Information and Computation 208 (2010) 221–229

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/ic

Efficient algorithms for the block-edit problems[☆]

Hsing-Yen Ann[a], Chang-Biau Yang[a,*], Yung-Hsing Peng[a], Bern-Cherng Liaw[b]

[a] Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung 80424, Taiwan
[b] Department of Finance and Banking, National Pingtung Institute of Commence, Pingtung, Taiwan

ARTICLE INFO

Article history:
Received 2 September 2007
Revised 4 October 2009
Available online 4 January 2010

Keywords:
Design of algorithms
String edit distance
Block-edit
Similarity
Dynamic programming

ABSTRACT

In this paper, we focus on the edit distance between two given strings where block-edit operations are allowed and better fitting to the human natural edit behaviors. Previous results showed that this problem is NP-hard when block moves are allowed. Various approximations to this problem have been proposed in recent years. However, this problem can be solved by the polynomial-time optimization algorithms if some reasonable restrictions are applied. The restricted variations which we consider involve character insertions, character deletions, block copies and block deletions. In this paper, three problems are defined 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 block edit distances of these three problems can be obtained by dynamic programming in $O(nm)$, $O(nm \log m)$ and $O(nm^2)$ time, respectively, where n and m are the lengths of the two input strings.

© 2009 Elsevier Inc. All rights reserved.

1. Introduction

The similarity computation of two strings or sequences is one of the most important fundamental in the computer area. Several various versions of this problem have been studied over the past three decades, such as *edit distance*, *longest common subsequence* (*LCS*) [1,2,3,4,5,6,7,8] and *Hamming distance* [9]. 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 between two given sequences. Wagner and Fischer [7] 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 [1] 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 [2] 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 [5] 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 [6] requires $O(\min\{pm, p(n - p)\})$ time and $O(n)$ space. Yang and Lee solved the problem with the parallel systolic scheme [8].

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 the only valid edit operation in the Hamming distance is replacement, we may note that edit distance is a more general similarity metric and it is closer to natural human edit

[☆] This research work was partially supported by the National Science Council of Taiwan under Contract NSC-95-2221-E-110-102.
[*] Corresponding author.
E-mail address: cbyang@cse.nsysu.edu.tw (C.-B. Yang).

0890-5401/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2009.12.001

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 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. Shapira and Storer [10] added the block-move operation to the traditional edit distance problem and proved that this problem is NP-hard. They also proposed a GREEDY algorithm and claimed that it is a $\log n$-approximation. Chrobak et al. [11] showed that the claim is false by proving that the approximation lower bound of GREEDY is $\Omega(n^{0.43})$. Kaplan and Shafrir [12] gave a tighter lower bound $\Omega(n^{0.46})$. Muthukrishnan and Sahinalp [13] proposed an algorithm of Monte Carlo type for solving the problems involving block copies, block deletions, block moves and block reversals, and the algorithm achieves $O(\log n \log^* n)$ approximation. Another algorithm, based on Lempel-Ziv-77 method, which was proposed by Ergun et al. [14] achieve a factor of 12-approximation. Shapira and Storer [15] reduced the constant factor to 3.5. Recently, Shapira and Storer proposed a revised version [16] of [10] and showed that the error can be corrected if only a subclass of instances of the general problem is coped with.

However, the problems which involve block-edit operations can still be solved by the polynomial-time optimization algorithms if some restrictions are applied. Muthukrishnan and Sahinalp [13] considered the problem which only consists of character replacements and block reversals and proposed an algorithm with $O(n \log^3 n)$ time. Shapira and Storer [15] considered the problem which consists of character insertions, block deletions and character moves. Their algorithm finds the optimal solutions by merging character insertions and character deletions to character moves in any optimal path of the traditional edit distance. Rather than matching the blocks exactly, Lopresti and Tomkins [17] showed a model in which the matched blocks can be further edited with character operations. They showed that some variations are NP-hard and gave polynomial-time algorithms for others.

Ukkonen [18] defined a *restricted block-edit problem* in which the block replacement operations are extended from the character replacement operations and the edit operations must be in a restricted order. With this proposed restriction, Ukkonen showed that this restricted variation can be solved by a dynamic programming (DP) algorithm. Although his algorithm is correct conceptually, the DP formula and the time complexity of his algorithm may be incorrect. He claimed that the time complexity is $O(s \cdot \min(m, n))$, where s denotes the edit distance, however, we think that the correct algorithm based on his idea requires $O(s^2 \cdot \min(m^2, n^2))$ time. This is because, in the worst case, $O(s \cdot \min(m, n))$ possible block replacements may be considered for each iteration, but his algorithm considers only one possible block replacement for each iteration greedily.

In this paper, we follow the same restriction defined by Ukkonen [18] and define some restricted variations which involve character insertions, character deletions, block copies and block deletions. The block-edit operations can be attached with some attributes, such as the copy behaviors and cost measures. The formal definitions of the attributes and the preliminaries are given in Section 2. In Section 3, three problems are defined 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 block edit distances of these three problems can be obtained by dynamic programming in $O(nm)$, $O(nm \log m)$ and $O(nm^2)$ time, respectively, where n and m are the lengths of the two input strings. Finally, the conclusion is given in Section 4.

