An Efficient Algorithm for Computing the Edit Distance with Non-overlapping Inversions*

Shie-Yan Lee
Chang-Biau Yang†
Department of Computer Science and Engineering
National Sun Yat-sen University
Kaohsiung, Taiwan
cbyang@cse.nsysu.edu.tw

Kuo-Tsung Tseng
Department of Shipping and Transportation Management
National Kaohsiung University of Science and Technology
Kaohsiung, Taiwan
tsengkt@nkust.edu.tw

ABSTRACT

Given two sequences with lengths m and n, the edit distance is the minimum cost required to transform one into the other. The classical operations involved in the traditional edit distance problem include insertion, deletion and replacement. In 2004, Gao *et al.* [5] proposed an algorithm for calculating the edit distance with non-overlapping inversion, where an inversion may overlap the classical operations, but not another inversion. The time complexity of their algorithm is $O(m^2n^2)$. In this paper, we introduce a new variant of the traditional edit distance problem, the edit distance with non-overlapping inversion (EDI) problem, which is to determine the minimum edit distance with four operations: insertion, deletion, replacement and inversion. The used operations cannot overlap one another. We propose an efficient algorithm for the EDI problem. The time complexity is $O(m^2n)$ in the worst case and $O(mn)$ in the average case.

CCS CONCEPTS

• **Theory of computation → Dynamic programming**.

KEYWORDS

dynamic programming, prefix and suffix, edit distance, edit distance with non-overlapping inversion (EDI) problem

ACM Reference Format:
Shie-Yan Lee, Chang-Biau Yang, and Kuo-Tsung Tseng. 2019. An Efficient Algorithm for Computing the Edit Distance with Non-overlapping Inversions. In *5th International Conference on Engineering & MIS 2019 (ICEMIS'19), June 6–8, 2019, Astana, Kazakhstan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3330431.3330437

*This research work was partially supported by the Ministry of Science and Technology of Taiwan under contract MOST107-2221-E-110-033.
†Corresponding author.

ICEMIS'19, June 6–8, 2019, Astana, Kazakhstan

ACM ISBN 978-1-4503-7212-1/19/06…$15.00
https://doi.org/10.1145/3330431.3330437

1 INTRODUCTION

Given two sequences (strings), one can be transformed into the other by inserting, deleting, or replacing some characters. The edit distance is the total cost of these insertions, deletions and replacements used in the transformation. The traditional *edit distance* problem is to find the used edit operations, consisting of insertions, deletions and replacements, with the minimal cost. When the cost of one insertion or deletion is one, and the cost of one replacement is two, the edit distance problem becomes to finding the minimal number of insertions and deletions, without considering replacements.

Levenshtein [11] presented an algorithm to calculate the edit distance of two sequences with three *classical operations*, insertion, deletion and replacement in 1966. In 1970, Needleman and Wunsch [13] presented a dynamic programming algorithm to align biological sequences such as DNA sequences or protein sequences. The alignment score can be seen as the similarity (opposite aspect of distance) of two biological sequences.

The similarity of DNA sequences is an important issue in bioinformatics. In the genome rearrangement problem, it finds the minimum mutation operations for the genome blocks. Besides the classical operations (insertion, deletion and replacement), there are some special operations for DNA sequences, such as reversal, inversion, and transposition. The genome rearrangement problem with overlapping operations has been proved to be NP-hard [4] and NP-complete [15]. Some approximate algorithms for the overlapping operations were proposed in the 1990s, such as 2-approximation for overlapping reversals with $O(n^2)$ time [9], 1.75-approximation for overlapping reversals with $O(n^2)$ time [3], and 2.25-approximation for overlapping transpositions with $O(n^2)$ time [20].

In 2004, Gao *et al.* [5] improved the algorithm presented by Schöniger and Waterman [14] and defined a new problem with non-overlapping inversions. Their problem allows inserting or deleting characters after a substring is inverted, so the classical operations may overlap an inversion but an inversion cannot overlap another inversion. The time complexity of their algorithm is $O(m^2n^2)$, where m and n denote the lengths of the two given sequences. In 2016, Ta *et al.* [16] proposed an algorithm for computing the mutation distance (including non-overlapping inversions and transpositions) of two sequences with the same length n. Hsu [6] improves the algorithm of Ta *et al.* by reducing the time complexity to $O(n^2)$.

In the past, the *longest common subsequence* (LCS) (the opposite aspect of edit distance) and its variants have been studied extensively. Some examples include block edit distance [1], block

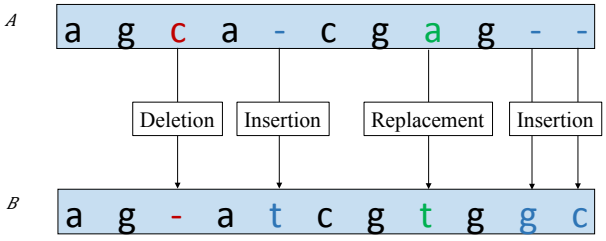

Figure 1: An example for transforming A = agcacgag into B = agatcgtggc with the edit cost $3\omega_{ins} + \omega_{del} + \omega_r$.

