# Efficient Algorithms for the Block Edit Problems *

Hsing-Yen Ann, Chang-Biau Yang† and Yung-Hsing Peng
Department of Computer Science and Engineering
National Sun Yat-sen University, Kaohsiung, Taiwan 80424
†cbyang@cse.nsysu.edu.tw

Bern-Cherng Liaw
Department of Finance and Banking
National Pingtung Institute of Commence, Pingtung, Taiwan

**Abstract**

*In this paper, we focus on the edit distance of two given strings where block-edit operations are allowed and better fitting to the natural edit behaviors. Previous results showed that this problem is NP-hard when recursive moves are allowed. Various approximations to this problem were also proposed in recent years. If no overlapping blocks are involved in these operations, however, this problem can be solved in polynomial time, even for the case of nested character-edit operations. In this paper, we define three problems with different measuring functions, which are $P(EIS, C)$, $P(EI, L)$ and $P(EI, N)$. Then we show that with some preprocessing, the minimum edit distances of these three problems can be obtained in $O(nm)$, $O(nm \log m)$ and $O((n+m)m^2)$ time, respectively, where $n$ and $m$ are the lengths of the two input strings.*

## 1 Introduction

The similarity computation of two strings or sequences is one of the most important fundamental in computer area. Several various versions of this problem have been studied over the past three decades, such as *edit distance*, *longest common subsequence* (*LCS*) [4, 6, 7, 8, 9, 13, 15, 17] and *Hamming distance*. The wide applications of this problem include finding similar strings, documents, pictures and even protein molecular sequences. In this paper, we shall focus on the edit distance of two given sequences. Wagner and Fischer [15] first proposed a dynamic programming method for solving this problem, with time complexity $O(nm)$, where $n$ and $m$ are the lengths of the two input subsequences. When a single character substitution can be replaced by a composition of an insertion and a deletion, Freschi and Bogliolo [4] presented a simple formula to do the transformation between the LCS lengths and edit distances. In addition to the original dynamic programming method, some more efficient algorithms have been proposed. Hirschberg [6] proposed methods with time complexity $O(pn + n \log n)$ and $O(p(m + 1 - p) \log n)$ where $p$ is the LCS length. Hunt and Szymanski [9] proposed a method with $O((r + n) \log n)$ time, where $r$ is the number of matches between the two input sequences. The algorithm given by Rick [13] requires $O(\min\{pm, p(n-p)\})$ time and $O(n)$ space. Yang and Lee solved the problem with the parallel systolic scheme [17].

Given two sequences $X$ and $Y$, the edit distance is defined as the distance caused by the mismatches between them. In other words, it can be regarded as the minimal cost to transform from $X$ to $Y$ by applying a series of valid operations on $X$. The traditional edit distance is defined by three types of operations: insertions, deletions and replacements. The edit distance can also be treated as a similarity metric of two given text sequences. Since a replacement is the only valid edit operation in the Hamming distance, we may note that edit distance is a more general similarity metric and it is closer to natural human edit behaviors on computers. In general, the costs of an insertion and a deletion may be different and the cost of a replacement may not be equal to an insertion plus a deletion. The costs of these edit operations can be defined by a score matrix. In this paper, we set one insertion or one deletion to be of a unit

*This research work was partially supported by the National Science Council of Taiwan under contract NSC-95-2221-E-110-102.

cost and a replacement operation is accomplished by one insertion followed by one deletion. In fact, our results can be applied to more variant forms of edit costs.

If the edit operations can be applied on segments of subsequence rather than single characters, the number of required operations may be drastically reduced. In another aspect, for better fitting to the human natural edit behaviors, we may include the *block-edit* operations. Besides the original character-edit operations, Shapira and Storer [14] added the block-move operation, where a block denotes one substring of a string. They proved that the edit distance problem is NP-hard when recursive block-moves are allowed, and an approximation to the problem was proposed. The approximation lower bound $\Omega(n^{0.43})$ was proved by Chrobak et al. [1]. Kaplan and Shafrir [10] gave a tighter lower bound $\Omega(n^{0.46})$.

