Information Processing Letters 108 (2008) 360–364

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

A fast and simple algorithm for computing the longest common subsequence of run-length encoded strings

Hsing-Yen Ann, Chang-Biau Yang *, Chiou-Ting Tseng, Chiou-Yi Hor

Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung 80424, Taiwan

ARTICLE INFO

Article history:
Available online 16 July 2008
Communicated by L. Boasson

Keywords:
Design of algorithms
Longest common subsequence
Run-length encoding
Edit distance

ABSTRACT

Let X and Y be two strings of lengths n and m, respectively, and k and l, respectively, be the numbers of runs in their corresponding run-length encoded forms. We propose a simple algorithm for computing the longest common subsequence of two given strings X and Y in $O(kl + \min\{p_1, p_2\})$ time, where p_1 and p_2 denote the numbers of elements in the bottom and right boundaries of the matched blocks, respectively. It improves the previously known time bound $O(\min\{nl, km\})$ and outperforms the time bounds $O(kl \log kl)$ or $O((k + l + q) \log(k + l + q))$ for some cases, where q denotes the number of matched blocks.

© 2008 Elsevier B.V. All rights reserved.

1. Introduction

Measuring the similarity of two strings is an important fundamental in many applications, such as pattern matching and computational biology. Two well-known measurements for computing the similarity of two strings, *longest common subsequence* (LCS) and *edit distance*, have been extensively studied for the past decades. Given a sequence or string, a *subsequence* is formed by deleting zero or more elements arbitrarily. The LCS problem measures the length of the longest subsequence which is contained in the both given sequences. The edit distance problem, or more precisely, the *Levenshtein distance* problem [9], measures the minimal number of edit steps which transform a given sequence into another one. The edit operations involved in the Levenshtein distance are *insertion*, *deletion* and *substitution*.

A special edit distance problem stipulates that one substitution is replaced by one deletion plus one insertion. A simple formula can do the transformation between the LCS lengths and the special edit distances [5]. Both the LCS problem and the Levenshtein distance problem can be solved by using the dynamic programming technique in $O(nm)$ time, where n and m denote the lengths of the two given strings [15]. Some improved algorithms have also been proposed, they include reducing the required space, considering smaller number of matching elements, using the parallelism and considering other important properties [7,8,13,16].

Run-length encoding (RLE) is a well-known method for compressing strings. It considers one string as a sequence of runs where each run consists of identical symbols and it is represented by its symbol and length. For example, a string *aaaddbbbbbbcccc* is encoded as $a^3d^2b^6c^4$ in the RLE format. A famous application for RLE is the optical character recognition (OCR) system, in which binary alphabet is used and the RLE usually achieves good compression ratio. It is more useful and efficient if we can measure the similarity of two RLE strings without uncompressing them.

Several related works for computing the LCS of RLE strings have been proposed. Bunke and Csirik [4] illustrated the concept that splits the *dynamic programming* (DP) lattice into blocks, where each block corresponds to a pair of runs in the two given strings. Their algorithm computes the LCS in $O(nl + km)$ time by considering the elements on the boundaries of the blocks only, where k and l denote the run numbers of the two given strings X and Y, respectively, and $|X| = n$ and $|Y| = m$. Apostolico et al. [1] proposed another algorithm with $O(kl \log kl)$ time

* Corresponding author.
E-mail address: cbyang@cse.nsysu.edu.tw (C.-B. Yang).

0020-0190/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.ipl.2008.07.005

by maintaining a collection of forced paths in the balanced binary search trees. Mitchell [12] reduced the LCS problem to the geometric shortest path problem and proposed an algorithm with $O((k+l+q)\log(k+l+q))$ time, where q denotes the number of matched blocks. Recently, Liu et al. [11] proposed an algorithm with $O(\min\{nl, km\})$ time to compute the LCS between an RLE string and an uncompressed string. Freschi and Bogliolo [5] proposed a parallel algorithm which computes the LCS in $O(n+m)$ steps on a systolic array of $k+l$ units. For another similar problem, computing the Levenshtein distance of two RLE strings, Arbell et al. [2] proposed an algorithm with $O(nl+km)$ time, while Liu et al. [10] proposed an algorithm with $O(\min\{nl, km\})$ time.

In this paper, we consider the computation of the LCS of two RLE strings. Some definitions and preliminaries are given in Section 2. In Section 3, we propose an algorithm and give the analysis of time complexity. And finally, the conclusion goes in Section 4.