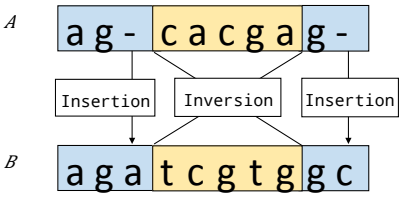

Figure 2: An example of the EDI cost for A = agcacgag and B = agatcgtggc, which is $2\omega_{ins} + \omega_{inv}$.

alignment [18], constrained LCS [2], merged LCS [17], and so on. A comprehensive survey on the edit distance and related variants can be found in the article of Lin *et al.* [12].

In this paper, we define a new variant of the edit distance problem, the edit distance with non-overlapping inversion (EDI) problem (including non-overlapping insertions, deletions, replacements and inversions), and propose an efficient algorithm for solving it.

The organization of this paper is given as follows. Some preliminaries are described in Section 2. We first present a brute-force algorithm for the EDI problem with $O(m^2n)$ time and space in Section 3. Then, one more efficient algorithm is presented in Section 4. There are two phases in the algorithm. We first build the needed tables for the inversions of the substrings, and then calculate the EDI distance by using these tables. The required time is $O(m^2n)$ in the worst case. The analysis of the average time complexity $O(mn)$ and experimental results are given in 5. Finally, the conclusions are given in Section 6.

2 PRELIMINARIES

Suppose we are given two sequences $A = a_1a_2\cdots a_m$ and $B = b_1b_2\cdots b_n$ with lengths $|A| = m$ and $|B| = n$, respectively, where $m \le n$. a_i denotes the ith character of A. The substring of A from positions i to j, $a_ia_{i+1}\cdots a_j$, is denoted by $A_{i..j}$. In this paper, A and B are over a finite alphabet $\Sigma = \{\mathtt{a}, \mathtt{t}, \mathtt{g}, \mathtt{c}\}$. Characters a and t are said to be *complementary* to each other, and so are g and c. The *complement* of the character a_i is denoted by $\overline{a_i}$. In other words, $\mathtt{a} = \overline{\mathtt{t}}$, $\mathtt{t} = \overline{\mathtt{a}}$, $\mathtt{g} = \overline{\mathtt{c}}$ and $\mathtt{c} = \overline{\mathtt{g}}$.

Traditionally, to transform sequence A into B can be considered as inserting/deleting/replacing some characters of A in the proper positions to make A and B identical. The cost of each insertion, deletion and replacement is denoted by ω_{ins}, ω_{del} and ω_r, respectively. The *edit distance* of A and B is the minimum cost to transform A into B [19]. Figure 1 shows an example for A = agcacgag and B = agatcgtggc. 3 insertions, 1 deletion and 1 replacement are used to make them identical. Thus edit cost for this transformation from A to B is $3\omega_{ins} + \omega_{del} + \omega_r$. When it is set that $\omega_{ins} = \omega_{del} = \omega_r$, this cost is the minimum and thus it is the edit distance of A and B.

Let $ED(i, j)$ denote the edit distance (minimum cost) to transform $A_{1..i}$ into $B_{1..j}$. The dynamic programming (DP) formula for solving the edit distance problem is given in Equation 1 [19].

$$ED(i,j) = \min \begin{cases} ED(i-1,j-1) + \omega(a_i, b_j), \\ ED(i-1,j) + \omega_{ins}, \\ ED(i,j-1) + \omega_{del}, \end{cases} \quad \text{for } 1 \le i \le m, 1 \le j \le n. \tag{1}$$

In Equation 1, the function $\omega(a, b)$ denotes the cost when a and b are aligned together, given as follows.

$$\omega(a,b) = \min \begin{cases} 0 & \text{if } a = b, \\ \omega_r & \text{if } a \ne b. \end{cases} \tag{2}$$

In this paper, we define a new variant of the edit distance problem, called the *edit distance with non-overlapping inversion* (EDI) problem, which involves four operations: insertion, deletion, replacement and inversion. An inversion is a special operation for DNA sequences. The *inverse* of sequence S is denoted by $\overline{S}$. The *inversion* operation is not only reversing the order of characters, but also replacing characters with their complements. For example, suppose that S = acaaca. Then $S_{3..6}$ = aaca, and its inverse $\overline{S_{3..6}} = \overline{s_6}\,\overline{s_5}\,\overline{s_4}\,\overline{s_3}$ = tgtt.

Definition 1. (inversion) *Given a sequence $S = s_1s_2s_3\cdots s_n$, an inversion operation from positions i to j is to reverse the order of substring $S_{i..j}$ and replace each character with its complement. After the inversion is done, the result is $S_{1..i-1}\,\overline{S_{i..j}}\,S_{j+1..n}$ = $s_1s_2\cdots s_{i-1}\overline{s_j}\,\overline{s_{j-1}}\cdots\overline{s_{i+1}}\,\overline{s_i}s_{j+1}\cdots s_n$, where $1 \le i \le j \le n$.*

Note that, in some researches [3, 7, 8], the operation that reverses the order of characters without replacing them by their complements is called a *reversal*.

