i,k']+\delta(j+1-k')\}\} \end{aligned} \tag{18}$$

In other words, $M_{wa}[i,k], k > l$, can be ignored since it is dominated by $M_{wa}[i,l]$ when computing $Ins[i,j+1]$. When $Ins[i,j+2], Ins[i,j+3], \cdots, Ins[i,n]$ are computed, it is still true that $M_{wa}[i,k] \leq M_{wa}[i,l]$ for $k > l$. This concept is illustrated in Figure 7.

Waterman constructed the candidate set $S_{Ins}(i) = \{l|Ins[i,l+1] = M_{wa}[i,l]+\delta(1)\}$ to record all positions of $M_{wa}[i,l]$ used in row i. The deletion case can be derived similarly. The DP formula is shown in Equation 19 [64] .

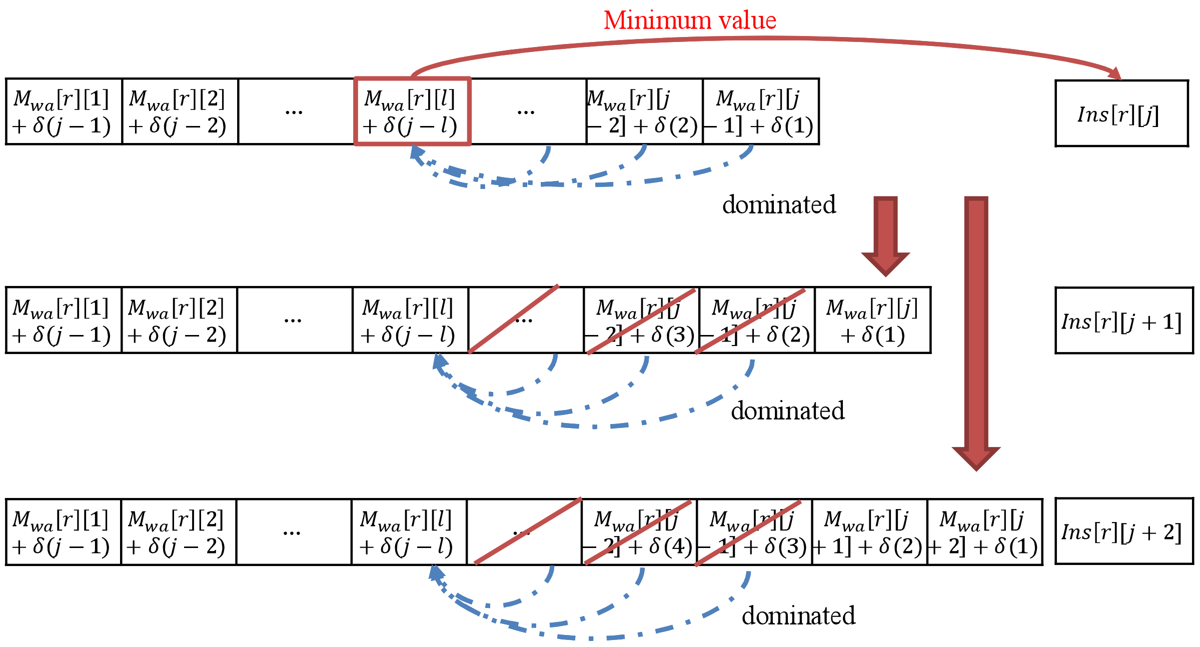

Minimum value
$M_{wa}[r][1] + \delta(j-1)$
$M_{wa}[r][2] + \delta(j-2)$
...
$M_{wa}[r][l] + \delta(j-l)$
$M_{wa}[r][j-2] + \delta(2)$
$M_{wa}[r][j-1] + \delta(1)$
$Ins[r][j]$
dominated
$M_{wa}[r][j-2] + \delta(3)$
$M_{wa}[r][j-1] + \delta(2)$
$M_{wa}[r][j] + \delta(1)$
$Ins[r][j+1]$
dominated
$M_{wa}[r][j-2] + \delta(4)$
$M_{wa}[r][j-1] + \delta(3)$
$M_{wa}[r][j+1] + \delta(2)$
$M_{wa}[r][j+2] + \delta(1)$
$Ins[r][j+2]$
dominated

Figure 7: An illustration of computing $Ins[r, j]$ with the concave function in Waterman's algorithm. Assume that $M_{wa}[r, l]+\delta(j-l)$ is the value selected for computing $Ins[r, j]$. When computing $Ins[r, j+1]$ and $Ins[r, j+2]$, we have $M_{wa}[r, l] \leq M_{wa}[r, k]$ for $l < k \leq j$. This situation can be extended to the computation of $Ins[i, n]$.

$$M_{wa}[i, j] = \min \begin{cases} M_{wa}[i-1, j-1] + REP(a_i, b_j) \\ Ins[i, j] \\ Del[i, j], \end{cases}$$

where

$$\begin{aligned} &Ins[i, j] = \min\{M_{wa}[i, j-k] + \delta(j-k), \\ &k \in S_{Ins}(i)\}, \\ &Del[i, j] = \min\{M_{wa}[i-k, j] + \delta(j-k), \\ &k \in S_{Del}(j)\}. \end{aligned} \tag{19}$$

The time complexity depends on the sizes of $S_{Ins}(i)$ and $S_{Del}(j)$. Waterman conjectured that this size does not grow faster than $\log n$. In summary, Waterman's algorithm reduces the number of candidates for consecutive insertions/deletions in each cell of the DP lattice for concave cost functions. The time complexity is $O(|S(i)|mn)$, where $|S(i)|$ is the maximum size of all candidate sets $S_{Ins}(i)$ and $S_{Del}(j)$, and it was conjectured that $|S(i)| = O(\log n)$ [64].

3.10 Consecutive Insertions and Deletions with the Concave Cost Function by Miller and Myers

Based on the candidate set of Waterman [64], in 1988, Miller and Myers [46] presented two edit distance algorithms for consecutive insertions/deletions with the concave cost function. The size of candidate set $S_{Ins}(i)$ and $S_{Del}(j)$ in Waterman's algorithm [64] may become larger and larger, when calculating $Ins[i, j]$ and $Del[i, j]$ in the same row or column. In other words, once a candidate cell (i, j) is put into the candidate set $S_{Ins}(i)$ or $S_{Del}(j)$, the candidate will never be eliminated.

The candidate set $S_{Ins}(i)$ can be arranged into a decreasing list $S_{Ins}(i, x)$, where i is the row index, x is the index of the candidate list. For example, suppose $S_{Ins}(7) = \langle 2, 4, 5 \rangle$. Then, $S_{Ins}(7, 1) = 2$, $S_{Ins}(7, 2) = 4$ and $S_{Ins}(7, 3) = 5$. Note that $S_{Ins}(i, 1)$ is always the minimum candidate of $S_{Ins}(i)$ according to the property presented by Waterman [64], as shown in Figure 7.

Here, we present the case of consecutive insertions to illustrate their algorithm, because the deletion case is similar to the insertion case. Their algorithm eliminates the dominated candidates in the candidate list by the idea of *p-curve*. The *p*-curve consists of all values calculated from $M_{mm}[i, p]$ in row i, as shown in Figure 8.

The example in Figure 8 shows that all possible candidates for calculating $Ins[i, 3]$ are the in-

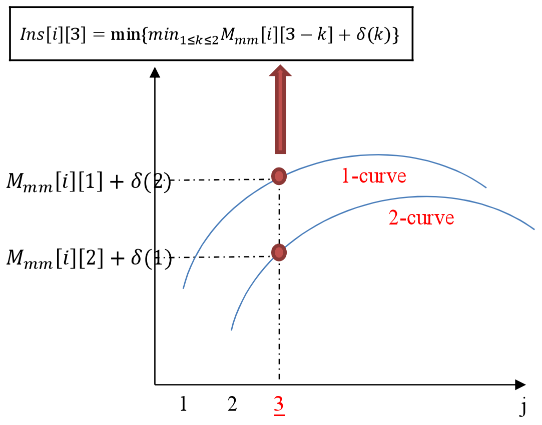

$Ins[i][3] = \min\{min_{1\le k\le 2} M_{mm}[i][3-k] + \delta(k)\}$
$M_{mm}[i][1] + \delta(2)$
$M_{mm}[i][2] + \delta(1)$
1-curve
2-curve
1
2
3
j

Figure 8: An example of p-curve used to calculate $Ins[i,3]$. The p-curve consists of $M_{mm}[i,p] + \delta(k), 1 \le k \le n-p$. The candidates for calculating $Ins[i,3]$ will choose one from 1-curve and one from 2-curve.

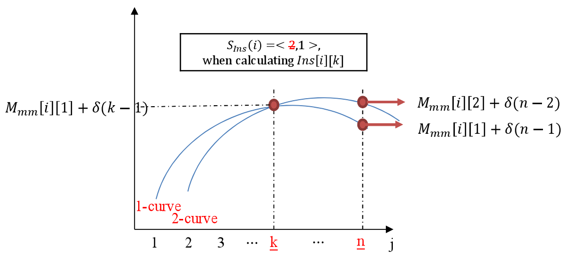

$S_{Ins}(i) =< \cancel{2},1 >$,
when calculating $Ins[i][k]$
$M_{mm}[i][1] + \delta(k-1)$
$M_{mm}[i][2] + \delta(n-2)$
$M_{mm}[i][1] + \delta(n-1)$
1-curve
2-curve
1
2
3
…
k
…
n
j

Figure 9: An example for eliminating the candidates of p-curve. $S_{Ins}(r) =< 2, 1 >$. Suppose that $M_{mm}[i,1]$ is the minimum candidate for calculating $Ins[i,k]$. In other words, $M_{mm}[i,2] + \delta(n-2) \ge M_{mm}[i,1] + \delta(n-1)$. So the value from $M_{mm}[i,1]$ (1-curve) should be smaller than the value from $M_{mm}[i,2]$ (2-curve) when calculating $Ins[i,j], k < j \le n$.

tersections between the p-curves and the vertical line $j = 3$. The candidate list records the p-curves which will be used in the future. These curves have two properties. First, all curves have the same shape, since they are of the same concave function. Second, the number of intersections between two curves is at most 1.