2. Definitions and preliminaries

We denote the input strings (sequences) $X = x_1x_2 \cdots x_n$ and $Y = y_1y_2 \cdots y_m$ as the initial string and final string, respectively, where $x_i \in \Sigma$, $1 \leqslant i \leqslant n$, and $y_j \in \Sigma$, $1 \leqslant j \leqslant m$. A substring $X_{i\ldots j}$ of X is defined as $X_{i\ldots j} = x_i x_{i+1} x_{i+2} \cdots x_j$, where $1 \leqslant i \leqslant j \leqslant n$. For easy representation, the prefix $X_{1\ldots i}$ of X is simply denoted as X_i. A reverse string X^R of X is defined as $X^R = x_n x_{n-1} \cdots x_1$. While processing the edit operations, the intermediate strings are denoted by a series of working strings $\{W_1, W_2, \ldots, W_K\}$, where W_i denotes the working string after the ith edit operation. Generally, X and Y are regarded as W_0 and W_{K+1}, respectively. The *traditional edit distance* between X and Y with character-edit operations are denoted as $d_{tra}(X, Y)$. The *local edit distance* [19,20] between X and Y, denoted as $d_{local}(X, Y)$, is defined as $\min\{d_{tra}(X_{i\ldots n}, Y) | 1 \leqslant i \leqslant n\}$. The recurrence formulas for determining $d_{tra}(X, Y)$ and $d_{local}(X, Y)$ are given in Fig. 1 [20]. The *substring edit distance* $d_{sub}(X, Y)$ between X and Y is defined as the minimal number of character-edit operations to transform any substring of X to Y, and the formal definition is given as $\min\{d_{tra}(X_{i\ldots j}, Y) | 1 \leqslant i \leqslant j \leqslant n\}$. It is easy to see that $d_{sub}(X, Y)$ is equal to $\min\{d_{local}(X_k, Y) | 1 \leqslant k \leqslant n\}$.

In this paper, we assume that the edit operations which transform X to Y form a *restricted editing sequence* defined by Ukkonen [18]. Let $W_i = U_iV_i$ denote the ith intermediate string, where W_i is divided into two parts, *inactive part* U_i and *active part* V_i. Let $A \to B$ be an edit operation which transforms a prefix A of V_i to a string B in one step, that is, V_i can be written as AV'_i for some (possibly empty) strings V'_i, A and B. After the edit operation $A \to B$ is performed, we can get the next intermediate string $W_{i+1} = U_{i+1}V_{i+1}$, where the inactive part $U_{i+1} = U_iB$ and the active part $V_{i+1} = V'_i$. There exists a *restricted editing sequence* from X to Y if one can produce $Y = W_{K+1}$ from $X = W_0$ in such a way. Note that even if there is a match, i.e. $A = B$, the active part should be shortened and the inactive part should be extended.

Initially, the active part is the whole string X and the inactive part is an empty string. After the last edit operation is performed, the active part becomes an empty string and the inactive part becomes the string Y. It derives that the block

$$
\begin{aligned}
d_{tra}(X_0, Y_0) &= 0 \\
d_{tra}(X_i, Y_0) &= i, \forall 1 \leqslant i \leqslant n \\
d_{tra}(X_0, Y_j) &= j, \forall 1 \leqslant j \leqslant m \\
d_{tra}(X_i, Y_j) &= \begin{cases} d_{tra}(X_{i-1}, Y_{j-1}), & \text{if } x_i = y_j \\ \min\{d_{tra}(X_i, Y_{j-1}), d_{tra}(X_{i-1}, Y_j)\} + 1, & \text{if } x_i \neq y_j \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
d_{local}(X_0, Y_0) &= 0 \\
d_{local}(X_i, Y_0) &= 0, \forall 1 \leqslant i \leqslant n \\
d_{local}(X_0, Y_j) &= j, \forall 1 \leqslant j \leqslant m \\
d_{local}(X_i, Y_j) &= \begin{cases} d_{local}(X_{i-1}, Y_{j-1}), & \text{if } x_i = y_j \\ \min\{d_{local}(X_i, Y_{j-1}), d_{local}(X_{i-1}, Y_j)\} + 1, & \text{if } x_i \neq y_j \end{cases}
\end{aligned}
$$

Fig. 1. The recurrence formulas for determining $d_{tra}(X, Y)$ and $d_{local}(X, Y)$.

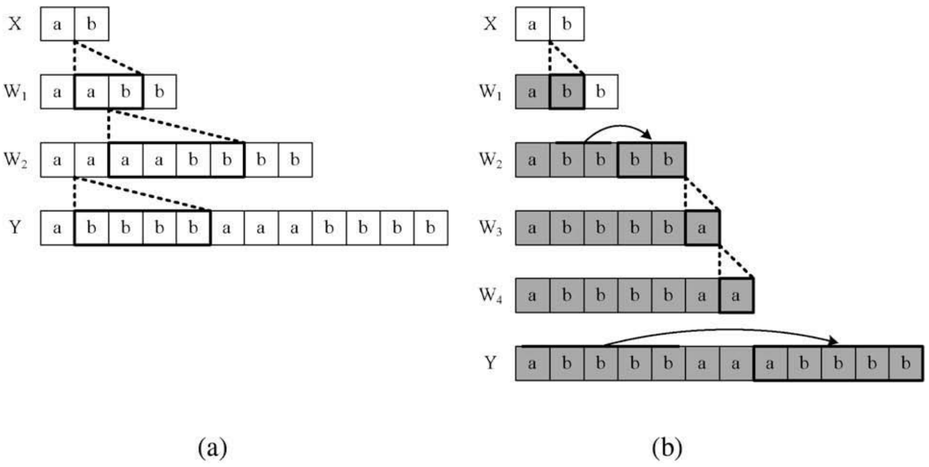

Fig. 2. Two examples to transform X to Y. (a) The recursive copies are allowed. (b) A restricted editing sequence defined by Ukkonen [18], where the white and gray cells represent the active and inactive parts, respectively.

