1997

# Matching for Run-Length Encoded Strings

Alberto Apostolico

Gad M. Landau

Steven Skiena

## Report Number:

97-018

MATCHING FOR RUN-LENGTH
ENCODED STRINGS

Alberto Apostolico
Gad. M. Landau
Steven Skiena

CSD-TR 97-018
March 1997

# Matching for Run-Length Encoded Strings

Alberto Apostolico* Gad M. Landau† Steven Skiena ‡

Feb. 15, 1997

## 1 Motivation

Measuring the similarity between two strings, through such standard measures as Hamming distance, edit distance, and longest common subsequence, is one of the fundamental problems in pattern matching. In this paper, we consider the problem of finding the longest common subsequence of two strings. The standard dynamic programming algorithm computes the longest common subsequence of strings $X$ and $Y$ in $O(|X| \cdot |Y|)$ time. Here, we develop significantly faster algorithms for a special class of strings which emerge frequently in pattern matching problems.

A string is *run-length encoded* if it is described as an ordered sequence of pairs, each consisting of an alphabet symbol $\sigma$ and an integer counting the number of consecutive occurrences of $\sigma$. For example, the string $aaaabbbbcccabbbbcc$ can be encoded as $a^4b^4c^3a^1b^4c^2$. Such a run-length encoded string can be significantly shorter than the expanded string representation. Indeed, run-length coding serves as a popular image compression technique, since some classes of images, e.g., binary images in facsimile transmission, typically contain large patches of identically-valued pixels.

The need to approximately match run-length encoded strings emerged during development of an OCR system in concert with Data Capture Systems Inc. [8], which has been designed to achieve a low substitution error-rate via fixed-font character recognition. The

*Department of Computer Sciences, Purdue University, Computer Sciences Building, West Lafayette, IN 47907, USA and Dipartimento di Elettronica e Informatica, Università di Padova, Padova, Italy. axa@cs.purdue.edu. Work supported in part by NSF Grant CCR-9201078, by NATO Grant CRG 900293, by the National Research Council of Italy, and by the ESPRIT III Basic Research Programme of the EC under contract No. 9072 (Project GEPPCOM).

†Department of Computer Science, Polytechnic University, 6 MetroTech Center, Brooklyn, NY 11201, USA, and Department of Mathematics and Computer Science, Haifa University, Haifa 31905, Israel. landau@poly.edu. Work supported in part by NSF grant CCR-9305873.

‡Dept. of Computer Science, State University of New York, Stony Brook, NY, 11794-4400 skiena@cs.sunysb.edu. This work is partially supported by ONR award 400x116yip01 and NSF Grant CCR-9625669.

$i$th row or column of pixels in a given query character image will define a binary string containing a small number of white-black transitions. By comparing this run-length encoded string against the $i$th row or column of each of the character image-models, we can identify similar characters. Since a typical row/column contains approximately 50 pixels but only 3-4 white-black transitions, a time savings of roughly two orders of magnitude would follow by matching in time proportional to the product of the run lengths, instead of the full string lengths.

This problem of matching of run-length encoded strings is a natural generalization of the original string matching problem. Indeed, any matching algorithm which takes time proportional to the product of the run lengths on encoded strings would have the same worst-case complexity as standard matching algorithms while exploiting any runs which happen to exist.

Our problem is a simplified version of the previously studied Set LCS and the Set-Set LCS problems [6, 9]. In this paper, we present the first algorithm for finding the longest common subsequence of strings $X$ and $Y$ which runs in time polynomial in the size of the compressed strings. Our final algorithm runs in $O(kl \log(kl))$ time, where $k$ and $l$ are the compressed lengths of strings $X$ and $Y$, and is a substantial improvement on the previously best algorithm of Bunke and Csirik [3], which runs in $O(l|Y| + k|X|)$ time. Our algorithm is elegant but non-trivial, and suitable for implementation.

## 2 Previous Work

Throughout this paper, we use the following notation. Let $X_1 X_2 \ldots X_l$ denote the run length encoding of string $X$, where $X_i$ is a maximal run of identical characters and $|X_i|$ denotes the length of this run. The length of string $X$, denoted $|X|$, represents the total number of characters in $X$, so $|X| = \sum_{i=1}^{l} |X_i|$. Let $x_i$ denote the unique character comprising run $X_i$. Similarly $Y_1 Y_2 \ldots Y_k$ denotes the run length encoding of string $Y$.