Definition 2. (edit distance with non-overlapping inversion problem) *Given two sequences A and B with lengths m and n, respectively, the* edit distance with non-overlapping inversion (EDI) problem *is to determine the minimum cost required to transform A into B with the four operations: insertion, deletion, replacement and inversion. The used operations cannot overlap one another and their costs are denoted by ω_{ins}, ω_{del}, ω_r and ω_{inv}, respectively.*

Using the same sequences A and B in Figure 1, we give an example of the EDI problem in Figure 2 with EDI cost $2\omega_{ins} + \omega_{inv}$. If the cost of each operation is one, the EDI distance is 3, while the traditional edit distance is 5.

It is interesting that the score (cost) of the replacement operation, or called the alignment, is not a constant in the alignment problem for biosequences. A score table, such as PAM250 or BLOSUM100, records the score of every protein replacement (alignment). The

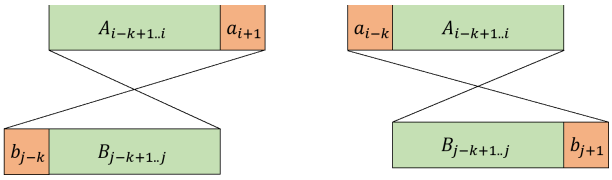

Figure 3: The extension of an inversion set.

goal of the alignment problem is to align two given biosequences with the maximum score [18].

3 A BRUTE-FORCE ALGORITHM

In this section, we first present a brute-force algorithm for the EDI problem with $O(m^2n)$ time and space. This algorithm can guide readers to understand the concept of our latter algorithm.

Definition 3. (inversion set) *Given two sequences A and B with lengths m and n, respectively, the inversion set $I_{i,j} = \{k \mid A_{i-k+1..i} = \overline{B_{j-k+1..j}}, 1 \le k \le \min\{i,j\}\}$, for $1 \le i \le m, 1 \le j \le n$.*

For example, suppose A = acaaca and B = tgttgt. There are two inversions ending at a_5 and b_6, which are $A_{1..5} = \overline{B_{2..6}}$ and $A_{4..5} = \overline{B_{5..6}}$. Thus, we have $I_{5,6} = \{2, 5\}$.

Definition 4. (extension of an inversion set) *Given an inversion set $I_{i,j}$, the extension of $I_{i,j}$ (to obtain $I_{i+1,j}$ or $I_{i,j+1}$) is to extend the lengths in $I_{i,j}$ by attaching one more character a_{i+1} or b_{j+1}. $ext(I_{i,j}, A) = \{k+1 \mid k \in I_{i,j} \cup \{0\}$ and $a_{i+1} = \overline{b_{j-k}}\}$. $ext(I_{i,j}, B) = \{k+1 \mid k \in I_{i,j} \cup \{0\}$ and $a_{i-k} = \overline{b_{j+1}}\}$.*

Figure 3 illustrates the extension of an inversion set. With the extension concept, we have the following lemma.

Lemma 1. $I_{i,j} = ext(I_{i-1,j}, A) = ext(I_{i,j-1}, B)$.

Proof. $ext(I_{i-1,j}, A) = \{p \mid p-1 \in I_{i-1,j} \cup \{0\}$ and $a_i = \overline{b_{j-p+1}}\}$, that is, $A_{i-p+1..i} = \overline{B_{j-p+1..j}}$. Assume that $g \in ext(I_{i-1,j}, A)$. Then $A_{i-g+1..i} = \overline{B_{j-g+1..j}}$ and we have $g \in I_{i,j}$. Thus, $ext(I_{i-1,j}, A) \subseteq I_{i,j}$.

Assume that $f \in I_{i,j}$. Then $A_{i-f+1..i} = \overline{B_{j-f+1..j}}$ and $a_i = \overline{b_{j-f+1}}$. $A_{i-f+1..i} = A_{i-f+1..i-1} + a_i$ and $\overline{B_{j-f+1..j}} = \overline{b_{j-f+1} + B_{j-f+2..j}} = \overline{B_{j-f+2..j}} + \overline{b_{j-f+1}}$. We have $A_{i-f+1..i-1} = \overline{B_{j-f+2..j}}$. The length of $A_{i-f+1..i-1}$ is $f-1$ and then $f-1 \in I_{i-1,j} \cup \{0\}$. Thus, $f \in ext(I_{i-1,j}, A)$ and then $I_{i,j} \subseteq ext(I_{i-1,j}, A)$. Combining the two facts, we get $I_{i,j} = ext(I_{i-1,j}, A)$. That $I_{i,j} = ext(I_{i,j-1}, B)$ can be proved similarly. Therefore, the lemma holds. □

Before calculating the distance, we build an inversion length table Θ that stores the lengths in all inversion sets. That is, we have to find every element k in $I_{i,j}$ for any i and j. Θ is a three-dimensional matrix. If $A_{i-k+1..i} = \overline{B_{j-k+1..j}}$, we assign $\Theta(i,j,k)$ to 1; and $\Theta(i,j,k) = 0$ if otherwise. That is, $\Theta(i,j,k) = 1$ if and only if $k \in I_{i,j}$. Procedure 1 describes our method for building the inversion length table Θ by Lemma 1.