2. Preliminaries

Consider two strings $X = x_1x_2\ldots x_n$ and $Y = y_1y_2\ldots y_m$ over finite alphabet Σ. The RLE string of X is denoted as $RX_1RX_2\ldots RX_k$, where all symbols in RX_i are the same and the symbols in RX_i and RX_{i+1}, $1 \leqslant i \leqslant k-1$, are different. The lengths of the runs $RX_1, RX_2, \ldots, RX_k$ are denoted by $n_1, n_2, \ldots, n_k$, respectively. A substring $x_ix_{i+1}\ldots x_j$ of X is denoted as $X_{i..j}$. A substring $RX_{i..j}$ of X consists of the consecutive runs $RX_i, RX_{i+1}, \ldots, RX_j$. Similarly, the RLE string of Y is denoted as $RY_1RY_2\ldots RY_l$, whose lengths are denoted by $m_1, m_2, \ldots, m_l$, respectively. The length of the LCS of X and Y is denoted as $LCS(X, Y)$.

2.1. The blocks

The original DP lattice can be divided into $k \times l$ sublattices, called *blocks* [4]. A block (i, j) is *dark* if the symbols in RX_i and RY_j are identical; otherwise, it is called a *light* block. Fig. 1 shows an example of dividing the blocks. The sets of elements on the bottom boundary and right boundary of a block are called the bottom *wall* and the right *wall* of the block, respectively. Note that the element at the lower right corner belongs to both right and bottom walls. When computing the LCS, similar to the original lattice, we use a two-dimensional lattice M to record the LCS lengths of the elements in the lower right corners of the blocks. For example, the element $M[i, j]$ holds the value of $LCS(RX_{1..i}, RY_{1..j})$. Another three-dimensional lattice W is used for holding the bottom walls, where $W[i, j, r]$ holds the value of the rth element on the bottom wall of block (i, j). Logically, W is a three-dimensional lattice, but physically, it is mapped to the original two-dimensional DP lattice. We denote $\alpha(j, r)$ as the original one-dimensional index in Y which is transformed from the two-dimensional pair (j, r), the rth element of run RY_j. It is easy to see that $\alpha(j, r) = \sum_{p=1}^{j-1} m_p + r$. This query can be answered in $O(1)$ time after the preprocessing on Y with $O(m)$ time.

In the original DP lattice, each monotonically nondecreasing path from $(0, 0)$ to (n, m) is mapped to a corresponding common subsequence. A path is called *forced* [1] if it traverses the dark blocks by strictly diagonal moves and traverses the light blocks by strictly horizontal (or vertical, respectively) moves. As shown in Fig. 1, the sequence of solid lines represents a forced path which starts at the star mark u_0 and ends at the square mark v_0 by traversing three dark blocks diagonally and three light blocks horizontally.

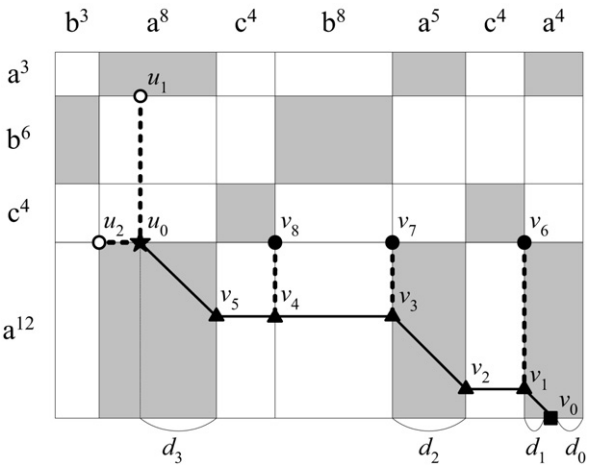

Fig. 1. An example for illustrating our idea.

The following lemmas show some properties of the LCS problem on RLE strings.

Lemma 1 *(Dark block). (See [5].) Let X and Y be two strings and a be one symbol. Let Xa^r and Ya^r denote the strings which are constructed by appending the run a^r to the strings X and Y, respectively. Then $LCS(Xa^r, Ya^r) = LCS(X, Y) + r$.*

Lemma 2 *(Light block). (See [5].) Let X and Y be two strings, and a and b be two distinct symbols. Let Xa^r and Yb^s denote the strings which are constructed by appending the run a^r to X and appending the run b^s to Y, respectively. Then $LCS(Xa^r, Yb^s) = \max\{LCS(Xa^r, Y), LCS(X, Yb^s)\}$.*

Lemma 3 *(Monotonicity). (See [5].) Let $X_{1..i}$ and $Y_{1..j}$ be two strings. Then $LCS(X_{1..i'}, Y_{1..j'}) \leqslant LCS(X_{1..i}, Y_{1..j})$ for each $i' \in [1, i]$ and $j' \in [1, j]$.*

By applying Lemma 2 recursively and then removing the unnecessary elements according to Lemma 3, the following corollary can be derived.