A string $W$ is said to be a *subsequence* of $X$ if $W$ can be obtained from $X$ by deleting one or more symbols. The Longest Common Subsequence (LCS) problem for input strings $X$ and $Y$ consists of finding a longest string $W$ which is a subsequence of both $X$ and $Y$. String editing and LCS problems have been extensively studied, resulting in a copious literature for which we refer, e.g., to [2].

When the size of the alphabet $\Sigma$ is unbounded, an $\Omega(|X| \log |X|)$ lower bound for computing LCS applies, due to Hirschberg [4]. The best known lower bound for bounded $\Sigma$ is linear. Aho, Hirschberg and Ullman [1] showed that, for unbounded alphabets, any algorithm using only "equal-unequal" comparisons must take $\Omega(|X|^2)$ time in the worst case. The asymptotically fastest general solution rests on the corresponding solution by Masek and Paterson [7] to the string editing, and hence takes time $O(|X|^2 \log\log |X| / \log |X|)$.

In practice, the following $\Theta(|X| \times |Y|)$ dynamic programming algorithm from Hirschberg [5] is used. The algorithm starts with a matrix $L[0...|Y|, 0...|X|]$ filled with zeroes, and then transforms $L$ in such a way that $L[i, j]$ $(1 \leq i \leq |Y|, 1 \leq j \leq |X|)$ contains the length of an

LCS between $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$, as follows.

**for** $i = 1$ **to** $|Y|$ **do**
  **for** $j = 1$ **to** $|X|$ **do if** $x_i \neq y_j$ **then** $L[i,j] = \text{Max}\ \{L[i,j-1], L[i-1,j]\}$
    **else** $L[i,j] = L[i-1,j-1] + 1$

The correctness of this paradigm follows from the following relations:

$$\begin{array}{lcccl} L[i-1,j] & \leq & L[i,j] & \leq & L[i-1,j]+1; \\ L[i,j-1] & \leq & L[i,j] & \leq & L[i,j-1]+1; \\ L[i-1,j-1] & \leq & L[i,j] & \leq & L[i-1,j-1]+1. \end{array}$$

# 3 Longest Common Subsequence – initial algorithm

In this section, we present an algorithm for computing the longest common subsequence of run length encoded strings $X = X_1 X_2 \ldots X_l$ and $Y = Y_1 Y_2 \ldots Y_k$ in $O(kl(k+l))$ time. This algorithm maintains an $l \times k$ matrix $M$ of *blocks*, such that $M[i,j]$ contains the value of an optimal solution between prefixes $X^{(i)} = X_1 X_2 \ldots X_i$ and $Y^{(j)} = Y_1 Y_2 \ldots Y_j$. The correctness of our algorithm follows because $M$ contains all the essential information of the standard $|X| \times |Y|$ alignment matrix $L$ associated with the uncompressed strings.

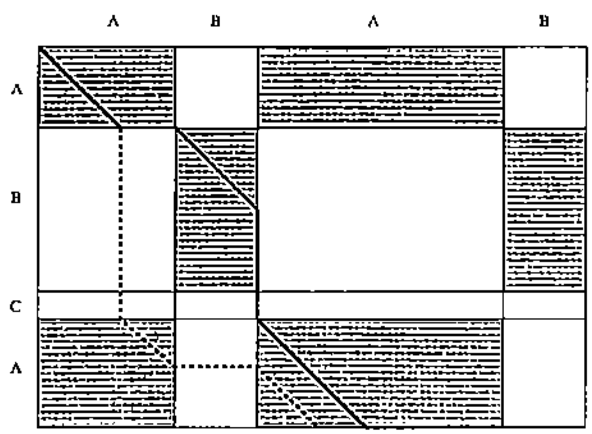


Figure 1: Light and dark blocks defined by strings $X$ and $Y$.