Figure 9 illustrates how to eliminate these dominated candidates. Once the minimum candidate for calculating $Ins[i,k]$ is determined to be $M_{mm}[i,p]$, we can eliminate the candidates $M_{mm}[i,q]$, where $M_{mm}[i,q] + \delta(n-q) \ge M_{mm}[i,p] + \delta(n-p)$. As shown in Figure 9, 1-curve is always lower than 2-curve after vertical line k. Thus, 2-curve is dominated and it can be eliminated after $Ins[i,k]$ has been calculated.

Furthermore, the algorithm records the lifetime of each candidate in the candidate list. To calculate the lifetime, each candidate needs to find the nearest intersection with other curves in the future. Since the binary search is invoked to find the intersection of two curves, the time required for the candidate list is $O(\log n)$ in each cell. Thus, the total time complexity of their algorithm is $O(mn \log n)$.

3.11 Consecutive Insertions and Deletions with the Mixed Convex and Concave Cost Functions by Eppstein

In 1990, Eppstein [20] considered consecutive insertions/deletions with a mixed convex and concave cost function, composed of interleaving convex and concave segments. For example, a mixed cost function $\delta(k)$ can be split into several segments with index $c_i, 1 \le i \le s$, where $\delta_1(k)$ is convex (concave in his paper) when $0 < k \le c_1$, $\delta_2(k)$ is concave (convex in his paper) when $c_1 < k \le c_2$, $\delta_3(k)$ is convex (concave in his paper) when $c_2 < k \le c_3$ and so on.

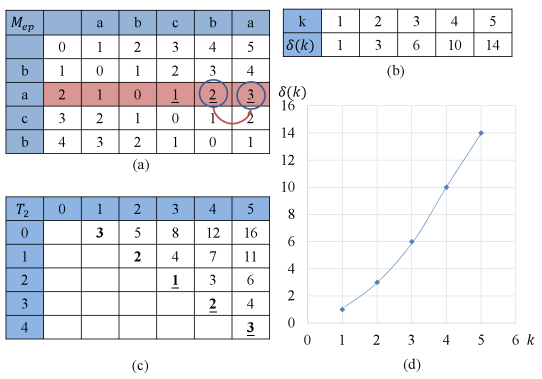
A concave function satisfies the *quadrangle inequality*,

$$w(i,j) + w(i',j') \ge w(i',j) + w(i,j'), \quad i \le i' \le j \le j'. \tag{20}$$

In the above inequality, $w(i,j) = \delta(j-i)$ denotes the cost of a consecutive insertion/deletion with length $j-i$. Similarly, a convex function satisfies the inverse quadrangle inequality by replacing $\ge$ with $\le$ in Equation 20.

It should be noted that there is some inconsistency in the definition of a *concave* function in the previous studies. The definition of a concave function by Waterman [64] follows the standard mathematical definition, that is, $f(x+y) \le f(x)+f(y)$. Miller and Myers [46] also follows this definition. However, starting from Yao [69], she interchanged the definitions of a concave function and a convex function (She called the inequality in Equation 20 a *convex quadrangle inequality*.). Then, some subsequent studies follow the definition of Yao, such as Wilber [66], Galil and Giancarlo [22], Klawe and Kleitman [36], Eppstein [20].

Eppstein's algorithm utilizes Wilber's algorithm [66] to deal with the convex (concave in the original paper) segments, and uses Klawe and Kleitman's algorithm [36] to deal with the concave



M_{ep}		a	b	c	b	a
	0	1	2	3	4	5
b	1	0	1	2	3	4
a	2	1	0	1	2	3
c	3	2	1	0	1	2
b	4	3	2	1	0	1

(a)

k	1	2	3	4	5
$\delta(k)$	1	3	6	10	14

(b)

T_2	0	1	2	3	4	5
0		**3**	5	8	12	16
1			**2**	4	7	11
2				**1**	3	6
3					**2**	4
4						**3**

(c)

(d)

Figure 10: An example for computing the insertion case with the convex cost function by Wilber's algorithm [66]. (a) The DP lattice M_{ep}. (b) A convex cost function $\delta(k)$, where k is the length of a consecutive insertion. (c) The matrix T_2 for calculating $M_{ep}[2][j]$. (d) The curve of $\delta(k)$ in (b).

(convex in the original paper) segments. Wilber's algorithm [66] can find the minimum of each column of an $n \times n$ matrix in $O(n)$ time when the cells in the matrix satisfy the convex quadrangle inequality (convex segment).

An example of the insertion case is shown in Figure 10. $\delta(k)$ shown in Figure 10 (b) is a convex segment, whose visual curve is shown in Figure 10 (d).

In Figure 10 (c), each column j of T_2 records all possible costs of consecutive insertions at $M_{ep}[2][j]$. For example, $M_{ep}[2,5]$ is obtained from $T_2[4,5] = M_{ep}[2,4] + \delta(5-4) = 2+1$, which is the minimum of column 5 of T_2. It means that the distance at $M_{ep}[2,5]$ is formed by $M_{ep}[2,4]$ plus one consecutive insertion of length 1, i.e. $\delta(1)$. Another possible source for $M_{ep}[2,5]$ is $= M_{ep}[2,3] + \delta(5-3) = 1+3 = 4$, which means $M_{ep}[2,3]$ plus one consecutive insertion of length 2, i.e. $\delta(2)$, but it is not the minimum.

T_2 is a matrix satisfying the convex quadrangle inequality, such as $T_2[1][3] + T_2[2][5] \leq T_2[2][3] + T_2[1][5]$. Since T_2 satisfies the convex quadrangle inequality, Wilber's algorithm calculates the minimum of column j, for all j, in $O(n)$ time. That is, the calculation of $M_{ep}[2][j]$ can be done in $O(n)$ time.

Klawe and Kleitman's algorithm can calculate the minimum of all candidates in the concave case with a similar way as Wilber's algorithm in $O(n\alpha(n))$ time, where $\alpha(\ )$ is the inverse Ackermann function, because both of their algorithms use the algorithm presented by Aggarwal *et al.* [1].

Eppstein's algorithm is shown in Equation 21 [20], whose time complexity is $O(n^2 s\alpha(n/s))$.

$$M_{ep}[i,j] = \min \begin{cases} M_{ep}[i-1,j-1] + REP(a_i,b_j) \\ Ins[i,j] \\ Del[i,j], \end{cases}$$

where

$$\begin{aligned} Ins[i,j] &= \min_{1\leq p\leq s} Ins_p[i,j], \\ Ins_p[i,j] &= \min\{M_{ep}[i,j-k] + \delta_p(j-k), 1 \leq k \leq j\}, \\ Del[i,j] &= \min_{1\leq p\leq s} \{Del_p[i,j]\}, \\ Del_p[i,j] &= \min\{M_{ep}[i-k,j] + \delta_p(i-k), 1 \leq k \leq i\}. \end{aligned} \tag{21}$$

In 1989, Galil and Giancarlo [22] presented two algorithms for consecutive insertions/deletions with the concave and convex cost functions. Both of their algorithms run in $O(nm \log n)$ or $O(n^2)$ time when the cost function satisfies the *closest zero property*. Their algorithm for the concave function is similar to the algorithm presented by Miller and Myers [46].

3.12 Cyclic Strings by Maes

In 1990, Maes [43] proposed the *cyclic string-to-string correction problem*, a variant of the edit distance problem, which requires to transform the rotations of A into B. A *k-rotation* of A is to remove its prefix with length k and to concatenate the removed prefix to the end. For example, `tatgagatca` is a 4-rotation of $A =$ `atcatatgag`.

The naïve method is to find the edit distance of all k-rotations, $0 \leq k \leq m-1$, of A versus B by applying the algorithm of Wagner and Fischer [62] m times, whose total time complexity is $O(m^2 n)$.

The algorithm of Maes constructs the edit lattice of AA , the concatenation of A with itself, versus B. Maes found that the minimal cost edit paths between AA and B in the lattice may not cross, but may intersect. So the search area is limited, instead of mn cells. The algorithm is shown in Algorithm 1. As the search area in each loop in line 6 is a subdivision of the area between the minimal cost path from $(0,0)$ to (m,n) and the minimal cost path from $(m,0)$ to $(2m,n)$ (inclusive), the total time required for each loop in line 3 is sum up to $O(mn)$. So the total time complexity is $O(mn \log m)$ and the required space is $O(mn)$.

3.13 Cyclic Strings with Divide-and-conquer by Marzal and Barrachina

In 2000, Marzal and Barrachina [44] proposed an improved algorithm over the algorithm of Maes

Algorithm 1 The algorithm for the cyclic string-to-string correction problem by Maes [43].

```
1: $q = \lceil \log m \rceil$
2: Find the minimal cost paths from $(0,0)$ to $(m,n)$ and from $(m,0)$ to $(2m,n)$
3: for $i = q-1$ down to 0 do
4:     $j = 2^i$
5:     while $j < m$ do
6:         Find the minimal cost path from $(j,0)$ to $(j+m,n)$ between the path from $(j-2^i,0)$ to $(j-2^i+m,n)$ and the path from $(\min(j+2^i,m),0)$ to $(\min(j+2^i,m)+m,n)$
7:         $j = j + 2^{i+1}$
8:     end while
9: end for
```

[43] and the algorithm of Gregor and Thomason [25]. Their algorithm is based on the divide and conquer strategy of Maes and applies a branch and bound strategy inspired from Gregor and Thomason [25]. The cost function is restricted that the cost of each insertion, deletion or replacement is 1.

The algorithm of Marzal and Barrachina calculates all minimal edit distance paths for $\{M_{mb}[0,0]$ to $M_{mb}[m,n]$, $M_{mb}[1,0]$ to $M_{mb}[m+1,n]$, $\cdots$, $M_{mb}[m-1,0]$ to $M_{mb}[2m-1,n]$, $M_{mb}[m,0]$ to $M_{mb}[2m,n]\}$, where M_{mb} denotes the DP lattice of AA and B. The algorithm utilizes the lower bound $g(i,j)$ of all edit paths starting between $M_{mb}[i,0]$ and $M_{mb}[j,0]$ to eliminate the calculation of some edit paths.