When the *block-copy* operations are combined with *shift* operations, the block edit distances would be more useful for many applications, such as music databases searching. Here we denote $Z = z_1 z_2 \cdots z_n$ as a shift string of $X = x_1 x_2 \cdots x_n$ if for any $1 \leq i, j \leq n$, $J_{x_i} - J_{z_i} = J_{x_j} - J_{z_j}$, where $J_{x_i}$ denotes the encoded index (in an arbitrarily given order) of character $x_i$. One can apply this method to find out two identical melodies but with different pitches.

In this paper, we first define a class of block-edit problems which can be composed of some attributes. Then, three instances of the problems are given and their algorithms are also proposed.

## 2 Definitions and Preliminaries

We denote the input strings (sequences) $X = x_1 x_2 \cdots x_n$ and $Y = y_1 y_2 \cdots y_m$ as the initial string and final string, respectively, where $x_i \in \Sigma$, $1 \leq i \leq n$, and $y_j \in \Sigma$, $1 \leq j \leq m$. A substring $X_{i..j}$ of $X$ is defined as $X_{i..j} = x_i x_{i+1} x_{i+2} \cdots x_j$, where $1 \leq i \leq j \leq n$. For easy representation, the prefix $X_{1..i}$ of $X$ is simply denoted as $X_i$. A reverse string $X^R$ of $X$ is defined as $X = x_n x_{n-1} \cdots x_1$. While processing the edit operations, the intermediate strings are denoted by a series of working strings $\{W_1, W_2, \cdots, W_k\}$, where $W_i$ denotes the working string after the $i$th edit operation. The *classical edit distance* between $X$ and $Y$ with character-edit operations are denotes as $d_c(X, Y)$. The *local edit distance* [2, 12] between $X$ and $Y$, denoted as $d_l(X, Y)$, is defined as $\min\{d_c(X_{i..n}, Y) | 1 \leq i \leq n\}$. The recurrence formulas for determining $d_c(X, Y)$ and $d_l(X, Y)$ are given in Figure 1 [12]. We define the *substring edit distance* $d_{ml}(X, Y)$ between $X$ and $Y$ as the minimal number of character-edit operations to transform any substring of $X$ to $Y$, and the formal definition is given as $\min\{d_c(X_{i..j}, Y) | 1 \leq i \leq j \leq n\}$. It is easy to see that $d_{ml}(X, Y)$ is equal to $\min\{d_l(X_k, Y) | 1 \leq k \leq n\}$.

In this paper, we define a set of attributes which can be composed and applied on block-edit operations to construct different versions of edit problems. In our definition, we assume that a series of edit operations are performed from left to right on the strings, and any two block-edit operations would not be performed on overlapping substrings of input strings.

For each edit problem, there are two kinds of attributes, one is for copy behaviors and the other is for cost measure. We denote an edit problem as $P(o, c)$, where $o$ denotes a composition of copy operations and $c$ denotes the class of cost measures. Note that *block-copy* and *block-delete* are two block-operations newly defined in this paper, and the block-delete operations are allowed in all edit problems. They can be composed of some given attributes as follows:

**Copy/Delete Operations**

**External Copy (E):** Copy a substring of $X$ to the end of $W_i$, and form a new working string $W_{i+1}$. .

**Internal Copy (I):** Copy a substring of $W_i$ to the end of $W_i$, and form a new working string $W_{i+1}$.

**Shifted Copy (S):** Given a string $S$, append the shifted string $S'$ to the end of $W_i$, and form a new working string $W_{i+1}$.

**Delete:** Delete a substring of $W_i$, and form a new working string $W_{i+1}$.

**Cost Measures**

**Constant Cost (C):** All copy (or delete) operations are of the same cost $p_{copy}$ ($p_{delete}$).

**Linear Cost (L):** The cost of copying (or deleting) a string is $p_s + ip_e$, where $p_s$ and $p_e$ are constant parameters for the starting and extension penalties, respectively, and $i$ denotes the length of copied (deleted) string.