Figure 1 illustrates this matrix of blocks for input strings $X = a^6 b^3 a^8 b^3$ and $Y = a^3 b^6 c^1 a^4$. We say that block $(i,j)$ is *dark* if the corresponding characters match, i.e. $x_i = y_j$. Block $(i,j)$ is *light* if $x_i \neq y_j$. Any common subsequence defines a monotonically non-decreasing path from $(0,0)$ to $(|X|,|Y|)$. Each rightward step on this path denotes the deletion of a character from $X$, and each downward step a deletion from $Y$. The matched characters in the common subsequence correspond to diagonal down-right steps across $M$, hence the LCS maximizes the total number of such diagonal steps through the dark blocks of $M$.

Any such path can exit a dark block in one of three ways – at the lower right corner, along the bottom side, or along the right side. The longest common subsequence of Figure 1 (shown as the solid line), happens to enter and exit each dark block only through its corners. An optimal path with this additional constraint is computed easily in $O(kl)$ by dynamic programming. However, paths which exit dark blocks through sides are more complicated to account for, since the number of possible exit points on either side of a block can dominate the number of blocks on very long runs.

We now consider two special classes of paths across $M$. We define a *corner path* as one which enters dark blocks only at the upper-left corner and exits only through the lower-right corner. We say that a path beginning at the upper-left corner of dark block $(i, j)$ is *forced* if it exits through a side of $(i, j)$, and proceeds to the next dark block by a straight horizontal or vertical "leap", according to the case. As illustrated by the dotted line in Figure 1, there is precisely one forced path beginning from the upper lefthand corner of any dark block.

A subpath $p_i \ldots p_j$ of path $P$ is a contiguous chain of edges from $P$. Subpaths of forced and corner paths can be composed to define an interesting class of paths through $M$:

**Lemma 1** *There is always a longest common subsequence $W$ of $X$ and $Y$ such that $W$ is defined by a path composed of subpaths of forced and corner paths.*

**Proof:** Consider any path through $M$ which defines the longest common subsequence of $X$ and $Y$. We now describe a sequence of transformations which reduce it to a path of the prescribed shape.

First, consider any maximal subpath passing only through light blocks. Such a subpath consists only of rightward and downward moves, for it contributes no matched characters to the longest common subsequence. Therefore, without loss of generality, all of the rightward moves can be collected to appear before any of the downward moves.

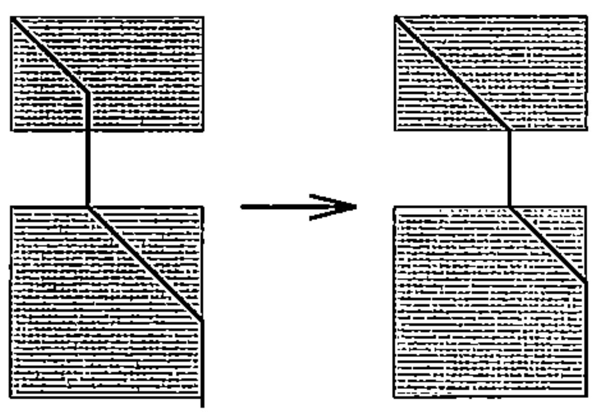

Figure 2: Converting an arbitrary subpath into a forced subpath.

Second, consider any maximal subpath through dark block $(i, j)$. This path cannot contain both a rightward and a downward move, since by replacing these with a diagonal move we increase the length of the putative longest common subsequence. Therefore, without loss of generality, all of the diagonal moves can be collected to appear before any of the vertical/horizontal moves.

Finally, we consider the dark blocks in the order they are encountered on the path from $(0,0)$ to $(|X|,|Y|)$. Consider the first dark block which is either (1) not entered through its upper-lefthand corner or (2) is not exited through its lower-righthand corner. Case (1) cannot occur in a longest common subsequence, since the subsequence will be lengthened by entering in the upper-lefthand corner. Case (2) describes the start of a forced subpath, unless dark blocks are not completely traversed. The reduction of Figure 2 converts this subpath into a forced subpath, thus giving the claimed result. ∎

**Theorem 2** *A longest common subsequence of run length encoded strings $X = X_1X_2\ldots X_l$ and $Y = Y_1Y_2\ldots Y_k$ can be computed in $O(kl(k+l))$ time.*