Let $\sigma^k(A)$ denote the k-rotation of A. The edit distance $d(\sigma^k(A),B)$ is the minimum edit path from $M_{mb}[k,0]$ to $M_{mb}[k+m,n]$. For example, suppose $m = 16$. The algorithm calculates $d_{min} = \min\{d(\sigma^0(A),B)$, $d(\sigma^8(A),B)$ and $d(\sigma^{16}(A),B)\}$, which divides the interval $[0,16]$ into $[0,8]$ and $[8,16]$. Recursively apply to $[0,8]$ if $g(0,8) < d_{min}$ and $[8,16]$ if $g(8,16) < d_{min}$. No edit distance $d(\sigma^k(A),B)$, $k \in [i,j]$, will be the minimum edit distance, if the lower bound $g(i,j) \geq d_{min}$, where d_{min} means the minimum edit distance of all computed $d(\sigma^k(A),B)$ before. The lower bound formulas are shown in Theorems 2 and 3. The time complexity of their algorithm is $O(mn\log m)$, which is the same as Maes' algorithm.

Theorem 2. *[44] The lower bound of $d(\sigma^k(A),B)$ for all $k \in [i,j]$, $0 \leq i < j \leq m$, is given by*

$$g_1(i,j) = \max(0, \lceil \tfrac{d(\sigma^i(A),B)+d(\sigma^j(A),B)}{2} \rceil - (j-i)) \leq d(\sigma^k(A),B).$$

Theorem 3. *[44] The lower bound of $d(\sigma^k(A),B)$ for all $k \in [i,j]$, $0 \leq i < j \leq m$ is given by*

$$g_2(i,j) = \max(g_1\{i,j\}, \min_{i<k'<j} \rho_{k'}(A,B)) \leq d(\sigma^k(A),B),$$

where $\rho_{k'}(A,B) = \min_{0 \leq q \leq n}(\rho(A_{1\cdots k'}, B_{q+1\cdots n}))+(\rho(A_{k'+1\cdots m}, B_{1\cdots q})))$, $\rho(A,B) = \max(m,n) - |\Sigma|$ and $|\Sigma|$ is the alphabet set of A and B.

3.14 Run-length Encoding Strings by Bunke and Csirik

In 1993, Bunke and Csirik [12] proposed an algorithm for calculating the edit distance on *run-length encoding* (RLE) strings (sequences). Suppose that two RLE strings A and B are of lengths m_a and n_b, respectively, and the lengths of the extracted plain text are m and n, respectively. For example, if $A =$ `aaaacccccbb`, then the RLE string is encoded as $A = \mathtt{a}^4\mathtt{c}^5\mathtt{b}^2$, $m_a = 3$ and $m = 11$.

A pair of matched or unmatched symbols in the RLE format represents a block in the plain text. The DP lattice is divided into two kinds of blocks. A *dark block* corresponds to a matched pair and a *light block* corresponds to an unmatched pair. In each block, only the cells on the right and bottom boundaries need to be calculated. Each cell can be calculated in constant time. Thus, the total time complexity is $O(n_b m + m_a n)$ if the lengths of all runs in both sequences are the same.

In 1995, Bunke and Csirik proposed an improved algorithm [13] for the same problem. The problem is restricted on the cost function that the costs of each insertion, deletion and replacement are 1, 1, and 2, respectively. There is no restriction on the length of each run.

Table 5 shows an example for the block division of the DP lattice. As examples, the dark block $D_{1,1}$ corresponds to the matched pair of $\mathtt{a}^3$ in A and $\mathtt{a}^2$ in B, the light block $D_{2,1}$ corresponds to the unmatched pair of $\mathtt{c}^3$ in A and the run $\mathtt{a}^2$ in B. Only the cells on the bottom row and rightmost column of each block needs to be calculated. An example is shown in Table 6. By Lemmas 1 and 2, the computation time of each cell is constant. Thus, the total time complexity is $O(n_b m + m_a n)$.

Lemma 1. *[13] Let x be a symbol, and A and B be two strings. Then $d(Ax^k, Bx^k) = d(A,B)$ for any $k \geq 0$.*

Lemma 2. *[13] Let x and y, $x \neq y$ be two symbols, and A and B be two strings. Then $d(Ax^k, By^h) = min\{d(Ax^k,B)+h, d(A,By^h)+k\}$ for any $k,h \geq 0$.*

Table 5: The block division of the DP lattice for RLE strings with $A = \mathtt{a}^3\mathtt{c}^3\mathtt{b}^2$ and $B = \mathtt{a}^2\mathtt{c}^2$ by Bunke and Csirik [13].

	-	a a	c c
-	$D_{0,0}$	$D_{0,1}$	$D_{0,2}$
a a a	$D_{1,0}$	$D_{1,1}$	$D_{1,2}$
c c c	$D_{2,0}$	$D_{2,1}$	$D_{2,2}$
b b	$D_{3,0}$	$D_{3,1}$	$D_{3,2}$

Table 6: The DP lattice for $A = \mathtt{a}^3\mathtt{c}^3\mathtt{b}^2$ and $B = \mathtt{a}^2\mathtt{c}^2$ by Bunke and Csirik [13].

	-	a	a	c	c
-	0	1	2	3	4
a	1		1		3
a	2		0		2
a	3	2	1	2	3
c	4		2		4
c	5		3		5
c	6	5	4	5	6
b	7		5		7
b	8	7	6	7	8

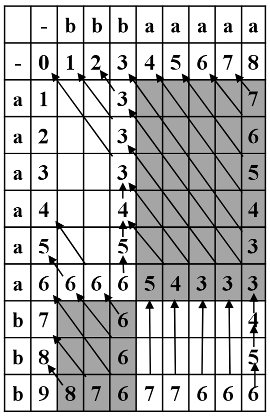


Figure 11: The DP lattice for RLE strings $A = \mathtt{a}^6\mathtt{b}^3$ and $B = \mathtt{b}^3\mathtt{a}^5$ by Arbell *et al.* [6], where the arrow lines mean where the values of the cells come from (not unique).

3.15 Run-length Encoding Strings by Arbell *et al.*

In 2002, Arbell *et al.* [6] proposed an algorithm for calculating the edit distance of two *run-length encoding* RLE strings (sequences). In their algorithm, on the cost of each insertion, deletion or substitution is assumed to be 1.

The DP lattice of two RLE strings is divided into several blocks, as shown in the example of Figure 11. In a dark block, the value of each cell $M_{ar}[i, j]$ is equal to $M_{ar}[i-1, j-1]$. In a light block, the top row and the leftmost column are separated into three zones (zone I, zone II, zone III) according to the position of the calculated cell and the form of block (horizontal block or vertical block), as shown in the example of Figure 12.

The algorithm chooses the minimum of each cell from two possible candidates, one from zone I and the other from zone II, because the value from zone III must be larger. For a horizontal block

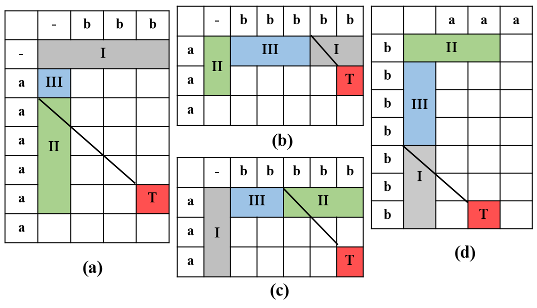


Figure 12: The zones in a light block of the DP lattice for RLE strings by Arbell *et al.* [6]. (a) An example for the rightmost column. (b) Another example for the rightmost column. (c) An example for the bottom row. (d) Another example for the bottom row.

Table 7: An example for computing the edit distance between an RLE string and an uncompressed string by Liu *et al.* [39], where $A = \mathtt{a}^6\mathtt{b}^3$ and $B = \mathtt{bbbaaaaa}$.

	-	b	b	b	a	a	a	a	a
-	0	1	2	3	4	5	6	7	8
a^6	6	6	6	6	5	5	5	5	5
b^3	9	8	7	8	8	9	9	8	8

, the calculation of $M_{ar}[i, j]$ in the rightmost column considers only two positions: $M_{ar}[i-1, j]$ and $M_{ar}[i_{top}, j_{diagonal}]$ (the intersection point of zone I and the diagonal line through $M_{ar}[i, j]$) in zone I. Other cases can be done similarly.

The calculation of each cell in both dark and light blocks can be done in $O(1)$ time, and the number of calculated cells is $O(n_b m + m_a n)$. Thus, the total time complexity of the algorithm is $O(n_b m + m_a n)$.

3.16 A Run-length Encoding String and an Uncompressed String by Liu *et al.*

In 2007, Liu *et al.* [39] proposed an algorithm for the edit distance between an RLE string (sequence) A and an uncompressed string (sequence) B. Here, the cost of each insertion, deletion or replacement is 1.

In the DP lattice $D_{li}[i, j]$, the index i represents A_i, the ith run in A. An example in shown in Table 7. Note that $D_{li}[i, j]$ is different from $M_{li}[i, j]$. For example, $D_{li}[1, 1] = M_{li}[6, 1]$ is the edit distance of the first run of A ($\mathtt{a}^6$) and $B = \mathtt{b}$, while $M_{li}[1, 1]$ is the edit distance of $A = \mathtt{a}$ and $B = \mathtt{b}$. The main idea is to check how the value of $D_{li}[i, j]$ is calculated from $D_{li}[i, j-1]$ with Lemma 3.

Lemma 3. *[60]* $M_{li}[i, j] - M_{li}[i, j-1] \in \{-1, 0, 1\}$.

The DP formula for calculating $D_{li}[i, j]$ is shown as follows [39].

$$D_{li}[i, j] = \min_{0 \le u \le j} \{D_{li}[i-1, u] + \varepsilon D_{li}[A_i, B_{u+1 \cdots j}]\}. \tag{22}$$

Table 8: The tables used in the algorithm of Liu *et al.* [39], where $T_{A^-}[T_A[u]] = u$, $T_{B^-}[T_B[u]] = u$ and $T_{C^-}[T_C[u]] = u$.