Corollary 1 *(Merged light blocks). Let $Xa_1^{r_1}a_2^{r_2}\ldots a_i^{r_i}$ and $Yb_1^{s_1}b_2^{s_2}\ldots b_j^{s_j}$ be two strings, and $a_{i'} \neq b_{j'}$ for each $i' \in [1, i]$ and $j' \in [1, j]$. Then*

$$LCS\left(Xa_1^{r_1}a_2^{r_2}\ldots a_i^{r_i}, Yb_1^{s_1}b_2^{s_2}\ldots b_j^{s_j}\right) = \max\left\{LCS\left(Xa_1^{r_1}a_2^{r_2}\ldots a_i^{r_i}, Y\right), LCS\left(X, Yb_1^{s_1}b_2^{s_2}\ldots b_j^{s_j}\right)\right\}.$$

2.2. The Range Minimal Query (RMQ) problem

Given an array of real numbers $A[1..n]$, the *range minimal* (*maximal*) *query* (RMQ) problem is to answer the con-

sequent queries on intervals. Given an interval $[i_1, i_2]$, $1 \leqslant i_1 \leqslant i_2 \leqslant n$, an RMQ asks to return the index of the minimal element in the interval $A[i_1, i_2]$. Straightforwardly, one can answer each query in $O(n)$ time without any preprocessing. On the other hand, a naive algorithm can achieve $O(n^2)$ preprocessing time and $O(1)$ answering time. The goal is to make both the preprocessing and answering as efficient as possible. Bender and Farach-Colton [3] proposed an algorithm which reduces the RMQ problem to the *lowest common ancestor* (LCA) problem [6] on the *Cartesian tree* [14] of array A. Their algorithm achieves $O(n)$ preprocessing time and $O(1)$ answering time. We denote $RMQ_{\min}(A, i_1, i_2)$ and $\mathrm{RMQ}_{\max}(A, i_1, i_2)$ as the minimal value and maximal value in the interval $A[i_1, i_2]$, respectively.

Theorem 1. *(See [3].) Given an array A of n elements, one can preprocess A in $O(n)$ time such that for any given interval $[i_1, i_2]$, one can determine $RMQ_{\min}(A, i_1, i_2)$ and $RMQ_{\max}(A, i_1, i_2)$ in $O(1)$ time.*

3. Our algorithms

Before starting to explain our algorithm, we take Fig. 1 as an example to illustrate the idea of our algorithm.

3.1. The basic idea

Let $X = a^3b^6c^4a^{12}$ and $Y = b^3a^8c^4b^8a^5c^4a^4$ be two RLE strings. We show how to calculate the value of element v_0, i.e. $W[4, 7, 2]$, with the following recursive steps. To calculate v_0, only v_1 is needed to consider according to Lemma 1, that is $v_0 = v_1 + d_1$. To calculate v_1, two values v_2 and v_6 are needed according to Lemma 2, that is $v_1 = \max\{v_2, v_6\}$. By expanding the values of the remaining elements on the forced path between u_0 and v_0, the following equations are obtained. $v_2 = v_3 + d_2$, $v_3 = \max\{v_4, v_7\}$, $v_4 = \max\{v_5, v_8\}$ and $v_5 = u_0 + d_3$.

By combining these equations and removing some variables, we can get a clearer equation $v_0 = \max\{v_6 + d_1, v_7 + (d_1 + d_2), v_8 + (d_1 + d_2), u_0 + (d_1 + d_2 + d_3)\}$. Observing this equation, we can see that $\{v_6, v_7, v_8\}$ are the lower right corners of the blocks, and u_0 is located at either the bottom wall of a light block or the lower right corner of a dark block. To calculate u_0 of the first case, we only consider two values u_1 and u_2 according to Corollary 1, where u_1 is at the bottom wall of the dark block by a vertical leap through the light blocks and u_2 is also a lower right corner of a block. Summarizing the above observation, the elements needed to be calculated in the original DP lattice include the lower right corners of all blocks and the bottom walls of the dark blocks.