operations cannot overlap and a series of edit operations are performed from left to right on the active parts of the intermediate strings. The inactive parts are fixed and cannot be changed any more. In fact, the active parts are always some suffixes of X and the inactive parts are always some prefixes of Y. Fig. 2 shows an example of the restricted editing sequences. If there is no restriction, the source string $X = ab$ can be transformed to the destination string $Y = abbbbaaabbbb$ with only three edit operations, as shown in Fig. 2(a). Fig. 2(b) shows how the destination string Y is formed by five edit operations with a restricted editing sequence. It can be seen that after the second edit operation is performed, the active parts of the following intermediate strings become empty.

The edit operations we consider in this paper are *character-insert*, *character-delete*, *block-copy* and *block-delete*. To specify the source where the blocks can be copied from, two models can be chosen, *external* and *internal*. As introduced in [14], the *external copy* indicates that the source of the block is the original string X and the *internal copy* indicates that the source of the block is the previous working string W_{i-1}. When the *block-copy* operations are combined with *shift* operations, one of the block linear-transformations shown in [13], the block edit distances would be more useful for many applications, such as searching on music databases. Here we denote $Z = z_1z_2 \cdots z_n$ as a shift string of $X = x_1x_2 \cdots x_n$ if $J_{x_i} - J_{z_i} = J_{x_j} - J_{z_j}$ for each $i, j \in [1, n]$, where J_{x_i} denotes the encoded index (in an arbitrarily given order) of character x_i in the alphabet Σ. One can apply this method to find out two identical melodies but with different pitches.

We adopt a set of attributes which can be composed and applied on block-edit operations to construct different versions of edit problems. For each edit problem, there are two kinds of attributes, one is for copy behaviors and the other is for cost measures. 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. The attributes are listed as follows:

Copy/Deletion Operations

External Copy (E): W_{i+1} is constructed out of copying a substring of X and inserting it into a valid position of the active part of W_i.

Internal Copy (I): W_{i+1} is constructed out of copying a substring of the inactive part of W_i and inserting it into a valid position of the active part of W_i.

Shifted Copy (S): W_{i+1} is constructed out of copying the shifted string S' of a given string S and inserting S' into a valid position of the active part of W_i. This attribute should be attached to the External and/or Internal attribute.

Deletion: W_{i+1} is constructed out of deleting a valid substring of the active part of W_i.

$$
\begin{aligned}
d(X_i, Y_j) &= \begin{cases} \infty, & \text{if } i < 0 \text{ or } j < 0 \\ 0, & \text{if } i = j = 0 \\ \min\left\{\begin{matrix} d_1(X_i, Y_j), d_2(X_i, Y_j), d_3(X_i, Y_j), \\ d_4(X_i, Y_j), d_5(X_i, Y_j), d_6(X_i, Y_j) \end{matrix}\right\}, & \text{otherwise} \end{cases} \\
d_1(X_i, Y_j) &= \begin{cases} d(X_{i-1}, Y_{j-1}), & \text{if } x_i = y_j \\ \min\{d(X_i, Y_{j-1}), d(X_{i-1}, Y_j)\} + 1, & \text{if } x_i \neq y_j \end{cases} \\
d_2(X_i, Y_j) &= \min\{d(X_{k-1}, Y_j) + p_{delete} \mid 1 \leqslant k \leqslant i\} \\
d_3(X_i, Y_j) &= \min\{d(X_i, Y_{k-1}) + p_{copy} \mid Y_{k\ldots j} \text{ is a substring of } X\} \\
d_4(X_i, Y_j) &= \min\{d(X_i, Y_{k-1}) + p_{copy} \mid Y_{k\ldots j} \text{ is a substring of } Y_{k-1}\} \\
d_5(X_i, Y_j) &= \min\{d(X_i, Y_{k-1}) + p_{shift\_copy} \mid Y_{k\ldots j} \text{ is a shifted substring of } X\} \\
d_6(X_i, Y_j) &= \min\{d(X_i, Y_{k-1}) + p_{shift\_copy} \mid Y_{k\ldots j} \text{ is a shifted substring of } Y_{k-1}\}
\end{aligned}
$$

Fig. 3. The recurrence formula for solving $P(EIS, C)$.

Cost Measures

Constant Cost (C): All copy (or deletion) operations are of the same cost p_{copy} (p_{delete}) as shown in [18].

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. This cost is similar to the *affine gap penalty* [21].

Nested Cost (N): All deletion operations are of the same cost p_{delete}, but the copied strings can be further edited with character-edit operations. Let s_1 denote the copied string and s_2 denote the string after editing, the cost of this copy operation is $p_{copy} + d_{tra}(s_1, s_2)$, as shown in [17].

With the combination of these attributes, one 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 block edit distance (cost) between them in various versions of edit problems.

3. Problems and algorithms

In this section, we introduce some problems with the attributes shown in the previous section and propose the corresponding algorithms.

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 Fig. 3. When calculating the value of $d(X_i, Y_j)$, we choose the minimum among $d_1(X_i, Y_j), d_2(X_i, Y_j), \ldots, d_6(X_i, Y_j)$, where $d_1(X_i, Y_j)$ denotes the minimal cost which ends with a character-edit operation, and the others denote the minimal costs which ends with different block-edit operations, respectively.