Name	**Value**	**Restriction**
$T_A[u]$	$D_{li}[i-1, u] + B_{1 \cdots u}(\tau)$	$\max\{j - \lvert A_i \rvert, 0\} \le u \le j$
$T_B[u]$	$D_{li}[i-1, u] + B_{1 \cdots u}(\tau) - u$	$0 \le u \le j - \lvert A_i \rvert$ and $B_{1 \cdots u}(\tau) \le \lvert A_i \rvert$
$T_C[u]$	$D_{li}[i-1, u] - u$	$0 \le u \le j$ and $\lvert A_i \rvert \le B_{1 \cdots u}(\tau)$
$T_{A^-}[v_1]$	$\max\{u \mid T_A[u] = v_1$ for $0 \le u \le j\}$	$-n \le v_1 - T_A[0] \le 2n$
$T_{B^-}[v_2]$	$\max\{u \mid T_B[u] = v_2$ for $0 \le u \le j - \lvert A_i \rvert\}$	$-2n \le v_2 - T_B[0] \le n$
$T_{C^-}[v_3]$	$\min\{u \mid T_C[u] = v_3$ for $0 \le u \le j\}$	$-2n \le v_3 - T_C[0] \le 0$

$$\begin{aligned} \varepsilon D_{li}[A_i, B_{u \cdots j}] &= max\{|A_i|, j-u\} - \\ &\quad min\{|A_i|, B_{u+1 \cdots j}(\tau)\} \\ &= \begin{cases} |A_i| - B_{u+1 \cdots j}(\tau) \\ \quad \text{if } |A_i| \ge j-u, \\ j-u-B_{u+1 \cdots j}(\tau) \\ \quad \text{if } B_{u+1 \cdots j}(\tau) \le |A_i| \le j-u, \\ j-u-|A_i| \\ \quad \text{if } |A_i| \le B_{u+1 \cdots j}(\tau). \end{cases} \end{aligned} \tag{23}$$

Combing Lemma 3 with Equation 22, $D_{li}[i, j]$ comes from three possible candidates $D_{li}[i, j-1] - 1$, $D_{li}[i, j-1]$ or $D_{li}[i, j-1] + 1$. Accordingly, their algorithm uses three tables T_A, T_B, and T_C to store the value of left-hand side of the three equations in Equation 22. To make their algorithm more efficient, they also build three inverted tables T_{A^-}, T_{B^-} and T_{C^-} for T_A, T_B, and T_C. The definition and restriction of these tables are shown in Table 8. The time complexity of the algorithm is $O(m_a n)$.

3.17 LCS of Run-length Encoding Strings by Liu *et al.* and Ann *et al.*

In 2008, Liu *et al.* also proposed an algorithm for calculating the LCS between an RLE string and an uncompressed string [40]. The time complexity is $\mathrm{O}(m_a n)$.

As mentioned in Section 2.2, the LCS problem is equivalent to calculate the edit distance with costs of insertion, deletion and replacement being 1, 1, and 2, respectively. The conversion formula is $L = \frac{m+n-d}{2}$, as shown in Equation 3.

In 2008, Ann *et al.* [4] presented an algorithm

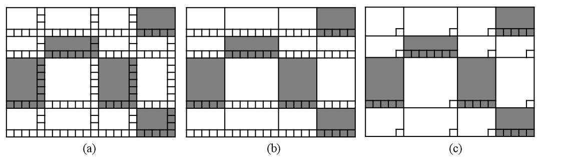


Figure 13: The comparison of three algorithms for calculating the LCS of RLE strings by Ann *et al.* [4], where only small cells are required to be calculated. (a) $O(n_b m + m_a n)$ by Bunke and Csirik [13]. (b) $O(m_a n)$ by Liu *et al.* [40]. (c) $O(n_b m_a + p_1)$ by Ann *et al.* [4], where p_1 denotes the number of cells in the bottom boundaries of all black blocks.

for calculating the LCS of two RLE strings [4] with $O(n_b m_a + \min\{p_1, p_2\})$ time, where p_1 and p_2 denote the number of cells in the bottom and right boundaries of all black blocks. They also illustrated the comparison of the time complexities of three algorithms, as shown in Figure 13.

In 2012, Ann *et al.* [5] presented an algorithm for computing the constrained LCS of RLE strings with $O(m_a n_b r + r \times \min\{q_1, q_2\} + q_3)$, where r denotes the length of constrained sequence P, q_1 and q_2 denote the numbers of cells in the south and east faces of cuboids blocks on the first layer of the DP lattice, and q_3 denotes the number of face cells of black cuboids of the DP lattice. Note that a black cuboid corresponds to a strong match, meaning that three runs in A, B and P have the same symbol.

3.18 The Block Edit Problem by Ann *et al.*

In 2010, Ann *et al.* [3] proposed an algorithm for the block edit problem. The block edit problem not only involves the character-edit operations, but also the block-edit operations. Three problems $P(EIS, C)$, $P(EI, L)$ and $P(EI, N)$ are presented with some restrictions that the block operations cannot overlap and they should be performed from left to right on the parts which have not been edited before.

These edit problems are formulated as $P(o, c)$, where o is the block copy operation and c is the cost measurement. The definition of allowed operations and cost measurement are given in Table 9. An example of the operations is shown in Figure 14. For the problem $P(EIS, C)$, the block operations include block deletion, internal block copy, external block copy and shift block copy, whose

Table 9: The allowed block operations and cost measurements by Ann *et al.* [3], with the example shown in Figure 14.

Block Operations		
Name	**Description**	**Example**
External Copy (E)	Copy a substring of X and insert it into a valid position of the active part of W_i.	$W_1 \to W_2$
Internal Copy (I)	Copy a substring of the inactive part of W_i and insert it into a valid position of the active part of W_i.	$W_4 \to Y$
Shift Copy (S)	Copy a shifted string, which is a substring of X or a substring of the inactive part of W_i, and insert it into a valid position of the active part of W_i.	$W_3 \to W_4$ $W_2 \to W_3$
Deletion	Delete a valid substring from the active part of W_i.	$X \to W_1$
Cost Measurement		
Name	**Description**	
Constant cost (C)	Costs of all block operations with different lengths are the same.	
Linear cost (L)	Cost of one block operation is $p_s + ip_e$, where p_s and p_e are constants and i is the length of the copied or deleted substring.	
Nested cost(N)	Cost of block deletions is constant. Cost of a block copy is $p_{copy} + d_{ch}(s_1, s_2)$ where s_1 is the copied string, s_2 is the string after editing, and $d_{ch}(s_1, s_2)$ is the edit distance from s_1 to s_2 with character edit operations.	

costs are constant.

The $P(EIS, C)$ problem is solved with a DP formula as shown in Figure 15. The straightforward DP algorithm requires $O(nm^2(n + m)|\Sigma|)$ time. By some processing techniques with $O(n + m^2)$ time, the time complexity can be reduced to $O(mn)$.

For more details in $P(EIS, C)$, the traditional edit distance $d_1(X_i, Y_j)$ with character edit operations can be calculated in $O(1)$ time. Block deletion distance $d_2(X_i, Y_j)$ can also be calculated in $O(1)$ time by preserving the current minimum value of $\{d(X_{k-1}, Y_j) | 0 \leq k \leq i\}$ in each iteration. To compute the costs of the external block copy d_3, internal block copy d_4, external block shifted copy d_5 and internal block shifted copy d_6, a preprocess for building a suffix tree and a minimum query structure is needed.

$P(EI, L)$ and $P(EI, N)$ problems can be solved by the DP algorithm with a similar strategy except

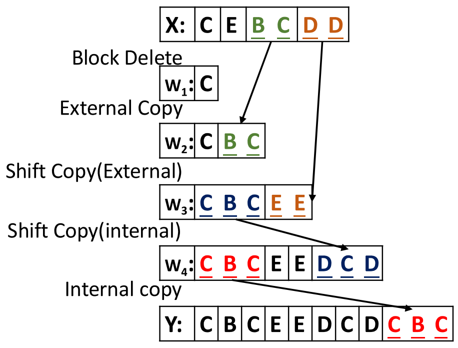


Figure 14: An example of the block edit operation by Ann *et al.* [3].

the cost calculation. As a result, $P(EI, L)$ can be solved in $O(mn \log m)$ time with $O(n + m^2)$ preprocessing time and $P(EI, N)$ can be solved in $O(nm^2)$ time with $O((n + m)m^2)$ preprocessing time.

In 2014, Peng and Yang [50] applied the incremental suffix maximum query with the set union and find technique to improve the algorithm for $P(EI, L)$, whose time complexity is further reduced to $O(nm + m^2)$.

4 Genome Rearrangement

The genome rearrangement involves a group of block edit operations, originally used to compare the genomes of different species. The operations include reversal, transposition, translocation, block move and duplication [3, 7–11, 14–19, 26–28, 32, 34, 37, 47, 54, 55, 63]. Since the problems involving overlapping operations are NP-hard, some restrictions on the operations were proposed, such as non-overlapping operations. With such restrictions, the problems become solvable with polynomial time.