3.2. The algorithm

Our algorithm for computing the LCS length of two RLE strings is given in Algorithm 1. However, tracing out the resulting sequence is also easy if an additional pointer is attached to each element of the DP lattices M and W after the value of this element is computed. As shown in Algorithm 1, the DP lattices M and W are computed row by row. If block (i, j) is light, we can get the value of $M[i, j]$ by finding the maximum of $M[i-1, j]$ and $M[i, j-1]$ immediately according to Lemma 2. On the other hand, if block (i, j) is dark, we have to compute all values on the bottom wall of block (i, j) besides the lower right corner. The function *ComputeWall*(i, j) shown in Algorithm 2 is used for this computation.

Algorithm 1. *LCS*(X, Y)

```
Initialize the boundaries of the DP lattices M, W
Do preprocessing for P
for i = 1 to k do
    Do preprocessing for C_i, L_i, H_i
    for j = 1 to l do
        if block (i, j) is a light block then
            M[i, j] ⇐ max{M[i − 1, j], M[i, j − 1]}
        else
            ComputeWall(i, j)
            M[i, j] ⇐ W[i, j, m_j]
        end if
    end for
end for
return M[k, l]
```

Algorithm 2. *ComputeWall*(i, j)

```
for r = 1 to m_j do
    if forced path does not across row (i − 1) then
        W[i, j, r] ⇐ RMQ_max(L_i, 0, j − 1) − C_i[j, r]
    else
        i' ⇐ P[i]
        (j', r') ⇐ H_i[j, r]
        β1 ⇐ RMQ_max(L_i, j', j − 1) − C_i[j, r]
        if (i − 1, j', r') is a lower right corner of a block then
            β2 ⇐ M[i − 1, j'] + n_i
        else
            β2 ⇐ max{M[i − 1, j' − 1], W[i', j', r']} + n_i
        end if
        W[i, j, r] ⇐ max{β1, β2}
    end if
end for
```

In Algorithm 2, to calculate element (i, j, r) at the bottom wall of block (i, j), some additional information is needed. Let σ_i denote the symbol of run RX_i. We define an array P where $P[i]$ points out the index of the former run of RX_i which contains the same symbol σ_i. An array C_i is prepared for each row i where $C_i[j, r]$ stores the number of occurrences of σ_i in the suffix $Y_{\alpha(j,r)..m}$ of Y. Another array L_i is prepared and it contains the values of the lower right corners of the blocks in row $(i-1)$ plus the number of occurrences of σ_i between the corners to the right end of Y. That is, $L_i[j] = M[i-1, j] + C_i[j, m_j]$ for each $j \in [0, l]$. When computing the value $W[i, j, r]$, we have to consider the forced path that ends with (i, j, r). Fig. 1 shows an example to trace along the forced path. We have to find out the location on row $(i-1)$ where the forced path is across on, and we denote the found element as $(i-1, j', r')$. An array H_i is prepared where $H_i[j, r]$ points out the head of the forced path corresponding to (i, j, r), we may say that $H_i[j, r] = (j', r')$.

If the forced path is never across on row $(i-1)$, we ignore the starting element of the forced path since its

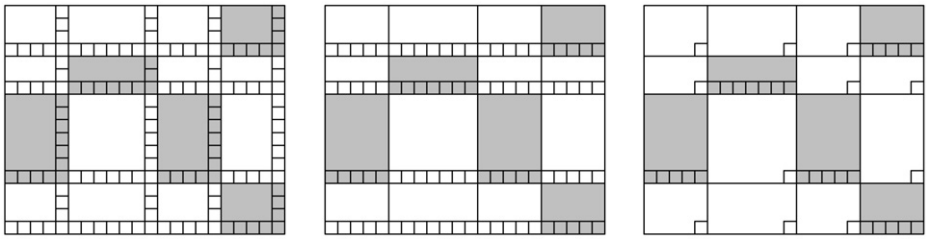

Fig. 2. An example for illustrating the time complexities. (a) $O(nl+km)$. (b) $O(km)$. (c) $O(kl+p_1)$.

value is equal to $M[i-1,0]$, which is 0, and the value $W[i,j,r]$ can be determined by considering the maximum of the values $\{L_i[0], L_i[1], \ldots, L_i[j-1]\}$, each representing the length of the corresponding common subsequence. Note that, after the maximum is found, the number of occurrences of σ_i in the range $[\alpha(j,r),m]$ of Y should be subtracted. On the other hand, if the starting element $(i-1,j',r')$ exists, its value must also be considered. If the starting element is located at the lower right corner of a block, its value has been computed and stored in M. Otherwise, according to Corollary 1, this value can be determined by finding the maximum of two values $M[i-1,j'-1]$ and $W[i',j',r']$. The temporary variables β_1 and β_2 are used to store the best lengths of these two kinds of common subsequences, respectively, and we can get the LCS length which ends at (i,j,r) among them.