**Proof:** Lemma 1 guarantees that a longest common subsequence of $X$ and $Y$ can always be obtained by concatenation of subpaths of forced and corner paths. The following algorithm exhaustively constructs all such subpaths via dynamic programming:

$LCS1(X,Y)$
  $M[i,j] = 0, \;\; 1 \le i \le l, \;\; 1 \le j \le k$
  **for** $i = 1$ **to** $k$
    **for** $j = 1$ **to** $l$
      if (color(i,j) == "light") **then** $M[i,j] = \max(M[i-1,j], M[i,j-1])$
      **else begin** (* dark block *)
        $d = \min(|X_i|, |Y_j|)$
        $M[i,j] = \max(M[i-1,j-1]+d, M[i,j], M[i-1,j], M[i,j-1])$
        ForcedPathUpdate(i, j, M)
      **end**

The procedure ForcedPathUpdate explicitly traces out the forced path originating at block $(i,j)$, proceeding vertically if $|X_i| > |Y_j|$ and horizontally if $|X_i| < |Y_j|$, until the next dark block (say $(i',j)$) is encountered. On exiting each dark block $(i',j)$ along this forced path, the block value is updated where $M[i',j] = \max(M[i',j], M[i,j]+d')$, where $d'$ is the diagonal length of the forced path through $(i',j)$. This process continues until the forced path exits the corner of a block, or the end of one of the strings is encountered. This ForcedPathUpdate operation can be computed in $O(k+l)$ time for any block $(i,j)$.

Each light block requires constant time to update, while each dark block takes $O(k+l)$. The total time complexity follows since there are $O(kl)$ dark blocks. ∎

# 4 Longest Common Subsequence – a faster algorithm

In this section we present an algorithm that computes the LCS of the run length encoded strings in $O(kl\log(kl))$ time.

In the previous algorithm, each iteration $(i,j)$ was computed in $O(1)$ if $color(i,j)$ is "light". When $color(i,j)$ is dark the iteration computed $M[i,j]$ in $O(1)$ time before performing a ForcedPathUpdate operation in $O(k+l)$ time. In this section, we show how to replace this ForcedPathUpdate by a much more efficient operation.

The ForcedPathUpdate operation starts from $(i,j)$ and updates all $M[i',j']$s encountered on the way toward the lower right corner. Eventually, each dark $M[i',j']$ is updated by all forced paths that cross its block. In this improved algorithm, the ForcedPathUpdate is eliminated. While computing $M[i,j]$, only two forced paths from previous iterations will be considered, and their relevant values will be computed upon request.

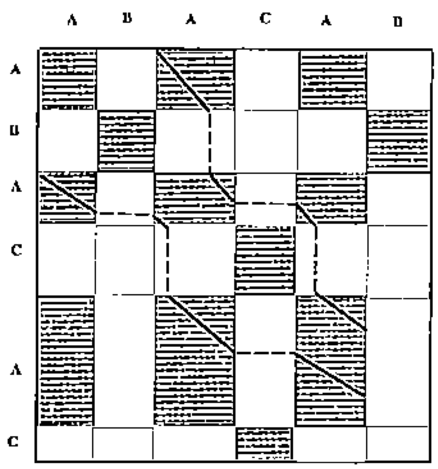


Figure 3: Two forced paths that match the character $A$.

**Lemma 3** *All characters which are matched on any given forced path will be identical. Also, two forced paths which proceed on matches of the same character will never cross each other (Fig 3).*

Consider a forced path that starts in an upper left corner of a dark block $(i,j)$ that matches the character $\alpha$. Its initial value $v$ is $M[i-1,j-1]$. This path moves down and to the right in light blocks and diagonally on dark blocks that match $\alpha$'s. By Lemma 3, this path will *not* cross blocks that match characters other then $\alpha$. A record is kept for each forced path, including the following information: (a) $(i,j)$ - starting location of the path; (b) the letter of the match; and (c) its initial value $v$.