Let $D(i,j)$ denote the EDI between $A_{1..i}$ and $B_{1..j}$. $D(i,j)$ depends not only on insertions and deletions but also on inversions ending

Procedure 1 Computation of the inversion length table Θ.

```
Θ(i, j, k) := 0, for 1 ≤ i ≤ m, 1 ≤ j ≤ n and 1 ≤ k ≤ min{i, j}.
for i := 1 → m do
    for j := 1 → n do
        if a_i = b_j then
            Θ(i, j, 1) = 1
        for k := 1 → min{i − 1, j − 1} do
            if Θ(i − 1, j, k) = 1 and a_i = b_{j−k} then
                Θ(i, j, k + 1) = 1 // ext(I_{i−1,j}, A)
```

at a_i and b_j. With the inversion length table Θ, we can calculate D with the dynamic programming approach given in Equation 3. Our brute-force algorithm is presented in Algorithm 1.

$$D(i,j) = \min \begin{cases} D(i-1,j-1) + \omega(a_i, b_j), \\ D(i-1,j) + \omega_{ins}, \\ D(i,j-1) + \omega_{del}, \\ D(i-k,j-k) + \omega_{inv}, \forall k \text{ s.t. } \Theta(i,j,k) = 1\ . \end{cases} \tag{3}$$

Algorithm 1 The brute-force algorithm for EDI

```
Compute Θ
for i := 1 → m do
    for j := 1 → n do
        d := min{D(i − 1, j − 1) + ω(a_i, b_j), D(i − 1, j) + ω_ins, D(i, j − 1) + ω_del}
        for k := 1 → min{i, j} do
            if Θ(i, j, k) = 1 then
                d := min{d, D(i − k, j − k) + ω_inv}
```

Since there are three for-loops in Procedure 1, the time complexity of building the inversion length table Θ is $O(m^2n)$, where $|A| = m \le |B| = n$. Therefore, the time complexity of Algorithm 1 (brute-force for EDI) is $O(m^2n)$ and the space complexity is $O(m^2n)$.

4 AN IMPROVED ALGORITHM

In this section, we improve the method for establishing the inversion length table to reduce the time complexity. Some new definitions are given for this improvement.

Definition 5. (longest common proper prefix-suffix, LCPPS) *Given a sequence S with length n, the* common proper prefix-suffix (*CPPS*) *is the prefix and suffix of S with length $p < n$ which are equal. That is, $S_{1..p} = S_{n-p+1..n}$. The* longest common proper prefix-suffix (*LCPPS*) *is the longest one in CPPS. If the LCPPS does not exist, the length of LCPPS is 0.*

For example, the CPPS's of acaaca are a and aca and its LCPPS is aca with length 3. The LCPPS of aaaaac does not exist, so the length of LCPPS is 0.

Given a sequence S, the prefix function of the KMP algorithm [10] $f(j) = \ell$ calculates the largest ℓ less than j such that $S_{1..\ell} = S_{j-\ell+1..j}$. For example, Figure 4 shows the results of the prefix function for acaacaca. That is, the result of the prefix function $f(j)$ is exactly the LCPPS of $S_{1..j}$. Let $\delta(i,j)$ denote the LCPPS length

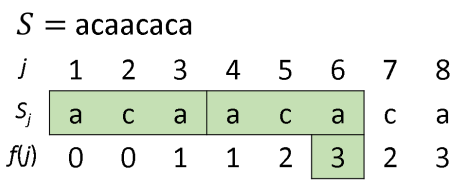

Figure 4: An example for the prefix function $f(j)$ of the KMP algorithm [10], which is $\delta(1, j)$ in our notation.

Procedure 2 Computing the δ table for string $A = a_1 a_2 \cdots a_m$

```
1: δ(i, j) := 0, for 1 ≤ i, j ≤ m.
2: for i := 1 → m − 1 do
3:     for j := i + 1 → m do
4:         ℓ := δ(i, j − 1)
5:         while ℓ > 0 and a_j ≠ a_{i+ℓ} do
6:             ℓ := δ(i, i + ℓ − 1)
7:         if a_j = a_{i+ℓ} then
8:             ℓ := ℓ + 1
9:         δ(i, j) := ℓ
```

of $A_{i..j}$. We modify the prefix function [10] to calculate $\delta(i, j)$, as shown in Procedure 2.