**Nested Cost (N):** All deletion operations are of the same cost $p_{delete}$, but the copied strings

$$d_c(X_i, Y_j) = \min \begin{cases} 0 & \text{, if } i = j = 0 \\ j & \text{, if } i = 0 \text{ and } j \neq 0 \\ i & \text{, if } i \neq 0 \text{ and } j = 0 \\ d_c(X_{i-1}, Y_{j-1}) & \text{, if } x_i = y_j \\ \min\{d_c(X_i, Y_{j-1}), d_c(X_{i-1}, Y_j)\} + 1 & \text{, if } x_i \neq y_j \end{cases}$$

$$d_l(X_i, Y_j) = \min \begin{cases} 0 & \text{, if } i = j = 0 \\ j & \text{, if } i = 0 \text{ and } j \neq 0 \\ 0 & \text{, if } i \neq 0 \text{ and } j = 0 \\ d_l(X_{i-1}, Y_{j-1}) & \text{, if } x_i = y_j \\ \min\{d_l(X_i, Y_{j-1}), d_l(X_{i-1}, Y_j)\} + 1 & \text{, if } x_i \neq y_j \end{cases}$$

Figure 1: The recurrence formulas for determining $d_c(X, Y)$ and $d_l(X, Y)$.

can be further edited with character-edit operations. Let $s_1$ denote the copied string and $s_2$ denote the string after editing. Then the cost of this copy operation is $p_{copy}+d_c(s_1, s_2)$.

With the combination of these attributes, we can easily define the class of block-edit problems. For example, $P(E, C)$ represents the problem that only external copies are allowed and all block-copy operations are with the same cost. As another example, in $P(EI, L)$, both external copies and internal copies are allowed and the cost of a block-copy operation depends linearly on the copied length. For two given strings $X$ and $Y$, we use $d(X, Y)$to denote the edit distance (cost) between them in various versions of edit problems.

## 3 Problems and Algorithms

In this section, we define some problems with the attributes defined in the previous section, and their corresponding algorithms are proposed.

### 3.1 Problem 1 – $P(EIS, C)$

Here we consider $P(EIS, C)$ in which all three copy operations (External, Internal and Shifted) are allowed and their costs are constant. We first propose a straightforward dynamic programming (DP) algorithm to solve this problem. The recurrence formula is given in Figure 2.

The time complexity of the straightforward DP algorithm is analyzed as follows. There are $O(nm)$ elements in the DP lattice $D$, where each element $D[i, j]$ stores the value of $d(X_i, Y_j)$. To calculate each $D[i, j]$, Equation (1) needs $O(n)$ time by linearly searching for the minimum. In Equation (2), for each suffix of $Y_j$, we can test if it is a substring of $X$ in $O(n + m)$ time by the KMP algorithm[11]. So, the testing of all suffixes needs $O(m(n+m))$ time. Then, we have to decide which length is the best one to be copied in the $O(m)$ candidates. The time needed by Equation (2) is $O(m(n+m))$. In Equation (3), by using the same strategy in (2), each suffix $Y_{l..j}$ of $Y_j$ can be easily tested if it is a substring of $Y_{l-1}$. The time needed by Equation (3) is $O(m^2)$. In Equations (4) and (5), for a given bias $\beta$, by using the same strategy in (2) and (3), the best string to be copied can be found in $O(m(n + m))$ and $O(m^2)$ time, respectively. After the testing all possible $O(|\Sigma|)$ biases, the time required by Equations (4) and (5) are $O(m(n + m)|\Sigma|)$ and $O(m^2|\Sigma|)$ time, respectively. Finally, we conclude that the time complexity of the above straightforward DP algorithm is $O(nm^2(n + m)|\Sigma|)$.

Next, we propose a more elegant algorithm to solve this problem. For Equation (1), the current minimum can be preserved for the next iteration, then it needs only $O(1)$ time per iteration. For Equation (2), the following two steps are involved:

**Step 1:** Find the longest suffix $Y_{l..j}$ of $Y_j$ that matches a substring of $X$.

**Step 2:** Find the best starting position $k$ in $Y_{l..j}$ so that the substring of $X$ is copied to $Y_{k..j}$.

In Step 1, $Y_{l..j}$ is a substring of $X$, but $Y_{l-1..j}$ is not, for $1 \leq l \leq j \leq m$ and it can be found in $O(1)$ time after the following preprocessing. First, a suffix tree[16] $T(X^R \# Y^R \$)$ is built, where $\{\#, \$\}$ are two dummy alphabets which are not in $\Sigma$. On each internal node of the suffix tree, there is a flag to preserve the information where its descendant leaf nodes came from ($\{ X^R, Y^R$, or both $\}$). Trace the path bottom-up from a leaf node $Y_j^R$ to the