The time and space complexities of the straightforward DP algorithm are analyzed as follows. $O(nm)$ space is needed to store the values of $d(X_i, Y_j)$ for each i and j. To calculate $d_2(X_i, Y_j)$, $O(n)$ time is needed by linear minimum searching. To calculate $d_3(X_i, Y_j)$, for each suffix of Y_j, one can test if it is a substring of X in $O(n + m)$ time by the KMP algorithm [22]. 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 for $d_3(X_i, Y_j)$ is $O(m(n + m))$. In the equation of $d_4(X_i, Y_j)$, by using the same strategy to solve $d_3(X_i, Y_j)$, each suffix $Y_{k\ldots j}$ of Y_j can be easily tested if it is a substring of Y_{k-1}. The time needed for $d_4(X_i, Y_j)$ is $O(m^2)$. In the equations of $d_5(X_i, Y_j)$ and $d_6(X_i, Y_j)$, for a given bias β, by using the same strategy for solving $d_3(X_i, Y_j)$ and $d_4(X_i, Y_j)$, 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 needed for $d_5(X_i, Y_j)$ and $d_6(X_i, Y_j)$ 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 efficient algorithm to solve this problem. For $d_2(X_i, Y_j)$, since the deletion operations are of the same constant cost p_{delete}, the formula can be modified as $d_2(X_i, Y_j) = \min\{d(X_{k-1}, Y_j) \mid 1 \leqslant k \leqslant i\} + p_{delete}$. This is to say that we are finding the minimum of the elements $\{d(X_0, Y_j), d(X_1, Y_j), \ldots, d(X_{i-1}, Y_j)\}$. By preserving the current minimum for the next iteration, it needs only $O(1)$ time to find the new minimum for each iteration. Note that, if there are several locations of the same minimal value, they are all valid prefixes to achieve the optimal solution, and we can choose any of them.

For $d_3(X_i, Y_j)$, the following two steps are involved:

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

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

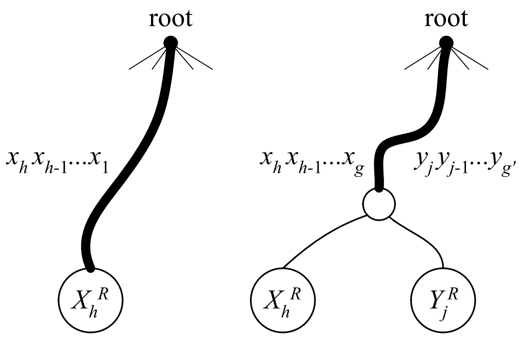

Fig. 4. The LCA of Y_j^R and X_h^R. $X_{g\ldots h}$ is the longest substring of X that matches a suffix of Y_j.

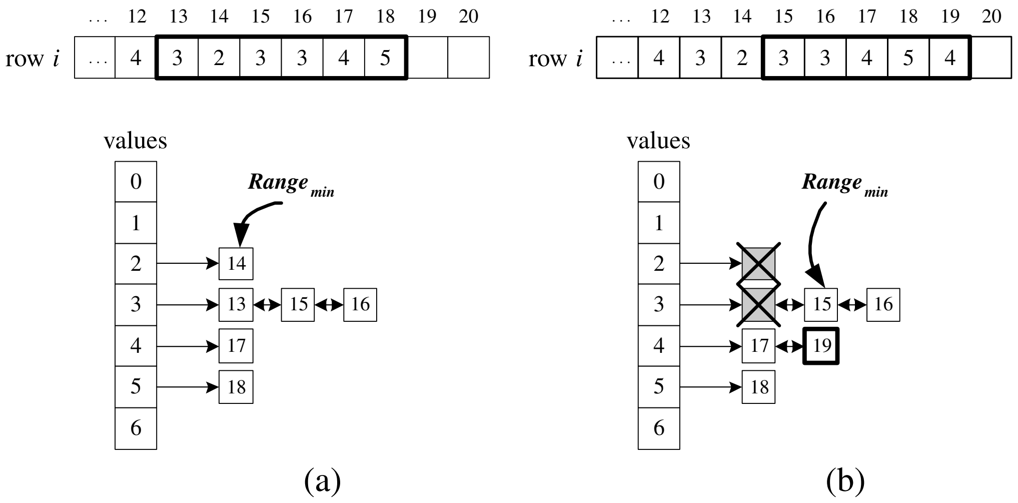

Fig. 5. A data structure for the range minimum query on compact integer data. (a) Finding the range minimum for calculating $d_3(X_i, Y_{19})$. (b) Updating the data structure for calculating $d_3(X_i, Y_{20})$.

In Step 1, we will find the position l such that $Y_{l\ldots j}$ is a substring of X, but $Y_{l-1\ldots j}$ is not, for $1 \leqslant l \leqslant j \leqslant m$, and l can be found in $O(1)$ time after the following preprocessing. First, a suffix tree [23] $T\left(X^R\#Y^R\$\right)$ is built, where $\{\#, \$\}$ are two dummy symbols which do not appear in Σ. On each internal node of the suffix tree, a flag is used to preserve the information where its descendant leaf nodes came from ($\{X^R, Y^R$, or both$\}$). Trace the path bottom-up from the leaf node Y_j^R to the root and find the first (deepest) internal node whose subtree contains some leaf nodes in X^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) $LCP\left(Y_j^R, X_h^R\right)$. However, this LCA cannot be found in constant time by the LCA query method shown in [24] because the leaf node X_h^R remains unknown before tracing the path from Y_j^R to the root. As shown in Fig. 4, the reverse string of $LCP\left(Y_j^R, X_h^R\right)$ 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 $j \in [1, m]$, one can record the location of LCA to those internal nodes on the searching path. In another query $j' \in [1, m]$, 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 position l in $O(1)$ time when Y_j is given.