4.1 Overlapping Reversals on Permutations by Keceioglu and Sankoff

In the *reversal distance on permutation* problem, the *reversal* operation is used to transform permutation π into permutation γ [33], and the operated intervals may overlap. To simplify this problem, we can regard the target permutation γ as a sorted sequence from 1 to n. A reversal $\rho(i, j)$ reverses the order of π in the interval $[i, j]$. Let π' be the result after applying a reversal $\rho(i, j)$ on

$$
\begin{aligned}
d(X_i, Y_j) &= \begin{cases} \infty, & \text{if } i < 0 \text{ or } j < 0 \\ 0, & \text{if } i = j = 0 \\ \min \left\{ \begin{array}{l} d_1(X_i, Y_j), \\ d_2(X_i, Y_j), \\ d_3(X_i, Y_j), \\ d_4(X_i, Y_j), \\ d_5(X_i, Y_j), \\ d_6(X_i, Y_j) \end{array} \right\} & \text{, otherwise} \end{cases} \\
d_1(X_i, Y_j) &= \begin{cases} d(X_{i-1}, Y_{j-1}), & \text{if } X_i = Y_j \\ \min \left\{ \begin{array}{l} d(X_i, Y_{j-1}), \\ d(X_{i-1}, Y_j) \end{array} \right\} + 1, & \text{if } X_i \neq Y_j \end{cases} \\
d_2(X_i, Y_j) &= \min\{d(X_{k-1}, Y_j) + p_{delete} | i \leq k \leq i\} \\
d_3(X_i, Y_j) &= \min\{d(X_i, Y_{k-1}) + p_{copy} | \\
&\quad Y_{k \cdots j} \text{ is a substring of } X\} \\
d_4(X_i, Y_j) &= \min\{d(X_i, Y_{k-1}) + p_{copy} | \\
&\quad Y_{k \cdots j} \text{ is a substring of } Y_{k-1}\} \\
d_5(X_i, Y_j) &= \min\{d(X_i, Y_{k-1}) + p_{copy} | \\
&\quad Y_{k \cdots j} \text{ is a shifted substring of } X\} \\
d_6(X_i, Y_j) &= \min\{d(X_i, Y_{k-1}) + p_{copy} | \\
&\quad Y_{k \cdots j} \text{ is a shifted substring of } Y_{k-1}\}
\end{aligned}
\tag{24}
$$

Figure 15: The DP formula for solving $P(EIS, C)$ by Ann *et al.* [3].

a sequence π. The formal definition is given as follows.

$$
\pi'_t = \begin{cases} \pi_{i+j-t} & \text{if } i \leq t \leq j, \\ \pi_t & \text{otherwise.} \end{cases} \tag{25}
$$

For example, if $\pi = 4213$, $\pi \cdot \rho(1, 3) = 1243$. An example for transforming from π to γ is shown in Figure 16.

The straightforward method is to apply one reversal to bring one element into its correct place. So the reversal distance is at most $n - 1$. In a permutation π, a break point is a pair of neighboring positions (i, j) such means that $|\pi_{i+1} - \pi_i| \neq 1$. In other words, π_i and π_{i+1} are not consecutively increasing or decreasing. For example, suppose that $\pi = 4213$. Then, $(1, 2)$, and $(3, 4)$ are break points. Whenever there exists a breakpoint in π, π is not sorted completely.

In 1993, Keceioglu and Sankoff [33, 34] presented a greedy approximation algorithm based on the concept of breakpoints. They apply a reversal operation between two breakpoints which can eliminate at least one breakpoint until there is no breakpoint in π. An example of their algorithm is shown in Figure 16. Their greedy algorithm is of 2-approximation, whose time complexity is $O(n^2)$ and space complexity is $O(n)$.

Keceioglu and Sankoff [33, 34] also presented a branch and bound method with linear programing for getting the exact solution. A reversal can

π:	[0	[4]	2]	[1	[3]	5]

$$\Downarrow \rho_1(1,3)$$

π':	0	1	[2	4]	[3	5]

$$\Downarrow \rho_2(3,4)$$

γ:	0	1	2	3	4	5

Figure 16: An example of the greedy algorithm for calculating the reversal distance on a permutation. The first and last elements are the pseudo boundaries 0 and $n+1$. $[\pi_i, \pi_{i+1}]$ means a breakpoint between positions i and $i+1$ in π.

remove at most two breakpoints. A series of i reversals can remove at most $i+1$ breakpoints, since only the ith reversal can removes two breakpoints. For example, given π=42315, applying a reversal $\rho(2,3)$ and then applying a reversal $\rho(1,4)$ on π can remove three breakpoints.

Keceioglu and Sankoff [33, 34] showed that a lower bound of the required reversals is $\lceil \frac{2b(\pi)}{3} - \frac{b_2(\pi)}{3} \rceil$, where $b_2(\pi)$ is the number of reversals which can remove two breakpoints at the same time and $b(\pi)$ is the number of breakpoints in π. To find a dynamic upper bound, they performed the greedy algorithm which always tries to apply a reversal to remove two breakpoints one time. Then, the exact algorithm is a branch and bound approach by eliminating some paths in the search tree with the upper bounds and lower bounds. The required time and space are $O(tL(n,n))$ and $O(n^2)$, respectively, where t is the size of the branch and bound search tree and $L(n,n)$ is the time required for solving a linear programming with n variables and n constraints.

4.2 Lower Bounds for Overlapping Reversals on Permutations by Bafna and Pevzner

Bafna and Pevzner [7] proved a tighter lower bound $\frac{2b(\pi)}{3} - \frac{c_4(\pi)}{3}$, where c_4 is the number of 4-cycles (a cycle with 4 edges) in the *breakpoint graph* [33]. They also proposed some efficient algorithms based on the breakpoint graph.

In the breakpoint graph $G(\pi)$ of a permutation π, each breakpoint contributes two edges, a solid eage and a dashed edge, as an example shown in Figure 17. A *solid edge* connects vertices π_i and π_j if $|i-j| = 1$ and $|\pi_i - \pi_j| > 1$. A *dashed edge* connects π_i and π_j if $|\pi_i - \pi_j| = 1$ and $|i-j| > 1$. For example, there is a solid edge between $\pi_3 = 1$ and $\pi_4 = 4$ and a dashed edge between $\pi_5 = 6$ and $\pi_{10} = 7$.

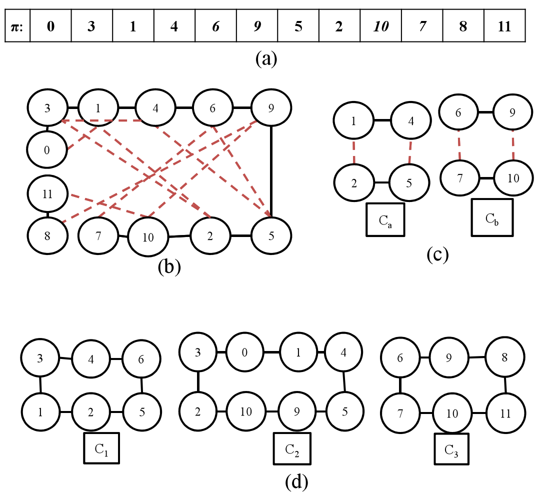


Figure 17: An example of the breakpoint graph. (a) A permutation π = `03146952(10)78(11)`, where `0` and `11` are dummies used as bounaries. (b) The breakpoint graph $G(\pi)$ with $c(\pi) = 3$ and $b(\pi) = 10$. (c) Crossing cycles C_a and C_b in $G(\pi)$. (d) A maximal cycle decomposition of $G(\pi)$.

In $G(\pi)$, the solid edge and dashed edge must be interleaved to form a cycle, such as C_a and C_b in Figure 17. Let $c(\pi)$ denote the number of cycles in the maximal cycle decomposition of $G(\pi)$. A tighter lower bound of the reversal distance $d(\pi)$ is given in Theorem 4.

Theorem 4. *[7] For any permutation π,*

$$d(\pi) \geq b(\pi) - c(\pi) \geq$$
$$b(\pi) - c_4(\pi) - (c(\pi) - c_4(\pi)) = \frac{2b(\pi)}{3} - \frac{c_4(\pi)}{3}.$$

When all the $c(\pi)$ are c_4, we have $c(\pi) = \frac{b(\pi)}{2}$. Clearly, $d(\pi) \geq \frac{b(\pi)}{2}$. A property for c_4 is described in Lemma 4. Two cycles are *crossing* means that their solid edge are interleaved. For example, in Figure 17, solid edges (1, 4), (5, 2) in C_a are interleaved with solid edges (6, 9), (10, 7) in C_b.

Lemma 4. *[7] Suppose that a 4-cycle C has no reversal which can remove at least one breakpoint in C. If C has a crossing cycle C', doing a reversal on C' will make C have a reversal removing at least one breakpoint.*

x-reversal, $x \in \{-2, -1, 0, 1, 2\}$, is a reversal which removes x breakpoints if $x \geq 0$, or adds x breakpoints if otherwise. With such an observation, Algorithm 2 sorts a signed permutation by at most $b(\pi) - \frac{c_4(\pi)}{2}$ steps with $\frac{3}{2}$ approximation ratio. Furthermore, with the properties of c_4, they presented another algorithm with $b(\pi)-\frac{c_4(\pi)}{4}$ steps and $\frac{7}{4}$ approximation ratio. Overall, the time complexities of these algorithms are $O(n^2)$.

Algorithm 2 Sorting a signed permutation π [7], where $Reversal(\pi)$ is presented in [33].

```
 1: while $\pi$ contains a breakpoints do
 2:     if $\pi$ has no decreasing strips then
 3:         if any 4-cycle $C$ remains in $G(\pi)$ then
 4:             Find a cycle $C'$ which crosses $C$.
 5:             Do a 0-reversal on $C'$.
 6:             Do a 2-reversal on the 4-cycle $C$.
 7:         else
 8:             Do a 0-reversal on an arbitrary cycle.
 9:         end if
10:     else
11:         $\rho = Reversal(\pi)$
12:         $\pi = \pi \cdot \rho$
13:     end if
14: end while
```

4.3 An Approximation Algorithm for Overlapping Transpositions on Permutations by Walter *et al.*

The *transposition distance* is defined to be the minimum number of transposition operations to transform an input sequence π into the target sequence γ [63]. A *transposition* $\tau(i,j,k)$ exchanges two substrings $\pi_{i..j-1}$ and $\pi_{j..k-1}$, $1 \le i < j < k \le n+1$. For example, Figure 18 illustrates the transposition operations. Let π' be the resulting sequence after applying a transposition $\tau(i,j,k)$ on a sequence π. The formal definition is given by Walter *et al.* as follows [63].

$$\pi'_t = \begin{cases} \pi_{t+j-i} & \text{if } i \le t < i+k-j, \\ \pi_{t-k+j} & \text{if } i+k-j \le t < k, \\ \pi_t & \text{otherwise.} \end{cases} \tag{26}$$

Walter *et al.* [63] presented an approximation algorithm with approximation ratio 2.25 for computing the transposition distance. Here, we still regard target sequence γ as a sorted sequence from 1 to n. The algorithm is also based on the breakpoint graph. We add two dummy elements $\pi_0 = 0$ to the front and $\pi_{n+1} = n+1$ to the rear of π. A breakpoint for transposition on permutation π exists between two adjacent elements π_{i-1} and π_i, when $\pi_i - \pi_{i-1} \ne 1$, $1 \le i \le n+1$. For example, given π = 41235687 as shown in Figure 18, (4, 1), (3, 5), (6, 8) and (8, 7) are the breakpoints. Note

π	0	4	1	2	3	5	6	8	7	9

$\Downarrow \tau(1,2,5)$

π'	0	1	2	3	4	5	6	8	7	9

$\Downarrow \tau(7,8,9)$

γ	0	1	2	3	4	5	6	7	8	9

Figure 18: An example of transforming permutation π = 41235687 into γ = 12345678 by transpositions. The first and last elements are the dummy boundaries $\pi_0 = 0$ and $\pi_9 = 9$.