$$d(X_i, Y_j) = \min \begin{cases} 0 & \text{, if } i = j = 0 \\ \infty & \text{, if } i < 0 \text{ or } j < 0 \\ d(X_{i-1}, Y_{j-1}) & \text{, if } x_i = y_j \\ \min\{d(X_i, Y_{j-1}), d(X_{i-1}, Y_j)\} + 1 & \text{, if } x_i \neq y_j \\ \min_{1 \le k \le i}\{d(X_{k-1}, Y_j) + p_{delete}\} & (1) \\ \min_{l \le k \le j}\{d(X_i, Y_{k-1}) + p_{copy}\} & \text{, where } Y_{l..j} \text{ is a substring of } X \quad (2) \\ \min_{l \le k \le j}\{d(X_i, Y_{k-1}) + p_{copy}\} & \text{, where } Y_{l..j} \text{ is a substring of } Y_{l-1} \quad (3) \\ \min_{l \le k \le j}\{d(X_i, Y_{k-1}) + p_{shift\_copy}\} & \text{, where } Y_{l..j} \text{ is a shifted substring of } X \quad (4) \\ \min_{l \le k \le j}\{d(X_i, Y_{k-1}) + p_{shift\_copy}\} & \text{, where } Y_{l..j} \text{ is a shifted substring of } Y_{l-1} \quad (5) \end{cases}$$

Figure 2: The recurrence formula for solving $P(EIS, C)$.

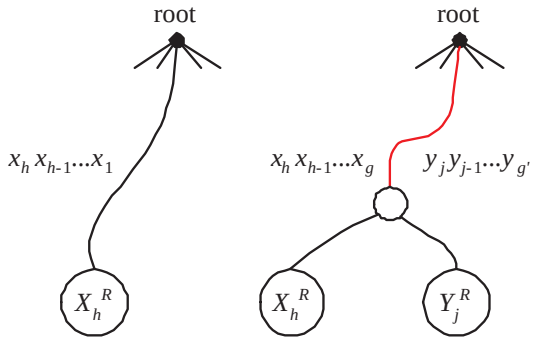


Figure 3: The LCA of $Y_j^R$ and $X_h^R$. $X_{g..h}$ is the longest substring of $X$ that matches a suffix of $Y_j$.

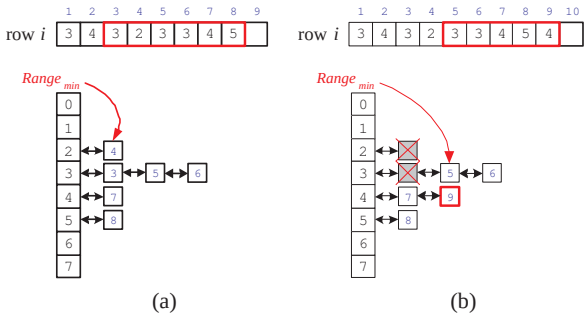


Figure 4: A data structure for the range minimum searching on integer data. (a) Finding the range minimum for $D(i, 9)$. (b) Updating the data structure for calculating $D(i, 10)$.

root and find the first internal node whose subtree contains both leaf nodes from $X^R$ and $Y^R$. This internal node is the *lowest common ancestor* (LCA) of $Y_j^R$ and some $X_h^R$, and our aim is to get the *longest common prefix* (LCP) $LCP(Y_j^R, X_h^R)$. Note that, this LCA is the lowest one among all LCAs which contain both leaf nodes from $X^R$ and $Y^R$. However, it cannot be found in constant time by the LCA query method shown in [5] because the leaf node $X_h^R$ remains unknown before tracing the path from $Y_j^R$ to the root. As shown in Figure 3, the reverse string of $LCP(Y_j^R, X_h^R)$ is exactly the longest suffix of $Y_j$ that matches in $X$ and it can be copied from $X$. To reduce the time spent by each query $j$, for $1 \le j \le m$, we can record the location of LCA to those internal nodes on the searching path. In another query $j'$, for $1 \le j' \le m$, if there exists an internal node that has been set by previous queries on the searching path, it can return the LCA information immediately, instead of the redundant searching. In summary, the preprocessing time for Step 1 can be done in $O(n+m)$ time, and then one can answer the longest substring to be copied in $O(1)$ time.