The time required for calculating the prefix function of a sequence S with length m is $O(m)$ [10]. Procedure 2 calculates the prefix function for every suffix $A_{i..m}$. There are $O(m)$ suffixes of A. Thus, the time required for building δ is $O(m^2)$.

Definition 6. (leftmost inversion) *Given an inversion set $I_{i,j}$, the leftmost inversion ending at a_i and b_j, denoted by $\lambda(i, j)$, is the inversion with maximum length in $I_{i,j}$. That is, $\lambda(i, j) = \max\{\ell \mid \ell \in I_{i,j}\}$.*

Instead of trying all possible lengths to check whether they could be inversions or not, we store only the leftmost inversion with which we can find all other inversions. For example, suppose A = acaaca and B = tgttgt. There are two inversions ending at a_5 and b_6, which are $A_{1..5} = \overline{B_{2..6}}$ and $A_{4..5} = \overline{B_{5..6}}$. So $\lambda(5, 6) = \max\{I_{5,6}\} = \max\{2, 5\} = 5$, which is the length of $A_{1..5}$.

There is a relationship between the leftmost inversion λ and the LCPPS length δ. Figure 5 illustrates an example of the relationship. Suppose that A = acaaca and B = tgttgt. $\lambda(5, 6) = 5$. Then $\lambda(6, 6) = 6$ since $a_6 = \overline{b_1}$. As another example, suppose that A = acaaca and B = cgttgt. $\lambda(5, 6) = 5$ but $a_6 \neq \overline{b_1}$. Then $\lambda(6, 6)$ is the LCPPS length of $A_{1..6}$. That is, $\lambda(6, 6) = \delta(1, 6) = 3$.

Lemma 2. *Suppose that $\lambda(i-1, j) = \alpha$. If $a_i = \overline{b_{j-\alpha}}$, then $\lambda(i, j) = \alpha + 1$. Otherwise, $\lambda(i, j) = \delta(i - \alpha, i)$.*

Proof. By definition, $\lambda(i, j) = \max\{I_{i,j}\} = \max\{ext(I_{i-1,j}, A)\}$. $A_{i-\alpha..i-1} = \overline{B_{j-\alpha+1..j}}$ since $\lambda(i-1, j) = \alpha \in I_{i-1,j}$. That is, $a_{i-\alpha+k} = \overline{b_{j-k}}$ for $0 \le k \le \alpha - 1$. The prefix of $A_{i-\alpha..i-1}$ with length x can be denoted by $A_{i-\alpha..i-\alpha+x-1}$, $1 \le x \le \alpha$. $A_{i-\alpha..i-\alpha+x-1} = \overline{B_{j-x+1..j}}$. $A_{i-\ell..i-1} = \overline{B_{j-\ell+1..j}} = A_{i-\alpha..i-\alpha+\ell-1}$ for $\ell \in I_{i-1,j}$, $\ell < \alpha$. $A_{i-\alpha..i-\alpha+\ell} = A_{i-\alpha..i-\alpha+\ell-1} + a_{i-\alpha+\ell} = \overline{B_{j-\ell+1..j}} + \overline{b_{j-\ell}} = \overline{B_{j-\ell..j}}$.

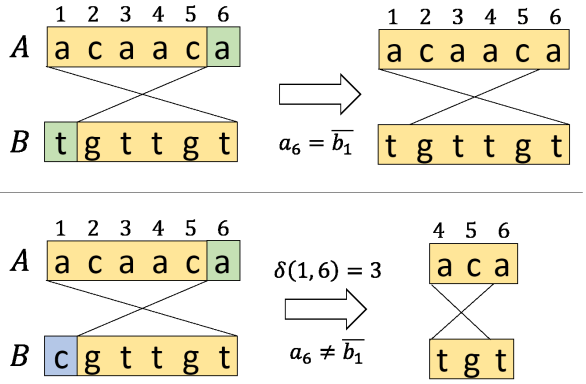

Figure 5: An example for the relationship between the leftmost inversion λ and the LCPPS δ.

If $a_i = \overline{b_{j-\ell}}$ and $\ell \in I_{i-1,j}$, then $\ell + 1 \in I_{i,j}$, that is, $A_{i-\ell..i} = \overline{B_{j-\ell..j}} = A_{i-\alpha..i-\alpha+\ell}$. Therefore, $\ell + 1$ is the length of the CPPS of $A_{i-\alpha..i}$. Let $\ell_m = \max\{\ell \mid \ell < \alpha \in I_{i-1,j}$ and $A_{i-\ell..i} = A_{i-\alpha..i-\alpha+\ell}\}$. $\ell_m + 1 \in I_{i,j}$. $\ell_m + 1$ is the length of the LCPPS of $A_{i-\alpha..i}$. If $a_i = \overline{b_{j-\alpha}}$, then $\max\{ext(I_{i-1,j}, A)\} = \alpha + 1$. Otherwise, $\max\{ext(I_{i-1,j}, A)\}$ is the LCPPS length of $A_{i-\alpha..i}$, that is $\delta(i - \alpha, i)$. □

