Matching for Run-Length Encoded Strings

Alberto Apostolico* Gad M. Landau† Steven Skiena ‡

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 (see, e.g., [2] and references therein). We consider the problem of finding the longest common subsequence of two strings. A well-known dynamic programming algorithm computes the longest common subsequence of strings X and Y in $O(|X| \cdot |Y|)$ time. In this paper, we develop significantly faster algorithms for a special class of strings which emerge frequently in pattern matching problems.

A string S is *run-length encoded* if it is described as an ordered sequence of pairs (σ, i), each consisting of an alphabet symbol σ and an integer i. Each pair corresponds to a run in S consisting of i consecutive occurrences of σ. 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 many classes of images, such as 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 optical character recognition (OCR) system. This system, built in association with Data Capture Systems Inc. [9], has been designed to achieve a low substitution error-rate via fixed-font character recognition. The ith row or column of pixels in a given query character image will define a binary string containing a small

*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 Grants CCR-9201078 and CCR-9700276, by NATO Grant CRG 900293, by British Engineering and Physical Sciences Research Council Grant GR/L19362, 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 and Information Science, Polytechnic University, 6 MetroTech Center, Brooklyn, NY 11201, USA, and Department of Computer Science, Haifa University, Haifa 31905, Israel. landau@poly.edu. Work supported in part by NSF Grants CCR-9305873 and CCR-9610238.

‡Department 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 431-0857A and NSF Grant CCR-9625669.

number of white-black transitions. By comparing this run-length encoded string against the ith row or column of each of the character image-models, we can identify similar characters. Since a typical row/column of the image contains approximately 50 pixels but only 3 to 4 white-black transitions, a time savings of roughly two orders of magnitude follows from 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, 10]. In this paper, we present the first algorithm which finds the longest common subsequence of strings X and Y 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 Preliminaries

Throughout this paper, we use the following notation. Let $X_1X_2\ldots X_l$ 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 $Y_1Y_2\ldots Y_k$ 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 to [2].

When the size of the alphabet Σ is unbounded, an $\Omega(|X|\log|X|)$ lower bound for computing LCS applies, due to Hirschberg [4]. The best known lower bound for bounded Σ 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 algorithm of Masek and Paterson [7] for string editing, and hence takes time $O(|X|^2\log\log|X|/\log|X|)$.

In practice, one could use the following $\Theta(|X| \times |Y|)$ dynamic programming algorithm from Hirschberg [5]. The algorithm starts with a matrix $L[0...|X|, 0...|Y|]$ filled with zeroes, and then transforms L so that $L[i, j]$ $(1 \leq i \leq |X|, 1 \leq j \leq |Y|)$ contains the length of an LCS between $x_1x_2...x_i$ and $y_1y_2...y_j$:

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

3 Longest Common Subsequence – initial algorithm

In this section, we present an algorithm for computing the longest common subsequence of run length encoded strings $X = X_1X_2\dots X_l$ and $Y = Y_1Y_2\dots Y_k$ 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 $X^{(i)} = X_1X_2\dots X_i$ and $Y^{(j)} = Y_1Y_2\dots Y_j$. 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.

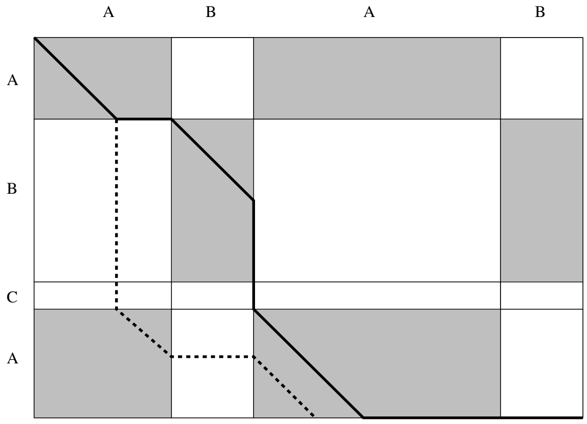

Figure 1: Light and dark blocks defined by strings X and Y.

Figure 1 illustrates this matrix of blocks for input strings $X = a^3b^6c^1a^4$ and $Y = a^6b^3a^8b^3$ 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 Y, and each downward step a deletion from X. 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 can be computed easily in $O(kl)$ time 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 a dark block is *forced* if it traverses dark blocks by strictly diagonal moves and, whenever the right (respectively, lower) side of an intermediate dark block is reached, proceeds to the next dark block by a straight horizontal (respectively, vertical) "leap" through the light blocks in between. 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 $p_i \dots p_j$ of path P is a contiguous chain of edges from P. Subpaths of forced and corner paths can be composed to define the longest common subsequence. Intuitively, whenever we enter a dark block in a upper-left corner, it will be the start of a forced path. The forced path stops when we hit a side, where we decide to follow the side to the lower-right corner. At this point, the LCS moves along sides till we start a new forced path by entering a new upper-left corner:

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. Since our maximal subpath is part of an optimal solution, there can be no matched character (whence, no dark block) between its beginning and end. In other words, the light blocks traversed by the subpath are lined up either horizontally or vertically. But then, without loss of generality, all of the rightward moves can be collected to appear before any of the downward moves in the subpath.