In Step 2, we need to find $\min_{l \le k \le j}\{d(X_i, Y_{k-1}) + p_{copy}\}$ that $Y_{k..j}$ is coped from $X$. This is to find the minimal value in the range from $D[i, l-1]$ to $D[i, j-1]$ in the DP lattice and then add it to $p_{copy}$. Here we consider the values of $p_{copy}$ and $p_{delete}$ are both integers, so the values in the searching range are also integers. For $p_{copy}$ and $p_{delete}$ of floating-point values, we will solve it with the same strategy shown in the next section. This searching can be done by the special data structure shown in Figure 4 which maintains the values of the current row (i.e. $D[i, 1]$ through $D[i, m]$) in the DP lattice. This data structure is composed of an array of pointers and a set of double linked lists. Note that the difference of $D(i, j)$ and $D(i, j-1)$ must be in $\{-1, 0, 1\}$. Therefore, the values stored in this data structure are compact. This data structure can guarantee that each insertion, deletion or range minimum searching can be achieved in $O(1)$ time.

Figure 4(a) shows the way of computing the value of $D[i, 9]$, where the searching range is assumed to be between $D[i, 3]$ and $D[i, 8]$. The indices from 3 to 8 are stored in the linked lists corre-

sponding to their values. For example, the index 5 is stored in the list of row 3 because $D[i,5]$ is 3. Note that the indices stored in each linked list are sorted since the indices are stored incrementally. A pointer $Range_{min}$ points to the first element of the linked list which contains the minimal value. Therefore, the query can be easily answered in $O(1)$ time. After obtaining the value of $D[i,9]$, the index 9 is stored in the end of the linked list in row 4 since $D[i,9] = 4$. When the iteration $D[i,10]$ begins, the indices 3 and 4 which are outside the range for the new minimum searching will be removed, as shown in Figure 4(b). Both the storing and the removing can be done in $O(1)$ time since the element to be removed is in the front of the linked list and the new element is to be stored in the end of the linked list. Note that the pointer $Range_{min}$ must be updated when the current minimum is removed or a smaller value with its index is inserted. Take Figure 4(b) as an example, the pointer $Range_{min}$ is not changed when the index 3 is removed, but when the index 4 is removed, $Range_{min}$ is updated to point to the new minimum, the index 5. The maintenance of the pointer $Range_{min}$ on the data structure needs $O(1)$ time for each store and each removal, and there are at most $m$ storing and $m$ removing operations for each row. In summary, one can answer the best starting position to be copied for Step 2 in $O(1)$ time. The overall answering time for Equation (2) is $O(1)$ for each iteration.

For Equation (3), similar to Equation (2), there are two steps involved:

**Step 1:** Find the longest suffix $Y_{l..j}$ of $Y_j$ that matches a substring of $Y_{l-1}$.

**Step 2:** Find the best starting position $k$ in $Y_{l..j}$ so that the substring of $Y_{l-1}$ is copied to $Y_{k..j}$.

In Step 1, $Y_{l..j}$ is a substring of $Y_{l-1}$, but $Y_{l-1..j}$ is not a substring of $Y_{l-2}$, for $1 \le l \le j \le m$. Note that in Equation (3), when the dynamic programming is used to find the best suffix that can be copied, the overlapping regions must be avoided. As shown in Figure 5, the best suffix $Y_{l..j}$ in the first case can be copied directly, but in the second case, the longest valid suffix that can be copied, rather than the best suffix, is $Y_{j'+1..j}$. This query can be answered in $O(1)$ time after the following preprocessing. First, a suffix tree $T(Y^R)$ is built, and on each internal node of the suffix tree, there is an extra pointer to the leaf node which has the smallest index in $Y$ among all its descendant leaf nodes. If two or more substrings can be copied

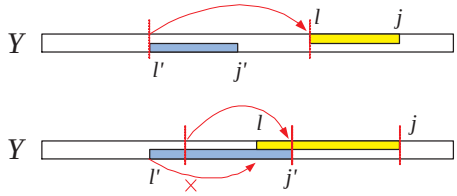


Figure 5: The avoidance of overlapping region when the suffix of $Y_j$ is copied.