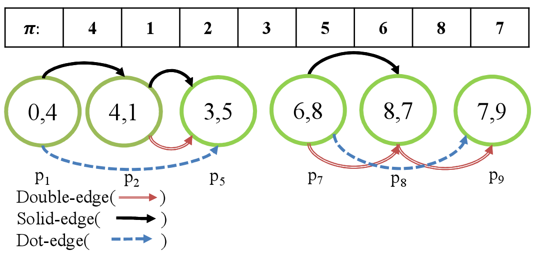


Figure 19: An example of the transposition breakpoint graph $D(\pi)$ for π = 41235687.

that (8, 7) is not a breakpoint for calculating the reversal distance.

In the transposition breakpoint graph $D(\pi)$, node p_i corresponds to a breakpoint between π_{i-1} and π_i. There are three kinds of edges, a solid-edge from p_i to p_j when $\pi_j - \pi_{i-1} = 1$, $i < j$ and $\pi_j \ne n+1$; a double-edge from p_i to p_j when $\pi_j - \pi_{i-1} = 1$, $i < j$ and $\pi_{i-1} \ne 0$; a dot-edge from p_i to p_j when $\pi_i - \pi_{j-1} = 1$ and $i < j$. An example of the transposition breakpoint graph is shown in Figure 19.

An x-transposition, $x \in \{-3,-2,-1,0,1,2,3\}$ removes x breakpoints if $x \ge 0$, or adds x breakpoints if otherwise. Note, to keep the consistency of this paper, we have reversed the definition from the original one defined by Walter *et al.* [63]. In $D(\pi)$, if there are $p_i \rightarrow p_j$, $p_j \Rightarrow p_k$ and $p_i \dashrightarrow p_k$, then there exists a 3-transposition. As the result, algorithm 3 is presented to find the transposition distance of a permutation. The approximation ratio of the algorithm is 2.25, provided by Lemma 5.

Lemma 5. *[63] Given a permutation π and its transposition breakpoint graph $D(\pi)$ with more than 5 nodes, if there is no 2-transposition nor 3-transposition, it is possible to remove at least four nodes in three transpositions.*

Algorithm 3 The approximation algorithm for finding the transposition distance of a permutation π by Walter *et al.* [63].

```
1:  Construct the transposition breakpoint graph $D(\pi)$
2:  $i := 0$
3:  while $|V| \neq 0$ do
4:      $i := i + 1$
5:      if there is a 3-transposition then
6:          $\tau_i :=$ 3-transposition
7:      else
8:          if there is a 2-transposition then
9:              $\tau_i :=$ 2-transposition
10:         else there is a 1-transposition
11:             $\tau_i :=$ 1-transposition
12:         end if
13:     end if
14:     $\pi := \pi \cdot \tau_i$
15:     Construct $D(\pi)$
16: end while
17: return $i, (\tau_1, \tau_2, \cdots, \tau_i)$
```

4.4 Upper and Lower Bounds for Reversals and Transpositions on Binary Strings by Christie and Irving

Christie and Irving [17] proved the upper and lower bounds for the reversal distance and the transposition distance between two binary strings with the concept of breakpoints. They also proved that the decision version of the reversal distance problem on two binary strings is NP-Hard, by transforming from the 3-partition problem, which is NP-complete [23].

The reversal breakpoint they defined for the reversal distance of two binary strings S and T is different from two permutations. In their new definition, a breakpoint exists in a length-2 common substring κ of S and T when the number of occurrences of κ in S is greater than the number of occurrences of κ in T. For example, if S has two substrings "00" and T has one, one of "00" in S has to be broken when transforming from S into T, which means a breakpoint in S. Furthermore, substrings "01" and "10" are counted together since reversals can convert "01" into "10" and vice versa. The number of breakpoints $b_r(S,T)$ of two binary strings S and T can be calculated by Equation 27, which calculates the occurrence difference of all length-2 substrings in S and T.

$$\begin{aligned} b_r(S,T) = \sum_{\alpha \le a < b \le \omega} \delta(f_{ab}(S) + f_{ba}(S) - f_{ab}(T) - \\ f_{ba}(T)) + \sum_{0 \le a \le 1} \delta(f_{aa}(S) - f_{aa}(T)), \end{aligned}$$

where $\delta(x) = \max\{x, 0\}$, $f_{ab}(S)$ is the number of substring "ab" in S and, α and ω are the dummy elements added to the front and the rear of S and T. (27)

Consequently, the lower bound of the reversal distance, provided in Theorem 5, can be obtained, because a reversal can remove at most two breakpoints.

Theorem 5. *[17] Given two binary strings S and T, let $d_r(S,T)$ be the reversal distance. Then,*

$$\lceil b_r(S,T)/2 \rceil \le d_r(S,T) \le \lfloor n/2 \rfloor.$$

For the transposition distance $d_t(S,T)$ of two binary strings, the number of occurrences of "01" and "10" have to be counted separately. Then the proof technique of the reversal distance can be applied to the transposition distance similarly. The lower bound and upper bound of $d_t(S,T)$ can be obtained with the number of transposition breakpoints $b_t(S,T)$.

Theorem 6. *[17] Given two binary strings S and T, we have*

$$\lceil b_t(S,T)/3 \rceil \le d_t(S,T) \le \lfloor n/2 \rfloor.$$

4.5 Non-overlapping Inversions by Schoniger and Waterman

Because the problem with overlapping reversals and transpositions is NP-hard, in 1992, Schoniger and Waterman [51] made some restrictions on the operations of the sequence alignment problem. Their operations include character insertion, deletion and replacement, and non-overlapping inversions. They presented a DP algorithm for the sequence alignment on DNA sequence with $O(n^6)$ time [51].

An inversion $\theta(i,j)$ on a sequence A is to reverse the substring $A_{i..j}$ and to replace each element of $A_{i..j}$ with its complement. Note that a DNA sequence consists of four characters a, t, c and g. Besides, a is complementary to t with each other, and c is complementary to g with each other. For example, suppose A = `taccgtca`. Then, $B = A \cdot \theta(4,7)$ = `tacgacga`. Let $A = a_1 a_2 \cdots a_n$

Table 10: An example for the algorithm of Schoniger and Waterman [51] with $A =$ `taccgtca` and $B =$ `gatcacgga`.

	-	g	a	t	c	a	c	g	g	a
-	0	0	0	0	0	0	0	0	0	0
t	0	0	0	1	0	0	0	0	0	0
a	0	0	$\boxed{1}$	0	0	1	0	0	0	1
c	0	0	0	0	1	0	2	1	0	0
c	0	0	0	0	1	0	1	2	1	0
g	0	1	0	0	0	0	0	2	3	2
t	0	0	1	1	0	0	0	1	3	3
c	0	0	1	0	2	0	1	0	2	2
a	0	0	1	1	1	3	2	1	$\boxed{3}$	3

and $B = b_1b_2 \cdots b_n$ be the resulting sequence of $\theta(i,j)$ on A. The formal transformation is given as follows.

$$b_t = \begin{cases} \overline{a_{i+j-t}} & \text{if } i \le t \le j, \\ a_t & \text{otherwise.} \end{cases} \qquad (28)$$

The operation of non-overlapping inversions means that the intervals of two inversions cannot overlap. For example, $\theta(3,5)$ and $\theta(5,7)$ have an overlap on a_5. They defined $Z(g,h;i,j)$ to represent the local alignment score of $A_{g..i}$ and $B_{h..j}$ after applying an inversion $\theta(h,j)$ on B. They defined two matching functions $d_1(a_i,b_j)$ and $d_2(a_i,b_j)$, and two gap functions $w_1(k) = \alpha_1 + \beta_1 k$ and $w_2(k) = \alpha_2 + \beta_2 k$, where k is the length of consecutive gaps. The cost function $w_1(k)$(insertion and deletion) and $d_1(a_i,b_j)$(replacement) are used to calculate $Z(g,h;i,j)$(local alignment), and $w_2(k)$ and $d_2(a_i,b_j)$ are used for the entire algorithm. The cost of one inversion is γ. Their algorithm calculates mn cells, each of which takes $O(m^2n^2)$ time to find the best inversion of $B_{h..j}$ to align with $A_{g..i}$. Thus, the total time complexity of their algorithm is $O(n^6)$ when $m = n$.

An example is shown in Table 10, where $\gamma = -1$, $w_1(k) = w_2(k) = 0 - k$, and $d_1(a_i,b_j) = d_2(a_i,b_j) = 1$ if $a_i = b_j$, otherwise -2. Let $M_{sw2}[i,j]$ denote the best alignment score of $A_{1..i}$ and $B_{1..j}$. $M_{sw2}[8,8]$ involves a score from an inversion of $A_{3..8} =$ `ccgtca` versus $B_{3..8} =$ `tcacgg`, In more detail, $M_{sw2}[8,8] = M_{sw2}[2,2]$ $+Z(3,3;8,8)+\gamma = 1+3-1 = 3$, where $Z(3,3;8,8)$ is the alignment score of $A_{3..8} =$ `ccgtca` and $\overline{B_{3..8}} =$ `ccgtga`.