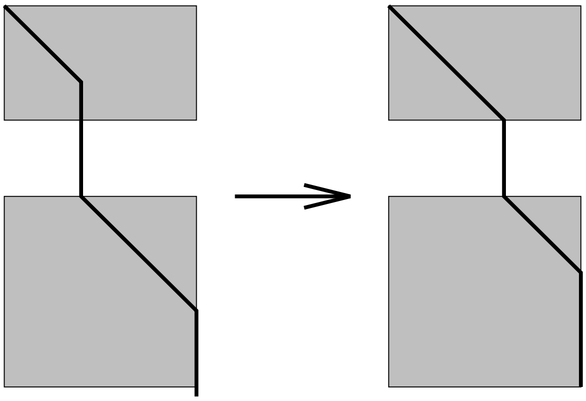

Figure 2: Converting an arbitrary subpath into a forced subpath.

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 before Case (2) 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. ▮

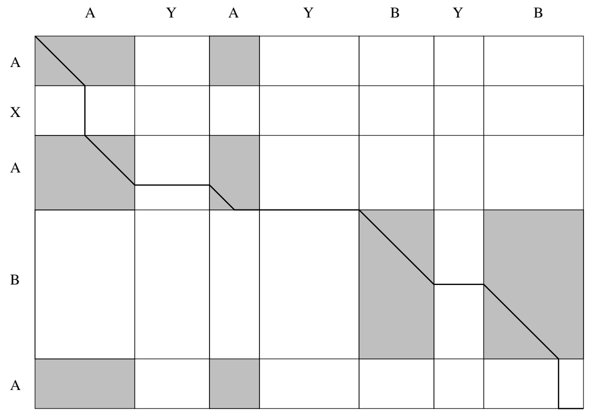

Figure 3: A longest common subsequence composed of two forced subpaths.