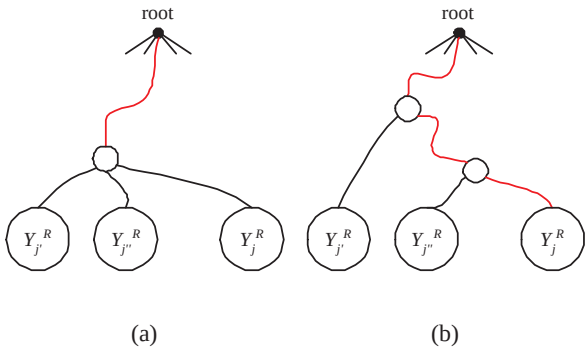


Figure 6: The suffix tree $T(Y_j^R)$.

to $Y_{l..j}$, the one with the smaller index is always better because it will make the overlapping region smaller, as shown in Figure 6(a). Then the longest valid suffix will appear on the path from the root to the leaf node $Y_j^R$ which is the LCP with removing the overlapping region, as shown in Figure 5 and Figure 6(b). The expected depth of the suffix tree $T(Y^R)$ is $O(\log m)$ [3] and the preprocessing time becomes $O(m \log m)$ time in average and $O(m^2)$ time in the worst case. In Step 2, one can answer the best starting position to be copied, by using the same strategy for Equation (2), in $O(1)$ time. The overall answering time for Equation (3) is $O(1)$ per iteration.

To find the string that can be copied with a shift operation, we can first compute the differential strings $X'$ and $Y'$ of $X$ and $Y$ which are defined as $X' = x'_1 x'_2 \cdots x'_{n-1}$ and $Y' = y'_1 y'_2 \cdots y'_{m-1}$, respectively, where $x'_i = J_{x_{i+1}} - J_{x_i}$, $1 \le i \le n-1$ and $y'_j = J_{y_{j+1}} - J_{y_j}$, $1 \le j \le m-1$. For Equation (4), the strategy for solving Equation (2) can be applied similarly by preprocessing the suffix tree $T(X'^R \# Y'^R \$)$, rather than $T(X^R \# Y^R \$)$. Then the preprocessing time is still $O(n+m)$ and the answering time is $O(1)$ per iteration. For Equation (5), the strategy for solving Equation (3) can be applied similarly by preprocessing the suffix tree $T(Y'^R)$ rather than $T(Y^R)$.

As a summary, we have the following theorem.

**Theorem 1.** *$P(EIS, C)$ can be solved in $O(nm)$ time with $O(n + m^2)$ preprocessing time in the worst cast and $O(n + m \log m)$ preprocessing time in average.*

### 3.2 Problem 2 – $P(EI, L)$

Figure 7 shows the recurrence formula for solving $P(EI, L)$. A straightforward implementation, similar to that for $P(EIS, C)$, can solve this problem in $O(nm^2(n + m))$ time. Note that this problem cannot be solved by the algorithm for $P(EIS, C)$ because of two key reasons.

First, in general, $p_e$ is less than the unit cost, which is the cost for a single character insertion. The data structure shown in Figure 4 is not workable for floating-point values because we cannot point to the linked list of a given value in $O(1)$ time. A balanced binary search tree can be used as an alternate, which can perform one insertion, deletion or query in $O(\log m)$ time. Second, in $P(EIS, C)$, the values stored in the searching range are never changed, but in $P(EI, L)$, once the iteration $D(i, j)$ passes to the iteration $D(i, j+1)$, all the values in the searching range are increased with the cost $p_e$ except the newly inserted element $D[i, j]$. To avoid updating all values stored in the balanced binary search tree, we subtract the amount $p_e$ from the newly inserted element, rather than add $p_e$ to all the stored elements.

The time required by this algorithm is analyzed as follows. For Equation (1), it needs $O(\log m)$ time to find the best suffix of $X_i$ with the minimal cost that can be deleted per iteration. For Equations (2) and (3), the preprocessing still requires $O(n + m)$ and $O(m^2)$ time, respectively, but the answering time is both increased to $O(\log m)$ time per iteration by using the balanced binary search tree.