3.3. How fast is it

As illustrated previously, the elements which have to be calculated in the original DP lattice include the lower right corners of all blocks and the bottom walls of the dark blocks, and each of the values can be calculated in $O(1)$ time. Let p_1 and p_2 denote the numbers of elements on the bottom and right boundaries of the dark blocks. Before the algorithm starts, we can precalculate the two values p_1 and p_2 to determine which of the input strings is considered as string X. Note that, although a logical three-dimensional array is used for storing the data in our algorithm, this array can be easily mapped to the compact one-dimensional memory space with excluding the unused elements. After the DP lattices M and W are computed, the resulting sequence can be traced out in $O(k+l)$ time according to the additional pointers. We can get the following theorem.

Theorem 2. *There exists an algorithm for computing the LCS of two given RLE strings X and Y in $O(kl+\min\{p_1,p_2\})$ time.*

Now we compare the time complexity of our algorithm with some previously proposed algorithms, $O(nl+km)$ [4], $O(\min\{nl,km\})$ [11], $O(kl\log kl)$ [1] and $O((k+l+q)\cdot\log(k+l+q))$ [12], where q denotes the number of matched blocks. Fig. 2 shows the comparison among the time complexities $O(nl+km)$, $O(km)$ and $O(kl+p_1)$. It shows that our algorithm outperforms the time complexities $O(nl+km)$ and $O(\min\{nl,km\})$ for all cases. If the number of blocks is large, our algorithm outperforms the time complexity $O(kl\log kl)$. If there are many small dark blocks, our algorithm outperforms the time complexity $O((k+l+q)\log(k+l+q))$.

4. Conclusion

In this paper, we propose a simple algorithm for computing the LCS of two given RLE strings. This algorithm adopts the subset of the elements in the original DP lattices. We improve the previously known time bound $O(\min\{nl,km\})$ and outperform the time bounds $O(kl\log kl)$ or $O((k+l+q)\log(k+l+q))$ for some cases. When considering the special edit distance problem in which one substitution is replaced by one deletion plus one insertion, we may first compute the LCS length and then transform to the edit distance [5]. It is an open question if there exists an algorithm for solving Levenshtein distance problem in the same time complexity $O(kl+\min\{p_1,p_2\})$, which outperforms the best known time bound $O(\min\{nl,km\})$ [10].

References

[1] A. Apostolico, G.M. Landau, S. Skiena, Matching for run-length encoded strings, Journal of Complexity 15 (1) (1999) 4–16.
[2] O. Arbell, G.M. Landau, J.S.B. Mitchell, Edit distance of run-length encoded strings, Information Processing Letters 83 (6) (2002) 307–314.
[3] M.A. Bender, M. Farach-Colton, The LCA problem revisited, in: LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, 2000, pp. 88–94.
[4] H. Bunke, J. Csirik, An improved algorithm for computing the edit distance of run-length coded strings, Information Processing Letters 54 (2) (1995) 93–96.
[5] 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.
[6] D. Harel, R.E. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM Journal on Computing 13 (2) (1984) 338–355.
[7] D.S. Hirschberg, Algorithms for the longest common subsequence problem, Journal of the ACM 24 (4) (1977) 664–675.
[8] J.W. Hunt, T.G. Szymanski, A fast algorithm for computing longest common subsequences, Communications of the ACM 20 (5) (1977) 350–353.
[9] V. Levenshtein, Binary codes capable of correcting spurious insertions and deletions of ones, Problems of Information Transmission 1 (1965) 8–17.
[10] J.J. Liu, G.S. Huang, Y.L. Wang, R.C.T. Lee, Edit distance for a run-length-encoded string and an uncompressed string, Information Processing Letters 105 (1) (2007) 12–16.
[11] J.J. Liu, Y.L. Wang, R.C.T. Lee, Finding a longest common subsequence between a run-length-encoded string and an uncompressed string, Journal of Complexity 24 (2) (2008) 173–184.
[12] J.S.B. Mitchell, A geometric shortest path problem, with application to computing a longest common subsequence in run-length encoded strings, Technical Report Department of Applied Mathematics, SUNY Stony Brook, 1997.

[13] C. Rick, Simple and fast linear space computation of longest common subsequences, Information Processing Letters 75 (6) (2000) 275–281.
[14] J. Vuillemin, A unifying look at data structures, Communications of the ACM 23 (1980) 229–239.
[15] R. Wagner, M. Fischer, The string-to-string correction problem, Journal of the ACM 21 (1) (1974) 168–173.
[16] 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.