By Lemma 2, we can build the leftmost inversion table λ with the recursive formula in Equation 4, where $\alpha = \lambda(i-1, j)$.

$$\lambda(i, j) = \max \begin{cases} \alpha + 1 & \text{if } a_i = \overline{b_{j-\alpha}}, \\ \delta(i - \alpha, i) & \text{if } \alpha > 1, \\ 1 & \text{if } a_i = \overline{b_j}, \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

Although we can find the leftmost inversions efficiently, we cannot calculate the EDI only with these leftmost inversions. For example, suppose A =agcagag and B = ctgctct. The EDI between A and B should be $2\omega_{inv}$ which is $A_{1..5} = \overline{B_{1..5}}$ and $A_{6..7} = \overline{B_{6..7}}$. However, $A_{6..7} = \overline{B_{6..7}}$ is not a leftmost inversion. Note that $A_{4..7} = \overline{B_{4..7}}$ is the leftmost inversion ending at a_7 and b_7. Therefore, we need to consider all inversions to get the correct EDI answer.

Given a leftmost inversion, we can find all other inversion lengths in the inversion set with LCPPS. For example, suppose that A = acaaca and B = tgttgt (see Figures 4 and 5). Then, $\lambda(6, 6) = 6$. The LCPPS length of $A_{1..6}$ is $\delta(1, 6) = 3 \in I_{6,6}$. The LCPPS length of $A_{4..6}$ is $\delta(4, 6) = 1 \in I_{6,6}$. That is, $I_{6,6} = \{1, 3, 6\}$.

We have the following theorem for describing the CPPS relationship. And, Figure 6 illustrates the CPPS relationship in Theorem 1.

Theorem 1. *Let $I_{i,j} = \{\ell_1, \ell_2, \ell_3, \cdots, \ell_{k-1}, \ell_k\}$ be an inversion set, where $\ell_1 < \ell_2 < \ell_3 < \cdots < \ell_{k-1} < \ell_k$. Then, ℓ_{t-1} is the LCPPS length of $A_{i-\ell_t+1..i}$ for $2 \le t \le k$. That is, $\ell_{t-1} = \delta(i - \ell_t + 1, i)$.*

Proof. Let $x \in I_{i,j}$. That is, $A_{i-x+1..i} = \overline{B_{j-x+1..j}}$. If there exists $y \in I_{i,j}$ and $y < x$, then $A_{i-y+1..i} = \overline{B_{j-y+1..j}} = A_{i-x+1..i-x+y}$ since $A_{i-x+1..i} = \overline{B_{j-x+1..j}}$. Then, y is the CPPS length of $A_{i-x+1..i}$. Therefore, for any $x, y \in I_{i,j}$ and $y < x$, y is the length of CPPS

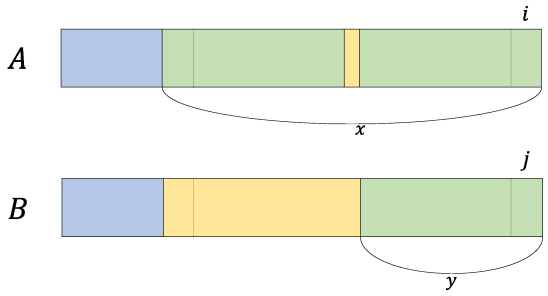

Figure 6: The CPPS relationship in Theorem 1.

of $A_{i-x+1..i}$. The CPPS lengths of $A_{i-\ell_t+1..i}$ are$\ell_1, \ell_2, \ell_3, \cdots$, and ℓ_{t-1}. Thus, ℓ_{t-1} is longest CPPS length (LCPPS length) of $A_{i-\ell_t+1..i}$ for $2 \le t \le k$. □

By Theorem 1, the improved algorithm for calculating the EDI with the inversion table λ and the LCPPS length table δ is presented in Algorithm 2.

Algorithm 2 An improved algorithm for EDI

```
1:  Compute δ
2:  Compute λ
3:  for i := 1 → m do
4:      for j := 1 → n do
5:          d := min{D(i − 1, j − 1) + ω(a_i, b_j), D(i − 1, j) + ω_ins, D(i, j − 1) + ω_del}
6:          if ℓ := λ(i, j) > 0 then
7:              d := min{d, D(i − ℓ, j − ℓ) + ω_inv}
8:              while ℓ := δ(i − ℓ + 1, i) > 0 do //Consider every inversion in I_{i,j}
9:                  d := min{d, D(i − ℓ, j − ℓ) + ω_inv}
10:         D(i, j) := d
```

Every entry of the λ table can be calculated in constant time. The time required for building λ is O(mn). In Algorithm 2, Line 5 includes the classical operations, which require O(mn) total time. Lines 6 through 9 include the inversion operations, which requires O(K) total time, where K denotes the number of inversions in sequences A and B. Therefore, the total time complexity of Algorithm 2 is O($mn + K$). The space complexity is O(mn).