As a summary, we have the following theorem.

**Theorem 2.** *$P(EI, L)$ can be solved in $O(nm \log m)$ time with $O(n + m^2)$ preprocessing time.*

### 3.3 Problem 3 – $P(EI, N)$

The recurrence formula for solving $P(EI, N)$ is given in Figure 8. A straightforward implementation requires $O(n^2m^3)$ time. We propose a more efficient algorithm as follows.

For Equation (1), $p_{delete}$ is a constant, and the strategy for $P(EIS, C)$ can be applied similarly, thus it needs $O(1)$ time per iteration. For Equation (2), we want to find out $k$ such that

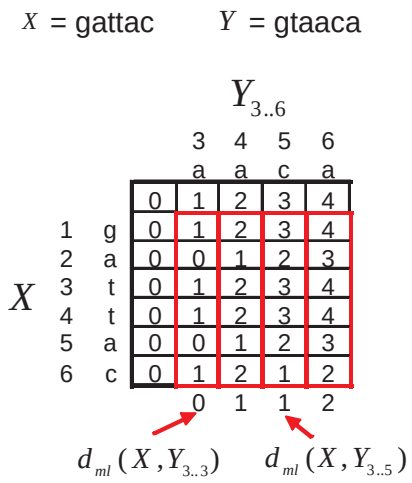


Figure 9: The DP lattice for finding the substring edit distances $d_{ml}(X, Y_{3..3})$, $d_{ml}(X, Y_{3..4})$, $d_{ml}(X, Y_{3..5})$ and $d_{ml}(X, Y_{3..6})$.

$d(X_i, Y_k) + d_{ml}(X, Y_{k..j})$ is minimal. To solve this problem, we have to get the substring edit distance $d_{ml}(X, Y_{k..j})$ first, where $1 \le k \le j \le m$. This can be done by preprocessing the DP lattice of $X$ and $Y_{k..m}$ for each $k$ between 1 and $m$, and there will be $m$ DP lattices. For example, Figure 9 shows a DP lattice of $X$ and $Y_{3..6}$, where $k = 3$ and $m = 6$. By computing the minimum value of column $k'$, we can get the substring edit distance $d_{ml}(X, Y_{k..k'})$. The construction of the $m$ DP lattices needs $O(nm^2)$ preprocessing time. Then, we need to find out $k$ such that $d(X_i, Y_{k-1}) + d_{ml}(X, Y_{k..j})$ is minimal, which needs $O(m)$ time per iteration. For Equation (3), we first have to get the substring edit distance $d_{ml}(Y_{k-1}, Y_{k..j})$, where $1 \le k \le j \le m$. Similarly, this can be done by preprocessing the DP lattice of $Y_{k-1}$ and $Y_{k..m}$ for each $k$ between 1 and $m$, and there will be $m$ DP lattices. The construction of the $m$ DP lattices needs $O(m^3)$ preprocessing time. Then, we can find out $k$ such that $d(X_i, Y_{k-1}) + d_{ml}(Y_{k-1}, Y_{k..j})$ is minimal, which needs $O(m)$ time per iteration.

As a summary, we have the following theorem.

**Theorem 3.** *$P(EI, N)$ can be solved in $O(nm^2)$ time with $O((n + m)m^2)$ preprocessing time.*

$$d(X_i, Y_j) = \min \begin{cases} 0 & \text{, if } i = j = 0 \\ \infty & \text{, if } i < 0 \text{ or } j < 0 \\ d(X_{i-1}, Y_{j-1}) & \text{, if } x_i = y_j \\ \min\{d(X_i, Y_{j-1}), d(X_{i-1}, Y_j)\} + 1 & \text{, if } x_i \neq y_j \\ \min_{1 \leq k \leq i}\{d(X_{k-1}, Y_j) + p_s + (i-k+1)p_e\} & (1) \\ \min_{l \leq k \leq j}\{d(X_i, Y_{k-1}) + p_s + (j-k+1)p_e\} & \text{, where } Y_{l..j} \text{ is a substring of } X \quad (2) \\ \min_{l \leq k \leq j}\{d(X_i, Y_{k-1}) + p_s + (j-k+1)p_e\} & \text{, where } Y_{l..j} \text{ is a substring of } Y_{l-1} \quad (3) \end{cases}$$