4.6 Non-overlapping Inversions and Transpositions by Ta *et al.*

The operations involved in the problem solved by Schoniger and Waterman [51] are character insertions, deletions, replacements and non-overlapping inversions. Since the algorithm of Schoniger and Waterman [51] needs $O(n^6)$ time, in 2016, Ta *et al.* [59] made more restrictions on the operations, including only non-overlapping inversions and non-overlapping transpositions, but without character insertions, deletions and replacements. In this situation, the two input strings should of the same length, that is $|A| = |B|$. They presented an algorithm for solving the *non-overlapping inversion and transposition distance problem* with $O(n^3)$ time and $O(n^2)$ space [59].

For the non-overlapping meaning here, for example, if an inversion $\theta(3,5)$ (applied on interval [3,5]) is applied to sequence A, no operations can be applied on a part of interval [3,5] of A, such as inversion $\theta(1,3)$ or transposition $\tau(5,7,10)$ (swapping $A_{5..6}$ and $A_{7..9}$).

Their algorithm uses two *mutation tables* M_{ta1} and M_{ta2} as tools to calculate all possible inversions and transpositions, respectively. $M_{ta1}[i,j] = \overline{a_j}$ and $M_{ta2}[i,j] = a_j$. Table 11 shows an example of the two mutation tables. As one can see, an inversion $\theta(1,3)$ on $A_{1..3}$ is equal to $B_{1..3} =$ `gat`. $M_{ta1}[1,3]$, $M_{ta1}[2,2]$ and $M_{ta1}[3,1]$ form a slash line whose content is equal to $B_{1..3}$. An inversion $\theta(5,6)$ on $A_{5..6}$ is equal to $B_{5..6} =$ `ct`. $\tau(5,7,10)$ on $A_{5..9}$ (swapping $A_{5..6}$ and $A_{7..9}$) is equal to $B_{5..9} =$ `ctagg`. $M_{ta2}[7,5]$, $M_{ta2}[8,6]$, and $M_{ta2}[9,7]$ forms a backslash line whose content is equal to $B_{5..7} =$ `agg`, $M_{ta2}[5,8]$ and $M_{ta2}[6,9]$ forms a backslash line whose content is equal to $B_{8..9} =$ `ct`. Note that a_4 is not covered by any mutation operation because $a_4 = \mathtt{t} = \mathtt{b}_4$.

With the mutation table M_{ta1}, whether the inversion of a substring $A_{i..j}$, for all $1 \le i \le j \le n$ is equal to $B_{i..j}$ can be checked in $O(n^2)$ time. However, to check all possible transpositions with the mutation table M_{ta2} needs $O(n^3)$ time, since there are $O(n)$ pairs of slash lines may form a transposition on the same substring. Thus, the total time complexity is $O(n^3)$ and space complexity is $O(n^2)$.

In 2017, Hsu [30] proposed a more efficient algorithm to solve the same problem as Ta *et al.* solved. His algorithm is based on the *run structure* of a string, proposed by Kolpakov and Kucherov [38]. The time complexiy of his algorithm is re-

Table 11: An example of mutation tables M_{ta1} and M_{ta2}, where $A =$ `atctaggct` and $B =$ `gatactagg`.

$M_{ta1}[i, j]$

	1	2	3	4	5	6	7	8	9
1	t	a	$\underline{\mathbf{g}}$	a	t	c	c	g	a
2	t	$\underline{\mathbf{a}}$	g	a	t	c	c	g	a
3	$\underline{\mathbf{t}}$	a	g	a	t	c	c	g	a
4	t	a	g	a	t	c	c	g	a
5	t	a	g	a	t	$\underline{\mathbf{c}}$	c	g	a
6	t	a	g	a	$\underline{\mathbf{t}}$	c	c	g	a
7	t	a	g	a	t	c	c	g	a
8	t	a	g	a	t	c	c	g	a
9	t	a	g	a	t	c	c	g	a

(a)

$M_{ta2}[i, j]$

	1	2	3	4	5	6	7	8	9
1	a	t	c	t	a	g	g	c	t
2	a	t	c	t	a	g	g	c	t
3	a	t	c	t	a	g	g	c	t
4	a	t	c	t	a	g	g	c	t
5	a	t	c	t	a	g	g	$\underline{\mathbf{c}}$	t
6	a	t	c	t	a	g	g	c	$\underline{\mathbf{t}}$
7	a	t	c	t	$\underline{\mathbf{a}}$	g	g	c	t
8	a	t	c	t	a	$\underline{\mathbf{g}}$	g	c	t
9	a	t	c	t	a	g	$\underline{\mathbf{g}}$	c	t

(b)

duced to $O(n^2)$.

5 Conclusions and Future Work

The traditional edit distance problem is to find the minimum edit distance between two input sequences (strings) with the edit operations, including character insertions, deletions and replacements. We use the traditional edit distance problem as the baseline to compare with other variants in three aspects: input, allowed operations and output. First, from the input aspect, the edit distance for cyclic strings is different because one of the input strings is viewed as a set of strings by rotation. The edit distance of RLE strings can be seen as a special case that the same characters are usually consecutive. Thus, the edit distance for RLE strings and genome rearrangement are the same as the traditional problem in the input aspect.

Second, from the allowed operations aspect, the block edit distance and genome rearrangement problems are different from the traditional edit distance problem as operations can be performed on substrings in the former problems. However, the operations on cyclic strings are the same as the traditional problem.

Third, from the output aspect, all of these problems desire to find the minimum cost required for transforming a string into another string. Thus, these problems are the same in the output aspect.

With these three aspects, we give the comparison of the variants to the traditional edit distance problem, as shown in Table 12. Furthermore, the characteristics of these variants lead them to be useful in different applications. For example, the edit distance for cyclic strings can be applied to pattern recognition, because the string which represents patterns like a polygon can be viewed as the same after rotation. The edit distance for RLE strings can be applied to compare the compressed information directly. The block edit distance can simulate human editing behaviors to compare two documents. And the block edit distance can also be used to compare DNA sequences.

The genome rearrangement problem simulates the mutation of genomes and finds the relations between genomes. With the same idea, we can use different operations to simulate human behaviors or possible operations to transform or destroy the data and then to recover the original information. For example, we can simulate how humans modify articles by string copies and deletions. We could also find out how the data in some special formation, such as images and compressed files, were destroyed in transmission.

We hope that this paper is useful for those who study or do not study in this field. Especially, it is helpful to understand the key points of these algorithms. In the future, we will survey the related longest common subsequence (LCS) problem and its variants.

References

[1] A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber, "Geometric applications of a matrix-searching algorithm," *Algorithmica*, Vol. 2, No. 1, pp. 195–208, 1987.

[2] A. V. Aho, D. S. Hirschberg, and J. D. Ullman, "Bounds on the complexity of the longest common subsequence problem,"

Table 12: The variants compared to the traditional edit distance problem in three aspects.

Problem	**Input**	**Allowed operations**	**Output**
Edit distance for cyclic strings	different	same	same
Edit distance for RLE strings	same	same	same
Block edit distance	same	different	same
Genome rearrangement	same	different	same

Journal of the ACM, Vol. 23, No. 1, pp. 1–12, 1976.

[3] H. Y. Ann, C. B. Yang, Y. H. Peng, and B. C. Liaw, "Efficient algorithms for the block edit problems," *Information and Computation*, Vol. 208(3), pp. 221–229, 2010.

[4] H. Y. Ann, C. B. Yang, C.-T. Tseng, and C. Y. Hor, "A fast and simple algorithm for computing the longest common subsequence of run-length encoded strings," *Information Processing Letters*, Vol. 108, pp. 360–364, 2008.

[5] H. Y. Ann, C. B. Yang, C.-T. Tseng, and C.-Y. Hor, "Fast algorithms for computing the constrained lcs of run-length encoded strings," *Theoretical Computer Science*, Vol. 432, pp. 1–9, 2012.

[6] O. Arbell, G. M. Landau, and J. S. B. Mitchell, "Edit distance of run-length encoded strings," *Information Processing Letters*, Vol. 83, No. 6, pp. 307–314, 2002.

[7] V. Bafna and P. A. Pevzner, "Genome rearrangements and sorting by reversals," *SIAM Journal of Computing*, Vol. 25, No. 2, pp. 172–289, 1996.

[8] V. Bafna and P. A. Pevzner, "Sorting by transpositions," *SIAM Journal on Discrete Mathematics*, Vol. 11, No. 2, pp. 224–240, 1998.

[9] A. Bergeron, J. Mixtacki, and J. Stoye, "Reversal distance without hurdles and fortresses," *Proceedings of 15th Annual Combinatorial Pattern Matching Symposium*, Vol. 3109, Istanbul, Turkey, pp. 389–399, 2004.

[10] P. Berman and S. Hannenhalli, "Fast sorting by reversal," *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, London, UK, pp. 168–185, 1996.

[11] P. Berman, S. Hannenhalli, and M. Karpinski, "1.375-approximation algorithm for sorting by reversals," *Proceedings of the 10th Annual European Symposium on Algorithms*, Vol. 2461, Rome, Italy, pp. 200–210, 2002.

[12] H. Bunke and J. Csirik, "An algorithm for matching run-length coded strings," *Computing*, Vol. 50, No. 4, pp. 297–314, 1993.

[13] H. Bunke and J. Csirik, "An improved algorithm for computing the edit distance of run-length coded strings," *Information Processing Letters*, Vol. 54, No. 2, pp. 93–96, 1995.

[14] A. Caprara, "Sorting by reversals is difficult," *Proceedings of the First International Conference on Computational Molecular Biology*, NM, USA, pp. 75–83, 1997.

[15] A. Caprara, "Sorting permutations by reversals and eulerian cycle decompositions," *SIAM Journal of Discrete Mathematics*, Vol. 12, No. 1, pp. 91–100, 1999.