5 ANALYSIS OF THE TIME COMPLEXITY

We first analyze the number of inversions in sequences A and B, denoted by K.

Definition 7. (substring pair) *Given two sequences A and B with lengths m and n, respectively, a substring pair with length ℓ is $A_{i..i+\ell-1}$ and $B_{j..j+\ell-1}$ for $1 \le i, i+\ell-1 \le m$ and $1 \le j, j+\ell-1 \le n$. The total number of all substring pairs with length ℓ is denoted by $N(\ell)$.*

Lemma 3. *Given two sequences with lengths m and n, $m \le n$, there are $O(m^2 n)$ substring pairs.*

$I_{i,j}$	1	2	3	4	5	6	7	8
1		1		1		1		1
2	1	2	1	2	1	2	1	2
3		1	2	1,3	2	1,3	2	1,3
4	1	2	1,3	2,4	1,3	2,4	1,3	2,4
5		1	2	1,3	2,4	1,3,5	2,4	1,3,5
6	1	2	1,3	2,4	1,3,5	2,4,6	1,3,5	2,4,6

A	a	g	a	g	a	g	
B	c	t	c	t	c	t	c t

Figure 7: The inversion set for A = agagag and B = ctctctct. For example, $I_{3,6} = \{1, 3\}$, where $A_{1..3} = \overline{B_{4..6}}$ and $A_{3..3} = \overline{B_{6..6}}$.

Proof. $N(i) = (m - i + 1)(n - i + 1)$, for $1 \le i \le \min\{m, n\} = m$. The number of all substring pairs is $\sum_{i=1}^{m} N(i) = \sum_{i=1}^{m}(m-i+1)(n-i+1) = \sum_{i=1}^{m}[m-(i-1)][n-(i-1)] = \sum_{i=1}^{m}[mn-(m+n)i+(m+n)+(i-1)^2] = m^2n-(m+n)\frac{m(m+1)}{2}+(m+n)m+\frac{(m-1)m(2m-1)}{6} = O(m^2n)$. □

Figure 7 shows the elements in every inversion set for A = agagag and B = ctctctct. There are totally 68 inversions in this example. The value of K is the maximum when every substring pair is an inversion. Accordingly, the time complexity of Algorithm 2 is O($mn + K$) = O(m^2n) in the worst case.

Theorem 2. *Given two sequences over alphabet Σ with lengths m and n, the average number of inversions $K = O(\frac{mn}{\sigma})$, where $\sigma = |\Sigma|$.*

Proof. To estimate the average value of K, it is assumed that each character appearing at every string position is independent and random. We have to consider the probability whether a substring pair is an inversion or not. The probability that a substring pair with length ℓ is an inversion is $(\frac{1}{\sigma})^\ell$. The expected number of inversions with length ℓ is $\frac{N(\ell)}{\sigma^\ell}$. Then, the average value of K equals to $\sum_{i=1}^{m}\frac{N(i)}{\sigma^i} = \sum_{i=1}^{m}\frac{(m-i+1)(n-i+1)}{\sigma^i} \le \sum_{i=1}^{m}\frac{mn}{\sigma^i} = \frac{mn}{\sigma^m}\sum_{i=1}^{m}\sigma^{m-i} = \frac{mn}{\sigma^m} \times \frac{\sigma^m-1}{\sigma-1} = O(\frac{mn}{\sigma})$. □

According to Theorem 2, the average time complexity of Algorithm 2 is O(mn).

Finally, we perform an experiment with random sequences to illustrate the efficiency of our algorithms in the real execution. The execution time of our algorithms is shown in Table 1 and Figure 8. The two input sequences are randomly generated with the same length from 200 to 2000.

6 CONCLUSION

In this paper, we define the edit distance with non-overlapping inversion (EDI) problem, which involves four operations, including insertion, deletion, replacement and inversion. The used operations

Table 1: The execution time of our algorithms in seconds.

String length	Algorithm 1 (brute-force)	Algorithm 2 (improved)
200	0.035	0.002
400	0.254	0.009
600	0.819	0.019
800	1.903	0.034
1000	3.655	0.053
1200	6.253	0.077
1400	9.889	0.105
1600	15.027	0.134
1800	20.779	0.17

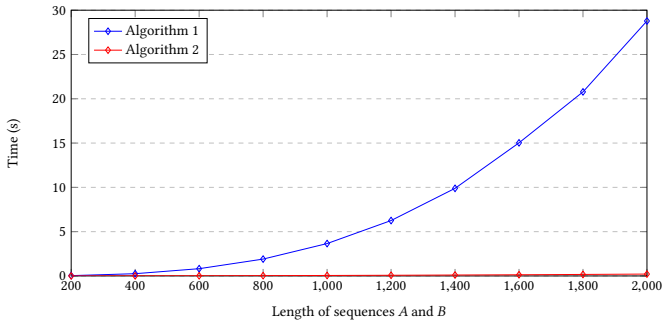

Figure 8: The execution time of our algorithms.

cannot overlap one another. We first present a brute-force algorithm for EDI with $O(m^2n)$ time. Then, we propose an efficient algorithm for the EDI problem which relies on the common proper prefix-suffix. The time complexity of the algorithm is $O(mn + K)$, where K is the number of all inversions between two sequences. Although the time complexity is $O(m^2n)$ in the worst case, the average time complexity is $O(mn)$.