In Step 2, we have to find $\min\{d(X_i, Y_{k-1}) + p_{copy} | l \leqslant k \leqslant q\}$ that $Y_{k\ldots j}$ is copied from X. This is to find the minimal value in the range from $d(X_i, Y_{l-1})$ to $d(X_i, Y_{j-1})$ in the DP lattice and then add it to p_{copy}. Here we assume that 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 strategy shown in the next section. This searching can be done by a special data structure shown in Fig. 5 which maintains the integer values of the current row (i.e. $d(X_i, Y_1)$ through $d(X_i, Y_m)$) 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 $d(X_i, Y_j)$ and $d(X_i, Y_{j-1})$ 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 query can be achieved in $O(1)$ time.

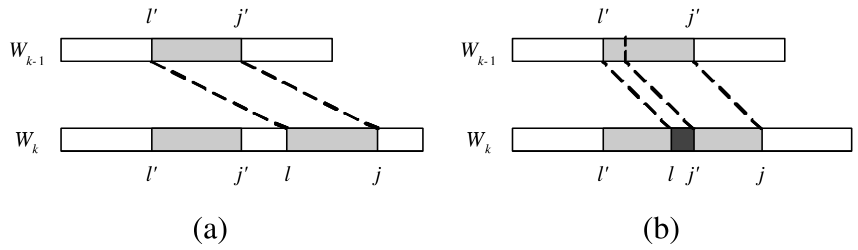

Fig. 6. Choosing the blocks for internal copy. (a) A valid copy of the longest matched suffix $Y_{l\ldots j}$ from $Y_{l'\ldots j'}$. (b) The longest valid suffix that can be copied is $Y_{j'+1\ldots j}$ due to the overlapping region on the working string.

Fig. 5(a) shows the way of computing the value of $d_3(X_i, Y_{19})$, where the searching range is assumed to be among $d(X_i, Y_{13}), d(X_i, Y_{14}), \ldots, d(X_i, Y_{18})$. The indices from 13 to 18 are stored in the linked lists corresponding to their values. For example, the index 15 is stored in the list of row 3 since $d(X_i, Y_{15}) = 3$. 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. Assuming that p_{copy} is 2 and $d_3(X_i, Y_{19})$ is the best cost among $d_1(X_i, Y_{19}), d_2(X_i, Y_{19}), \ldots, d_6(X_i, Y_{19})$, we can obtain that $d(X_i, Y_{19}) = d_3(X_i, Y_{19}) = d(X_i, Y_{14}) + 2 = 4$. After $d(X_i, Y_{19})$ is obtained, the index 19 is stored at the end of the linked list in row 4 since its value is 4. When the iteration $d_3(X_i, Y_{20})$ begins, the indices 13 and 14 which are outside the range for the new minimum searching will be removed, as shown in Fig. 5(b). Both the storing and the removing operation 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 at 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 Fig. 5(b) as an example, the pointer $Range_{min}$ is not changed when the index 13 is removed, but when the index 14 is removed, $Range_{min}$ is updated to point to the new minimum, the index 15. The maintenance of the pointer $Range_{min}$ on the data structure can be done in $O(1)$ time for each store and each removal. Note that there are at most m storing operations and m removing operations for computing the row $\{d_3(X_i, Y_1), d_3(X_i, Y_2), \ldots, d_3(X_i, Y_m)\}$, therefore, one can answer the best starting position k to be copied for Step 2 and computes $d_3(X_i, Y_j)$ in $O(1)$ amortized time for each iteration.

For $d_4(X_i, Y_j)$, similar to $d_3(X_i, Y_j)$, there are two steps involved:

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

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

In Step 1, we will find the position $l \in [1, j]$, such that $Y_{l\ldots j}$ is a substring of Y_{l-1}, but $Y_{l-1\ldots j}$ is not a substring of Y_{l-2}. Note that in the equation of $d_4(X_i, Y_j)$, when the dynamic programming is used to find the best suffix that can be copied, the overlapping regions must be avoided. As shown in Fig. 6(a), the longest suffix $Y_{l\ldots j}$ is valid and $Y_{k\ldots j}$ is a candidate block to be copied for each $k \in [l, j]$. But in Fig. 6(b), we cannot produce $Y_{l\ldots j}$ by the internal copy from the working string due to the overlapping region. Therefore, the longest *valid* suffix that can be copied is $Y_{j'+1\ldots j}$, other suffixes longer than $Y_{j'+1\ldots j}$ will be invalid. 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, as shown in Fig. 7. If two or more substrings of Y can be copied to $Y_{l\ldots j}$, the one with the smaller index is always better because it will make the overlapping region smaller. To find the longest valid suffix $Y_{l\ldots j}$ that can be copied, we should consider the internal nodes on the path from the root to the leaf node Y_j^R. Let $V = \{v_1, v_2, \ldots, v_p\}$ denote the set of internal nodes on the path and $\left\{Y_{j_1}^R, Y_{j_2}^R, \ldots, Y_{j_p}^R\right\}$ denote the set of corresponding leaf nodes pointed by $\{v_1, v_2, \ldots, v_p\}$. It can be seen that the candidate blocks in Y_j to be copied begin with the locations $j_1, j_2, \ldots, j_p$, respectively. Considering an internal node $v_q \in V$ and its corresponding copying location Y_{j_q}, if there exists no overlapping region, $\left|LCP\left(Y_{j_q}^R, Y_j^R\right)\right|$ is the length of the block that can be copied. On the other hand, if there exists an overlapping region, the length of the block that can be copied is $(j - j_q)$, as shown in Fig. 6(b). In the worst case, the length of the path is $O(m)$, however, Fayolle and Ward [25] shown that the expected depth of the suffix tree $T(Y^R)$ is $O(\log m)$. For a given $j \in [1, m]$, one can find the longest valid suffix of Y_j that can be copied in $O(m)$ time in the worst case and $O(\log m)$ time on average. Thus, the preprocessing time becomes $O(m^2)$ time in the worst case and $O(m \log m)$ time on average.