[16] D. A. Christie, "A 3/2-approximation algorithm for sorting by reversals," *Proceeding of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, Pennsylvania, USA, pp. 244–252, 1998.

[17] D. A. Christie and R. W. Irving, "Sorting strings by reversals and by transpositions," *SIAM Journal on Discrete Mathematics*, Vol. 14, No. 2, pp. 193–206, 2001.

[18] G. Cormode and S. Muthukrishnan, "The string edit distance matching problem with moves," *Proceedings of the 13th annual ACM-SIAM symposium on Discrete algorithms*, San Francisco, USA, pp. 667–676, 2002.

[19] N. EI-Mabrouk, "Reconstructing an ancestral genome using minimum segments duplications and reversals," *Journal of Computer and System Sciences*, Vol. 65, No. 3, pp. 442–464, 2002.

[20] D. Eppstein, "Sequence comparison with mixed convex and concave cost," *Journal of Algorithms*, Vol. 11, pp. 85–101, 1990.

[21] F. Ergun, S. Muthukrishnan, and C. Sahinalp, "Comparing sequences with segment rearrangements," *Proceedings of the 23rd Foundations of Software Technology and Theoretical Computer Science*, Mumbai, India, pp. 183–194, 2003.

[22] Z. Galil and R. Giancarlo, "Speeding up dynamic programming with application to

molecular biology," *Theoretical Computer Science*, Vol. 64, pp. 107–118, 19889.

[23] M. R. Garey and D. S. Johnson, "Complexity results for multiprocessor scheduling under resource constraints," *SIAM Journal on Computing*, Vol. 4, pp. 397–411, 1975.

[24] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of Molecular Biology*, Vol. 162, No. 3, pp. 705–708, 1982.

[25] J. Gregor and M. G. Thomason, "Dynamic programming alignment of sequences representing cyclic patterns," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 15, No. 2, pp. 129–135, 1993.

[26] S. Hannenhalli, "Polynomial-time algorithm for computing translocation distance between genomes," *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, Espoo, Finland, pp. 162–176, 1996.

[27] S. Hannenhalli and P. A. Pevzner, "Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals," *Proceedings of the 27th Annual Symposium on Theory of Computing*, New York, USA, pp. 178–189, 1995.

[28] T. Hartman and R. Shamir, "A simpler 1.5-approximation algorithm for sorting by transpositions," *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, Morelia, Mexico, pp. 156–169, 2003.

[29] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequence," *Communications of the ACM*, Vol. 24, pp. 664–675, 1975.

[30] T.-C. Hsu, *An Algorithm for Computing the Distance of the Non-overlapping Inversion and Transposition.* National Sun Yat-sen University, Master's Thesis, Kaohusing, Taiwan, 2017.

[31] T. Jiang, G. Lin, B. Ma, and K. Zhang, "A general edit distance between RNA structures," *Journal of Computational Biology*, Vol. 9, No. 2, pp. 371–388, 2002.

[32] H. Kaplan and N. Shafrir, "The greedy algorithm for edit distance with moves," *Information Processing Letters*, Vol. 97, No. 1, pp. 23–27, 2006.

[33] J. Kececioglu and D. Sankoff, "Exact and approximation algorithms for the inversion distance between two permutations," *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching*, Vol. 684, Berlin, Germany, pp. 87–105, 1993.

[34] J. Kececioglu and D. Sankoff, "Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement," *Algorithmica*, Vol. 13, No. 1-2, pp. 180–210, 1995.

[35] J. W. Kim, A. Amir, G. M. Landau, and K. Park, "Computing similarity of run-length encoded strings with affine gap penalty," *Theoretical Computer Science*, Vol. 395, No. 2-3, pp. 268–282, 2008.

[36] M. M. Klawe and D. J. Kleitman, "An almost linear time algorithm for generalized matrix searching," *SIAM Journal on Discrete Mathematics*, Vol. 3, pp. 81–97, 1990.

[37] P. Kolman, "Approximating reversal distance for strings with bounded number of duplicates in linear time," *Proceedings of 30th International Symposium on Mathematical Foundations of Computer Science*, Gdansk, Poland, pp. 580–590, 2005.

[38] R. Kolpakov and G. Kucherov, "Finding maximal repetitions in a word in linear time," *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, New York, USA, pp. 596–604, IEEE Computer Society, 1999.

[39] J. J. Liu, G. S. Huang, Y. L. Wang, and R. C. T. Lee, "Edit distance for a run-length-encoded string and an uncompressed string," *Information Processing Letters*, Vol. 105, No. 1, pp. 12–16, 2007.

[40] J. J. Liu, Y. L. Wang, and R. C. T. Lee, "Finding a longest common subsequence between a run-length-encoded string and an uncompressed string," *Journal of Complexity*, Vol. 24, No. 2, pp. 173–184, 2008.

[41] D. Lopresti and A. Tomkins, "Block edit models for approximate string matching," *Theoretical Computer Science*, Vol. 181, pp. 159–179, 1997.

[42] R. Lowrance and R. A. Wagner, "An extension of the string-to-string correction problem," *Journal of the ACM*, Vol. 22, No. 2, pp. 177–188, 1975.

[43] M. Maes, "On a cyclic string-to-string correction problem," *Information Processing Letters*, Vol. 35, pp. 73–78, 1990.

[44] A. Marzal and S. Barrachina, "Speeding up the computation of the edit distance for cyclic strings," *In Proceeding of the 15th International Conference on Pattern Recognition*, Barcelona, Spain, pp. 895–898, 2000.

[45] W. J. Masek and M. S. Paterson, "A faster algorithm computing string edit distances,"

Journal of Computer and System Sciences, Vol. 20, pp. 18–31, 1980.

[46] W. Miller and E. W. Myers, “Sequence comparison with concave weighting functions,” *Bulletin of Mathematical Biology*, Vol. 50, No. 2, pp. 97–120, 1988.

[47] S. Muthukrishnan and S. C. Sahinalp, “Approximate nearest neighbors and sequence comparison with block operations,” *Proceedings of the 32nd Symposium on the Theory of Computing*, Portland, USA, pp. 416–424, 2000.

[48] E. W. Myers, “An $O(ND)$ difference algorithm and its variations,” *Algorithmica*, No. 1, pp. 251–266, 1986.

[49] D. B. Needleman and C. D. Wunsch, “A general method applicable to the search for similarities in the amino acid sequence of two proteins,” *Journal of Molecular Biology*, Vol. 48, No. 3, pp. 443–453, 1970.

[50] Y. H. Peng and C. B. Yang, “Finding the gapped longest common subsequence by incremental suffix maximum queries,” *Information and Computation*, Vol. 237, pp. 95–100, 2014.

[51] M. Schoniger and M. S. Waterman, “A local algorithm for DNA sequence alignment with inversions,” *Bulletin of Mathematical Biology*, Vol. 54, No. 4, pp. 521–536, 1992.

[52] P. H. Sellers, “An algorithm for the distance between two finite sequences,” *Journal of Combinatorial Theory*, Vol. 16, pp. 253–258, 1974.

[53] P. H. Sellers, “On the theory and computation of evolutionary distances,” *SIAM Journal on Applied Mathematics*, Vol. 26, No. 4, pp. 787–793, 1974.

[54] D. Shapira and J. A. Storer, “Large edit distance with multiple block operations,” *Proceedings of 10th International Symposium, String Processing and Information Retrieval*, Vol. 2857, Manaus, Brazil, pp. 369–377, 2003.

[55] D. Shapira and J. A. Storer, “Edit distance with move operations,” *Journal of Discrete Algorithms*, Vol. 5, No. 2, pp. 380–392, 2007.

[56] T. F. Smith and M. S. Waterman, “Comparison of biosequences,” *Advances in Applied Mathematics*, Vol. 2, pp. 482–489, 1981.

[57] T. F. Smith and M. S. Waterman, “Identification of common molecular subsequences,” *Journal of Molecular Biology*, Vol. 147, No. 1, pp. 195–197, 1981.

[58] T. F. Smith, M. S. Waterman, and W. M. Fitch, “Comparative biosequence metrics,” *Journal of Molecular Mubon*, Vol. 18, pp. 38–46, 1981.

[59] T. T. Ta, C. Y. Lin, and C. L. Lu, “An efficient algorithm for computing non-overlapping inversion and transposition distance,” *Information Processing Letters*, Vol. 116, pp. 744–749, 2016.

[60] E. Ukkonen, “Algorithms for approximate string matching,” *Information and Control*, Vol. 64, pp. 100–118, 1985.

[61] R. A. Wagner, “On the complexity of the extended string-to-string correction problem,” *Proceedings of Seventh Annual ACM Symposium on Theory of Computing*, New York, USA, pp. 218–223, 1975.

[62] R. A. Wagner and M. J. Fischer, “The string-to-string correction problem,” *Journal of the ACM*, Vol. 21, No. 1, pp. 168–173, 1974.

[63] M. E. M. T. Walter, Z. Dias, and J. Meidanis, “A new approach for approximating the transposition distance,” *Proceedings of the Seventh International Symposium on String Processing Information Retrieval*, Corunna, Spain, pp. 199–208, 2000.

[64] M. S. Waterman, “Efficient sequence alignment algorithms,” *Journal of Theoretical Biology*, Vol. 108, pp. 333–337, 1984.

[65] M. S. Waterman, T. F. Smith, and W. Beyer, “Some biological sequence metrics,” *Advances in Mathematics*, Vol. 20, No. 3, pp. 367–387, 1976.

[66] R. Wilber, “The concave least-weight subsequence problem revisited,” *Journal of Algorithms*, Vol. 9, pp. 418–425, 1988.

[67] C. K. Wong and A. K. Chandra, “Bounds of the string editing problem,” *Journal of the ACM*, Vol. 23, pp. 13–16, 1976.

[68] S. Wu, U. Manber, G. Myers, and W. Miller, “An O(NP) sequence comparison algorithm,” *Information Processing Letters*, Vol. 35, pp. 317–323, 1990.

[69] F. F. Yao, “Speed-up in dynamic programming,” *SIAM Journal on Algebraic Discrete Methods*, Vol. 3, pp. 532–540, 1982.