In the future, we may add transposition or other special operations for DNA sequences into the problem. We hope that our algorithm could not only discover the appropriate similarity between DNA sequences but also calculate the similarity efficiently.

REFERENCES

[1] Hsing-Yen Ann, Chang-Biau Yang, Yung-Hsing Peng, and Bern-Cherng Liaw. 2010. Efficient Algorithms for the Block Edit Problems. *Information and Computation* (March 2010), 221–229. Issue 3.

[2] Hsing-Yen Ann, Chang-Biau Yang, and Chiou-Ting Tseng. 2014. Efficient Polynomial-Time Algorithms for the Constrained LCS Problem with Strings Exclusion. *Journal of Combinatorial Optimization* 28, 4 (November 2014), 800–813.

[3] Vineet Bafna and Pavel A Pevzner. 1996. Genome rearrangements and sorting by reversals. *SIAM J. Comput.* 25, 2 (1996), 272–289.

[4] Alberto Caprara. 1997. Sorting by reversals is difficult. In *Proceedings of the First Annual International Conference on Computational Molecular Biology*. ACM, Santa Fe, New Mexico, USA, 75–83.

[5] Yong Gao, Zhi-Zhong Chen, Guohui Lin, Robert Niewiadomski, Yang Wang, and Junfeng Wu. 2004. A space-efficient algorithm for sequence alignment with inversions and reversals. *Theoretical Computer Science* 325, 3 (2004), 361–372.

[6] Tzu-Chiang Hsu. 2017. An Algorithm for Computing the Distance of the Non-overlapping Inversion and Transposition. *Master thesis, Department of Computer Science and Engineering, National Sun Yat-sen University, Taiwan* (2017).

[7] Yen-Lin Huang and Chin Lung Lu. 2010. Sorting by Reversal, Generalized Transpositions, and Translocations Using Permutation Groups. *Journal of Computational Biology* 17, 5 (2010), 685–705.

[8] John Kececioglu and David Sankoff. 1993. Exact and approximation algorithms for the inversion distance between two chromosomes. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching*. Springer, 87–105.

[9] John Kececioglu and David Sankoff. 1995. Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement. *Algorithmica* 13, 1-2 (1995), 180–210.

[10] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. 1977. FAST PATTERN MATCHING IN STRINGS. *SIAM J. Comput.* 6, 2 (1977), 323–350.

[11] Vladimir Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions and Reversal. *Soviet Physics Doklady* 10, 8 (1966), 707–710.

[12] Shian-Liang Lin, Chiou-Ting Tseng, and Chang-Biau Yang. 2018. A Survey on the Algorithms of the Edit Distance Problem, the Genome Rearrangement Problem and Related Variants. In *Proceedings of the 35th Workshop on Combinatorial Mathematics and Computation Theory*. Taichung, Taiwan, 65–89.

[13] Saul B. Needleman and Christian D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* 48, 3 (1970), 443–453.

[14] Michael Schöniger and Michael S. Waterman. 1992. A Local Algorithm for DNA Sequence Alignment with Inversions. *Bulletin of Mathematical Biology* 54 (1992), 521–536. Issue 4.

[15] Dana Shapira and James A Storer. 2002. Edit distance with move operations. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching*. Springer, Fukuoka, Japan, 85–98.

[16] Toan Thang Ta, Cheng-Yao Lin, and Chin Lung Lu. 2016. An Efficient Algorithm for Computing Non-overlapping Inversion and Transposition Distance. *Inform. Process. Lett.* 116 (2016), 744–749. Issue 12.

[17] Kuo-Tsung Tseng, De-Sheng Chan, Chang-Biau Yang, and Shou-Fu Lo. 2018. Efficient Merged Longest Common Subsequence Algorithms for Similar Sequences. *Theoretical Computer Science* 7 (January 2018), 75–90.

[18] Kuo-Tsung Tseng, Chang-Biau Yang, Kuo-Si Huang, and Yung-Hsing Peng. 2008. Near-optimal Block Alignments. *IEICE Transactions on Information and Systems* E91-D, 3 (2008), 789–795.

[19] Robert A Wagner and Michael J Fischer. 1974. The string-to-string correction problem. *J. ACM* 21, 1 (1974), 168–173.

[20] Maria Emilia MT Walter, Zanoni Dias, and João Meidanis. 2000. A new approach for approximating the transposition distance. In *Proceedings of the International Symposium on String Processing and Information Retrieval*. La Coruna,Spain, 27–29.