In Step 2, one can answer the best starting position k to be copied, by using the same strategy to solve $d_3(X_i, Y_j)$, in $O(1)$ amortized time. The overall amortized answering time for the equation of $d_4(X_i, Y_j)$ 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 $X' = x'_1x'_2 \cdots x'_{n-1}$ and $Y' = y'_1y'_2 \cdots y'_{m-1}$, respectively, where $x'_i = J_{x_{i+1}} - J_{x_i}, 1 \leqslant i \leqslant n-1$ and $y'_j = J_{y_{j+1}} - J_{y_j}, 1 \leqslant j \leqslant m-1$. For $d_5(X_i, Y_j)$, the strategy for solving $d_3(X_i, Y_j)$ can be applied similarly by preprocessing the suffix tree $T\left(X'^R\#Y'^R\$\right)$, rather than $T\left(X^R\#Y^R\$\right)$. Then the preprocessing time is still $O(n + m)$ and the amortized answering

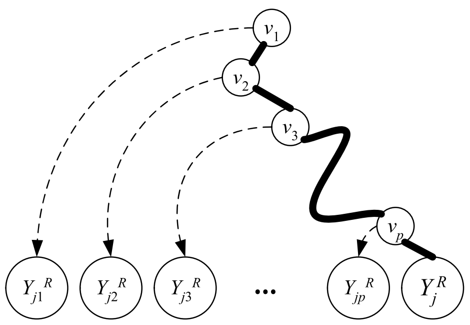

Fig. 7. The suffix tree $T(Y_j)$ for finding the longest valid suffix $Y_{l...j}$ to be copied.

$$\begin{aligned}
d(X_i, Y_j) &= \begin{cases} \infty, & \text{if } i < 0 \text{ or } j < 0 \\ 0, & \text{if } i = j = 0 \\ \min\{d_1(X_i, Y_j), d_2(X_i, Y_j), d_3(X_i, Y_j), d_4(X_i, Y_j)\}, & \text{otherwise} \end{cases} \\
d_1(X_i, Y_j) &= \begin{cases} d(X_{i-1}, Y_{j-1}), & \text{if } x_i = y_j \\ \min\{d(X_i, Y_{j-1}), d(X_{i-1}, Y_j)\} + 1, & \text{if } x_i \neq y_j \end{cases} \\
d_2(X_i, Y_j) &= \min\{d(X_{k-1}, Y_j) + p_s + (i - k + 1)p_e \mid 1 \leqslant k \leqslant i\} \\
d_3(X_i, Y_j) &= \min\{d(X_i, Y_{k-1}) + p_s + (j - k + 1)p_e \mid Y_{k...j} \text{ is a substring of } X\} \\
d_4(X_i, Y_j) &= \min\{d(X_i, Y_{k-1}) + p_s + (j - k + 1)p_e \mid Y_{k...j} \text{ is a substring of } Y_{k-1}\}
\end{aligned}$$

Fig. 8. The recurrence formula for solving $P(EI, L)$.

time is still $O(1)$ per iteration. For $d_6(X_i, Y_j)$, the strategy for solving $d_4(X_i, Y_j)$ 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 by a dynamic programming algorithm in $O(nm)$ time with $O(n + m^2)$ preprocessing time in the worst cast and $O(n + m \log m)$ preprocessing time on average.*

3.2. Problem 2 – $P(EI, L)$

Fig. 8 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.

To solve this problem, we may use a strategy similar to which solves $P(EIS, C)$. Note that this problem cannot be solved by the algorithm for $P(EIS, C)$ directly because of two key differences. 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 Fig. 5 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 $d(X_i, Y_j)$ passes to the next iteration $d(X_i, Y_{j+1})$, all the values in the searching range are increased with the cost p_e except the newly inserted element $d(X_i, Y_j)$. To avoid updating all values stored in the balanced binary search tree, we subtract the corresponding amount of p_e from the newly inserted element, rather than add p_e to all the stored elements. The remaining part of the algorithm for $P(EIS, C)$, such as the preprocessing on suffix trees, is still workable for this problem.