Define $TOP^j(\alpha)$ to be the number of occurrences of the letter $\alpha$ in the uncompressed version of $X_1 \ldots X_j$, and $LEFT^i(\alpha)$ to be the number of occurrences of the letter $\alpha$ in $Y_1 \ldots Y_i$. For example, when string $Y = aaaabbbbcccabbbbcc$ is encoded as $a^4b^4c^3a^1b^4c^2$,

$LEFT^5(b)$ is 8. $LEFT^i(\alpha)$ will be defined only when $Y_i = \alpha$ or $Y_{i+1} = \alpha$, and $TOP^j(\alpha)$ defined only when $X_j = \alpha$ or $X_{j+1} = \alpha$.

Consider a forced path which starts at $(i, j)$ and matches $\alpha$ with an initial value $v$. When this path crosses column $j' > j$ its value will be $v' = v + TOP^{j'}(\alpha) - TOP^{j-1}(\alpha)$ (Fig 3). Moreover, it crosses column $j'$ at row $i^*$, where $i^*$ is the minimum row such that

$$LEFT^{i^*}(\alpha) = LEFT^{i-1}(\alpha) + TOP^{j'}(\alpha) - TOP^{j-1}(\alpha)$$

Similarly, when this path crosses row $i' > i$, its value will be $v' = v + LEFT^{i'}(\alpha) - LEFT^{i-1}(\alpha)$, and it crosses row $i'$ on column $j^*$ such that

$$TOP^{j^*}(\alpha) = TOP^{j-1}(\alpha) + LEFT^{i'}(\alpha) - LEFT^{i-1}(\alpha)$$

**Lemma 4** *Consider a forced path which starts at $(i, j)$ and matches $\alpha$ with an initial value $v$. Given a column $j'$ (row $i'$), the value of each forced path that crosses this column (row) can be computed in $O(1)$ time following $O(k + l)$ time preprocessing.*

**Proof:** By a performing a prefix-sum computation, the functions $LEFT$ and $TOP$ can be precomputed in $O(k + l)$ time, such that $TOP^j(\alpha)$ or $LEFT^i(\alpha)$ can retrieved in constant time. The appropriate values can be computed using the formulas above. ∎

As described in Section 3, $M[i, j]$ is the maximum of $M[i-1, j]$, $M[i, j-1]$ and the forced paths that cross its block, including the one that starts on its upper left corner. The set of forced paths can be divided into two groups. The first group contains all paths that cross column $j$ above row $i$, while the second group contains all paths that cross row $i$ left to column $j$. Our goal is to find the path with the highest score in each group, so that $M[i, j]$ can be computed in $O(1)$ time. Below, we discuss only how to find the highest in the first group, considering forced paths that match the character $\alpha$; the second group and other characters can be handled in the same way.

Since two forced paths that match the same character never intersect, the forced paths of character $\alpha$ define a top-down order. We define the order of a path starting from $M[i, j]$ as $ORDER(\alpha; i, j) = TOP^{j-1}(\alpha) - LEFT^{i-1}(\alpha)$. The paths intersect any column $j'$ according to the value of $ORDER$. However, the values of the forced paths at column $j'$ do not increase monotonically in their crossing order, because certain paths may begin with lower initial values, and they maintain the following property:

**Lemma 5** *Consider two forced paths with values $v_1'$ and $v_2'$ when they cross column $j'$, and $v_1''$ and $v_2''$ when they cross column $j''$. These paths maintain the equality: $v_1' - v_2' = v_1'' - v_2''$ (Fig 3).*

Therefore, if a forced path $p_1$ intersect column $j'$ lower than another forced path $p_2$ and its value on $j'$ is smaller than the value of $p_2$ on $j'$, then path $p_1$ can be deleted. Our goal is to maintain, in order, only the paths which have higher values than the paths above them. A balanced binary search tree can be built with the records of the forced paths matching $\alpha$, with the key of each path defined by its $ORDER$ function. This tree will be pruned so as to insure that for any given column $j'$, the values of the paths in the nodes increase during an in-order traversal.