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 the concatenation of subpaths of forced and corner paths. The following algorithm exhaustively constructs all such subpaths via dynamic programming:

```
$LCS1(X, Y)$
          $M[i, j] = 0, \quad 1 \leq i \leq l, \quad 1 \leq j \leq k$
          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 contribution to 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 quickly computed upon request.

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

Proof: See Figure 4. Consider a forced path that starts in an upper left corner of a dark block (i, j) of character α. 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 $\alpha's$. This path cannot cross blocks that match characters other than α, because it never leaves a row or column of character α. Take now any other forced path that shares, say, some initial column j' with the path under consideration. As long as these paths co-exist, however, we have that each diagonal move of the second path must be accompanied, on the same column, by a diagonal move of the first one. Therefore, the two paths cannot meet. ▮

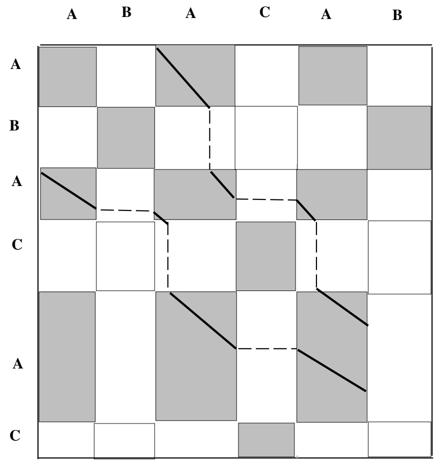

Figure 4: Two forced paths that match the character A.

In our algorithm, the following information is kept for each forced path: (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 α in the uncompressed version of $Y_1 \ldots Y_j$, and $LEFT^i(\alpha)$ to be the number of occurrences of the letter α in $X_1 \ldots X_i$. For example, when string $X = aaaabbbbcccabbbbcc$ is encoded as $a^4b^4c^3a^1b^4c^2$, $LEFT^5(b)$ is 8. $LEFT^i(\alpha)$ will be defined only when $X_i = \alpha$ or $X_{i+1} = \alpha$, and $TOP^j(\alpha)$ defined only when $Y_j = \alpha$ or $Y_{j+1} = \alpha$. Tables $LEFT$ and TOP are computed straightforwardly in $O((k+l)$ time from the encoded strings.

Consider a forced path which starts at (i,j) and matches α with an initial value v. When this path crosses column $j' > j$, its value will be $v' = v + TOP^{j'}(\alpha) - TOP^{j-1}(\alpha)$. See Figure 4 for an example. In addition, it crosses column j' at row i^*, where i^* is the minimum row such that

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

Moreover, in the uncompressed version it crosses at the i^{uc}th row of the i^*th block and

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

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

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

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

Proof: Immediate from the above discussion. ∎

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 on the left of 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 α. The second group and other characters can be handled similarly.

Since two forced paths that match the same character never intersect, the forced paths of character α obey a top-down order. We define the rank relative to this order of a path starting from $M[i,j]$ as $RANK(\alpha;i,j) = TOP^{j-1}(\alpha) - LEFT^{i-1}(\alpha)$. The paths intersect any column j' according to the value of $RANK$. In principle, the values of the candidate partial solutions associated with all forced paths at column j' do not necessarily increase monotonically according to their crossing order, because some of the forced paths may begin with lower initial values. However, consider two arbitrary forced paths of a same character α, both crossing some column j'. In order for these paths to reach some column j'', they must both match precisely all instances of α that fall between j' and j''. In other words, forced paths 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''. Then those values obey the equality: $v_1' - v_2' = v_1'' - v_2''$*

Therefore, whenever a forced path p_1 intersects column j' lower than another forced path p_2, but the value of p_1 at j' is smaller than the value of p_2 at j', then path p_1 can be deleted from further consideration. Our goal is to maintain, in order, only the paths which have higher values than the paths above them. Namely, to keep the forced paths sorted by ranks and by values.

In order to be able to maintain the above properties we need a data structure that keeps the forced paths ordered, and allows adding and deleting of forced paths. A record with its $RANK$, its initial value v, and its starting location (i,j) is kept for each forced path. The key of the record is the $RANK$. We are using balanced binary search trees, where the records are stored in the leaves, as our data structure. Note that in the balanced binary search tree the leaves are sorted according to their keys, and a record can be found, added, or deleted in logarithmic time. The tress will keep the paths sorted according to their ranks and the algorithm bellow will keep them sorted according to their values.

Since forced paths that matches different letters are independent they are maintained separately. In addition, we maintain separate trees for the forced paths crossing rows and columns.

Hence, we will maintain two balanced binary search trees for each letter α, one maintaining the ordered list of paths matching the letter α and crossing columns, the other maintaining the ordered list of paths matching the letter α and crossing rows. These two trees will be used in dealing with all dark blocks that match α. For each such block $M[i,j]$, we insert, separately, into both trees a new forced path that starts from the upper left corner of $M[i,j]$.

As was described in the previous section when $color(i,j)$ is dark, $M[i,j]$ is the maximum of $M[i-1,j]$, $M[i,j-1]$ and the values of all forced paths that cross its block. Here, since the paths are sorted according to their ranks and values, it is sufficient to consider only two forced paths. These paths are the closest paths to the lower right corner of $M[i,j]$, one that crosses the right side of $M[i,j]$ and one that crosses the lower side of $M[i,j]$, and we get them one from each tree.

When computing a dark block $M[i,j]$, we perform the following operations:

> Step I. Insert a new forced path according to its rank, and keep the paths sorted according to their value.
>
> 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 - Inserting a new path.
(a) Compute $RANK(\alpha; i,j) := TOP^{j-1}(\alpha) - LEFT^{i-1}(\alpha)$.
(b) Compute $v := M[i-1,j-1]$.
(c) Insert the new path into the trees.
(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 until 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* $X = X_1X_2\ldots X_l$ *and* $Y = Y_1Y_2\ldots Y_k$ *can be computed in* $O(kl\log(k+l))$ *time.*

Proof: The correctness of this procedure follows because 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 Conclusion

It is well known that the LCS problem may be regarded as a particular case of the more general string editing problem. In this latter problem, we are asked to transform one of two given strings in the other by an optimal sequence of elementary *edit operations* consisting each of the deletion of a symbol, or the insertion of a symbol, or the substitution of a symbol with another one. Note that a substitution may be always implemented by one deletion followed by one insertion. The special case of string editing represented by the LCS problem is achieved by assigning, e.g., unit weights to insertion and deletion, and a weight of at least 2 to substitution. Under these conditions, an optimal solution may always achieve the effect of a substitution by a combined deletion-insertion. In the notation of the present paper, this means that in pursuing an optimal path it is safe to jump across a light box by an either horizontal or vertical transition. The algorithm presented in this paper makes crucial use of this fact, which no longer holds when considering more general cases.

Mitchell [8] has recently obtained $O((d+k+l)\log(d+k+l))$ algorithm for more general string matching in run-length encoded strings, where d is the number of dark blocks. His algorithm is based on computing geometric shortest paths using special convex distance functions.

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] J. Mitchell. A Geometric Shortest Path Problem, with Application to Computing a Longest Common Subsequence in Run-Length Encoded Strings. Technical Report, Dept. of Applied Mathematics, SUNY Stony Brook, 1997.

[9] G. Sazaklis, E. Arkin, J. Mitchell, and S. Skiena. Geometric decision trees for optical character recognition. Proc. 13th ACM Symposium on Computational Geometry, 1997, short communications.

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