Figure 7: The recurrence formula for solving $P(EI, L)$.

$$d(X_i, Y_j) = \min \begin{cases} 0 & \text{, if } i = j = 0 \\ \infty & \text{, if } i < 0 \text{ or } j < 0 \\ d(X_{i-1}, Y_{j-1}) & \text{, if } x_i = y_j \\ \min\{d(X_i, Y_{j-1}), d(X_{i-1}, Y_j)\} + 1 & \text{, if } x_i \neq y_j \\ \min_{1 \leq k \leq i}\{d(X_{k-1}, Y_j) + p_{delete}\} & (1) \\ \min_{l \leq k \leq j}\{d(X_i, Y_{k-1}) + d_{ml}(X, Y_{k..j}) + p_{copy}\} & (2) \\ \min_{l \leq k \leq j}\{d(X_i, Y_{k-1}) + d_{ml}(Y_{k-1}, Y_{k..j}) + p_{copy}\} & (3) \end{cases}$$

Figure 8: The recurrence formula for solving $P(EI, N)$.

## 4 Conclusion

We solve the $P(EIS, C)$ problem in $O(nm)$ time, and it is useful when it is compared to the classical edit distances with character-edit operations. For $P(EI, L)$, the time complexity is increased to $O(nm \log m)$ because of the floating-point cost. The $P(EI, N)$ problem is better fitted to the human natural edit behavior, and it can be solved in $O(nm^2)$ time. We are still interested in whether the $P(EI, N)$ problem can be solved by more efficient algorithms.

## References


[1] M. Chrobak, P. Kolman, and J. Sgall, "The greedy algorithm for the minimum common string partition problem," *ACM Transactions on Algorithms*, Vol. 1, No. 2, pp. 350–366, 2005.

[2] R. Cole and R. Hariharan, "Approximate string matching: a simpler faster algorithm," *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pp. 463–472, 1998.

[3] J. Fayolle and M. D. Ward, "Analysis of the average depth in a suffix tree under a markov model," *International Conference on Analysis of Algorithms DMTCS Proc. AD*, pp. 95–104, 2005.

[4] V. Freschi and A. Bogliolo, "Longest common subsequence between run-length-encoded strings: a new algorithm with improved parallelism," *Information Processing Letters*, Vol. 90, pp. 167–173, 2004.

[5] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[6] D. S. Hirschberg, "Algorithms for the longest common subsequence problem," *Journal of the ACM*, Vol. 24, No. 4, pp. 664–675, 1977.

[7] K. S. Huang, C. B. Yang, K. T. Tseng, H. Y. Ann, and Y. H. Peng, "Algorithms for the merged-lcs problem and its variant with block constraint," *Proc. of the 23rd Workshop on Combinatorial Mathematics and Computation Theory, Chang-Hua, Taiwan*, pp. 232–239, 2006.

[8] K. S. Huang, C. B. Yang, K. T. Tseng, Y. H. Peng, and H. Y. Ann, "Dynamic programming algorithms for the mosaic longest common subsequence problem," *Information Processing Letters*, Vol. 102, pp. 99–103, 2007.

[9] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, Vol. 20, No. 5, pp. 350–353, 1977.

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

[11] D. E. Knuth, J. H. Morris Jr., and V. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, Vol. 6, No. 2, pp. 323–350, 1977.

[12] G. M. Landau and U. Vishkin, "Fast parallel and serial approximate string matching," *Journal of Algorithms*, Vol. 10, No. 2, pp. 157–169, 1989.

[13] C. Rick, "Simple and fast linear space computation of longest common subsequences," *Information Processing Letters*, Vol. 75, No. 6, pp. 275–281, 2000.

[14] D. Shapira and J. A. Storer, "Edit distance with move operations," *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching*, Vol. 2373, pp. 85–98, 2002.

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

[16] P. Weiner, "Linear pattern matching algorithm," *In Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pp. 1–11, 1973.

[17] C. B. Yang and R. C. T. Lee, "Systolic algorithm for the longest common subsequence problem," *Journal of the Chinese Institue of Engineers*, Vol. 10, No. 6, pp. 691–699, 1987.