We will maintain two balance binary search trees for each letter $\alpha$, one maintaining the order of paths crossing columns, the other maintaining the order of paths crossing rows. These same two trees will be used in dealing with all dark blocks that match $\alpha$. For each such block $M[i,j]$, we insert, separately, to both trees a new forced path that starts from the upper left corner of $M[i,j]$. Then we get the highest scores crossing the lower and right sides of $M[i,j]$, one from each tree. When computing a dark block $M[i,j]$ the following operations are performed:

- Step I. Insert a new forced path.
- Step II. Find the highest score ($C$) of the forced paths on column $j$, above row $i$.
- Step III. Find the highest score ($R$) of the forced paths on row $i$, left to column $j$.
- Step IV. $M[i,j] = \max(M[i-1,j], M[i,j-1], C, R)$.

Step I - Insertion of a new path.
(a) Compute $ORDER(\alpha; i,j) := TOP^{j-1}(\alpha) - LEFT^{i-1}(\alpha)$.
(b) Compute $v := M[i-1, j-1]$.
(c) Insert the new path into the tree.
(d) Compute the value of the path that is stored in the leaf on the left. If its value is greater than $v$ delete the new path.
(e) Compute the value of the path that is stored in the leaf on the right. If its value is smaller than $v$ delete the old path. Continue till you reach a path with a greater value.

Step II - Finding the highest score of the forced paths on column $j$, above row $i$.
(a) Compute $O := TOP^j(\alpha) - LEFT^i(\alpha)$.
(b) Find the location of $O$ in the tree.
(c) Compute the value $C$, of the path that is stored in the leaf on the left.

Step III is computed in an analogous way to Step II.

**Theorem 6** *A longest common subsequence of run length encoded strings $X = X_1X_2 \dots X_l$ and $Y = Y_1Y_2 \dots Y_k$ can be computed in $O(kl\log(k+l))$ time.*

**Proof:** The correctness of this procedure follows from the fact that all the relevant forced paths from the algorithm of Theorem 2 are evaluated in the dynamic programming phase of the current algorithm. The time complexity may be analyzed as follows. Precomputing the variables $LEFT$ and $TOP$ as in Lemma 4 takes $O(k+l)$ time. Each of the $2 \cdot \Sigma$ balanced binary search trees has at most $kl$ nodes, so any insertion, deletion or membership operation takes $O(\log(kl))$ time. We perform Steps I to IV for each of the $kl$ blocks. Step I takes $O(\log(kl) + (\textit{number of deletions}) \log(kl))$ time. Since each deleted block must previously have been inserted, the total number of deletions is $O(kl)$. Steps II and III are computed in $O(\log(kl))$ while Step IV requires $O(1)$ time. Therefore, $O((kl)\log(kl))$ time suffices to compute the longest common subsequence of $X$ and $Y$.

# 5 Open Problems

What can be said about more general versions of string matching, in particular edit distance computations?

# References


[1] A.V. Aho, D.S. Hirschberg and J.D. Ullman. Bounds on the complexity of the longest common subsequence problem. *J. Assoc. Comput. Mach.*, **23** pp. 1-12 (1976).

[2] A. Apostolico. String editing and longest common subsequences. *Handbook of Formal Languages (G. Rozenberg and A. Salomaa, Eds.)*, **Vol II**, pp. 361–398, Springer-Verlag (1996).

[3] H. Bunke, and J. Csirik. An improved algorithm for computing the edit distance of run-length coded strings. *IPL*, **54**, pp. 93-96 (1995).

[4] D.S. Hirschberg. An information theoretic lower bound for the longest common subsequence problem. *IPL* **7**, 1, pp. 40-41 (1978).

[5] D.S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the CACM* **18**, 6, pp. 341-343 (1975).

[6] D.S. Hirschberg, and L.L. Larmore. The set-set LCS problem. *Algorithmica*, **4**, pp. 503-510 (1989).

[7] W.J. Masek, and M. S. Paterson. A faster algorithm computing string edit distances. *J. Comput. System Sci.*, **20**, pp. 18–31 (1980).

[8] G. Sazaklis, E. Arkin, J. Mitchell, and S. Skiena. Geometric decision trees for optical character recognition. submitted for publication, 1996.

[9] B. Wang, G. Chen, and K. Park. On the set LCS and set-set LCS problems. *J. of Algorithms*, **14**, pp. 466-477 (1993).