The time required by this algorithm is analyzed as follows. For $d_2(X_i, Y_j)$, finding the best suffix of X_i to be deleted can still be done in $O(1)$ time by preserving the current minimum which adds the corresponding amount of p_e for the next iteration. For $d_3(X_i, Y_j)$ and $d_4(X_i, Y_j)$, the preprocessing for constructing the suffix trees and finding the longest valid suffixes that can be copied to still requires $O(n + m)$ and $O(m^2)$ time, respectively. However, the amortized time needed for answering $d_3(X_i, Y_j)$ and $d_4(X_i, Y_j)$ is both increased to $O(\log m)$ per iteration because the values in the searching range are stored in a balanced binary search tree.

As a summary, we have the following theorem.

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

$$
\begin{aligned}
d(X_i, Y_j) &= \begin{cases} \infty, & \text{if } i < 0 \text{ or } j < 0 \\ 0, & \text{if } i = j = 0 \\ \min\{d_1(X_i, Y_j), d_2(X_i, Y_j), d_3(X_i, Y_j), d_4(X_i, Y_j)\}, & \text{otherwise} \end{cases} \\
d_1(X_i, Y_j) &= \begin{cases} d(X_{i-1}, Y_{j-1}), & \text{if } x_i = y_j \\ \min\{d(X_i, Y_{j-1}), d(X_{i-1}, Y_j)\} + 1, & \text{if } x_i \neq y_j \end{cases} \\
d_2(X_i, Y_j) &= \min\{d(X_{k-1}, Y_j) + p_{delete} \mid 1 \leqslant k \leqslant i\} \\
d_3(X_i, Y_j) &= \min\{d(X_i, Y_{k-1}) + d_{sub}(X, Y_{k\ldots j}) + p_{copy} \mid 1 \leqslant k \leqslant j\} \\
d_4(X_i, Y_j) &= \min\{d(X_i, Y_{k-1}) + d_{sub}(Y_{k-1}, Y_{k\ldots j}) + p_{copy} \mid 1 \leqslant k \leqslant j\}
\end{aligned}
$$

Fig. 9. The recurrence formula for solving $P(EI, N)$.

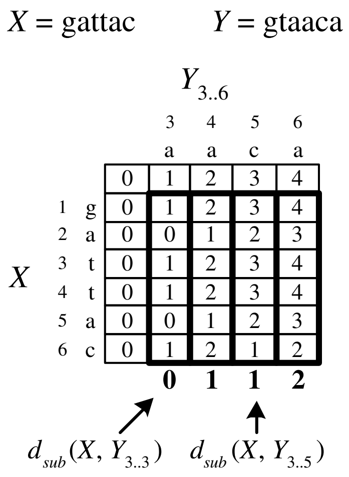

Fig. 10. The DP lattice for finding the substring edit distances $d_{sub}(X, Y_{3\ldots3})$, $d_{sub}(X, Y_{3\ldots4})$, $d_{sub}(X, Y_{3\ldots5})$ and $d_{sub}(X, Y_{3\ldots6})$.

3.3. Problem 3 – $P(EI, N)$

The recurrence formula for solving $P(EI, N)$ is given in Fig. 9. A straightforward implementation requires $O(n^2m^3)$ time. We propose a more efficient algorithm as follows.

For $d_2(X_i, Y_j)$, p_{delete} is a constant, and the strategy for $P(EIS, C)$ can be applied similarly, thus it can be done in $O(1)$ time per iteration. For $d_3(X_i, Y_j)$, we want to find out the position k such that $d(X_i, Y_k) + d_{sub}(X, Y_{k\ldots j})$ is minimal, and this requires all the substring edit distance $d_{sub}(X, Y_{k\ldots j})$, $1 \leqslant k \leqslant j \leqslant m$. This can be done by preparing the DP lattice of X and $Y_{k\ldots m}$ for each $k \in [1, m]$, and hence m DP lattices are generated. For example, Fig. 10 shows a DP lattice of X and $Y_{k\ldots m}$, where $k = 3$ and $m = 6$. By computing the minimum value of column k', one can get the substring edit distance $d_{sub}(X, Y_{k\ldots k'})$. The construction of the m DP lattices needs $O(nm^2)$ preprocessing time. Then, we need to find out k such that $d(X_i, Y_{k-1}) + d_{sub}(X, Y_{k\ldots j})$ is minimal, which needs $O(m)$ time per iteration. For $d_4(X_i, Y_j)$, each substring edit distance $d_{sub}(Y_{k-1}, Y_{k\ldots j})$ is required, $1 \leqslant k \leqslant j \leqslant m$. Similarly, this can be done by preparing the DP lattice of Y_{k-1} and $Y_{k\ldots m}$ for each $k \in [1, m]$. The construction of the m DP lattices needs $O(m^3)$ preprocessing time. Then, we can find out the position k such that $d(X_i, Y_{k-1}) + d_{sub}(Y_{k-1}, Y_{k\ldots j})$ 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 by a dynamic programming algorithm in $O(nm^2)$ time with $O((n + m)m^2)$ preprocessing time.*

4. Conclusion

Most previous research on the block-edit operations focused on approximations rather than optimal solutions because of the NP-completeness. In this paper, we show that by applying some slight and reasonable restrictions, the optimal solutions can be obtained and it is very practical. For example, we solve the $P(EIS, C)$ problem in $O(nm)$ time, and it is useful when it is compared to the traditional edit distances with character-edit operations. Besides, we introduce the concept of attaching the attributes to form various problems which are suitable for different scenarios. The remaining problems are solved as follows. 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

Table 1
Summary of the three problems and our methods.

	Straightforward DP	Our methods		
		Modified DP	Preprocessing	Total
$P(EIS, C)$	$O(nm^2(n+m)\|\Sigma\|)$	$O(nm)$	$O(n+m^2)$ in worst case $O(n+m\log m)$ in average case	$O(nm+m^2)$
$P(EI, L)$	$O(nm^2(n+m))$	$O(nm\log m)$	$O(n+m^2)$ in worst case $O(n+m\log m)$ in average case	$O(nm\log m+m^2)$
$P(EI, N)$	$O(n^2m^3)$	$O(nm^2)$	$O((n+m)m^2)$	$O((n+m)m^2)$

is better fitted to the human natural edit behavior, and it can be solved in $O(nm^2)$ time. The summary of the three problems and our methods is shown in Table 1.

As shown in Section 3.1, the problems which are attached with shifted copies of constant cost can be solved easily by preprocessing the differential strings and the suffix trees. This strategy can also be applied when the shifted copies are of linear cost. However, it becomes troublesome when the copied strings can be further edited with character-edit operations, i.e. nested cost. The preprocessing of the differential strings and the suffix trees is not sufficient to cope with the nested cost behavior. These are why the shift operation is included in neither Problem 2 nor Problem 3, because they are either too easy or too difficult, respectively.

In the future, we are interested in how to solve the $P(EIS, N)$ problem which includes shift operation with the same time complexity of the $P(EI, N)$ problem. We are also interested in whether the $P(EI, N)$ problem can be solved by more efficient algorithms to make it more practical.

Acknowledgments

We thank the anonymous referees for their careful work and helpful references.

References

[1] V. Freschi, A. Bogliolo, Longest common subsequence between run-length-encoded strings: a new algorithm with improved parallelism, Information Processing Letters, 90 (2004) 167–173.
[2] D.S. Hirschberg, Algorithms for the longest common subsequence problem, Journal of the ACM 24 (4) (1977) 664–675.
[3] K.S. Huang, C.B. Yang, K.T. Tseng, Y.H. Peng, H.Y. Ann, Dynamic programming algorithms for the mosaic longest common subsequence problem, Information Processing Letters 102 (2007) 99–103.
[4] K.S. Huang, C.B. Yang, K.T. Tseng, H.Y. Ann, Y.H. Peng, Efficient algorithms for finding interleaving relationship between sequences, Information Processing Letters 105 (5) (2008) 188–193.
[5] J.W. Hunt, T.G. Szymanski, A fast algorithm for computing longest common subsequences, Communications of the ACM 20 (5) (1977) 350–353.
[6] C. Rick, Simple and fast linear space computation of longest common subsequences, Information Processing Letters 75 (6) (2000) 275–281.
[7] R. Wagner, M. Fischer, The string-to-string correction problem, Journal of the ACM 21 (1) (1974) 168–173.
[8] C.B. Yang, R.C.T. Lee, Systolic algorithm for the longest common subsequence problem, Journal of the Chinese Institute of Engineers 10 (6) (1987) 691–699.
[9] R.W. Hamming, Error detecting and error correcting codes, Bell System Technical Journal 26 (2) (1950) 147–160.
[10] D. Shapira, J.A. Storer, Edit distance with move operations, in: Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching, vol. 2373, 2002, pp. 85–98.
[11] M. Chrobak, P. Kolman, J. Sgall, The greedy algorithm for the minimum common string partition problem, ACM Transactions on Algorithms 1 (2) (2005) 350–366.
[12] H. Kaplan, N. Shafrir, The greedy algorithm for edit distance with moves, Information Processing Letters 97 (1) (2006) 23–27.
[13] S. Muthukrishnan, S.C. Sahinalp, Approximate nearest neighbors and sequence comparison with block operations, in: Proceedings of the Symposium on the Theory of Computing (STOC 2000), 2000, pp. 416–424.
[14] F. Ergun, S. Muthukrishnan, C. Sahinalp, Comparing sequences with segment rearrangements, in: Proceedings of the FSTTCS'03, 2003, pp. 183–194.
[15] D. Shapira, J.A. Storer, Large edit distance with multiple block operations, in: String Processing and Information Retrieval (SPIRE), vol. 2857, 2003, pp. 369–377.
[16] D. Shapira, J.A. Storer, Edit distance with move operations, Journal of Discrete Algorithms 5 (2) (2007) 380–392.
[17] D. Lopresti, A. Tomkins, Block edit models for approximate string matching, Theoretical Computer Science 181 (1997) 159–179.
[18] E. Ukkonen, Algorithms for approximate string matching, Information and Control 64 (1985) 100–118.
[19] R. Cole, R. Hariharan, Approximate string matching: a simpler faster algorithm, in: Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, 1998, pp. 463–472.
[20] G.M. Landau, U. Vishkin, Fast parallel and serial approximate string matching, Journal of Algorithms 10 (2) (1989) 157–169.
[21] O. Gotoh, An improved algorithm for matching biological sequences, Journal of Molecular Biology 162 (3) (1982) 705–708.
[22] D.E. Knuth, J.H. Morris Jr., V. Pratt, Fast pattern matching in strings, SIAM Journal on Computing 6 (2) (1977) 323–350.
[23] P. Weiner, Linear pattern matching algorithm, in: Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.
[24] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, Cambridge, 1997..
[25] J. Fayolle, M.D. Ward, Analysis of the average depth in a suffix tree under a markov model, in: International Conference on Analysis of Algorithms DMTCS Proceedings of the AD, 2005, pp. 95–104.