Approximate Matching of Run-Length Compressed Strings*

Veli Mäkinen † Gonzalo Navarro ‡ Esko Ukkonen *

Abstract

We focus on the problem of approximate matching of strings that have been compressed using run-length encoding. Previous studies have concentrated on the problem of computing the longest common subsequence (LCS) between two strings of length m and n, compressed to m' and n' runs. We extend an existing algorithm for the LCS to the Levenshtein distance achieving $O(m'n + n'm)$ complexity. Furthermore, we extend this algorithm to a weighted edit distance model, where the weights of the three basic edit operations can be chosen arbitrarily. This approach also gives an algorithm for approximate searching of a pattern of m letters (m' runs) in a text of n letters (n' runs) in $O(mm'n')$ time. Then we propose improvements for a greedy algorithm for the LCS, and conjecture that the improved algorithm has $O(m'n')$ expected case complexity. Experimental results are provided to support the conjecture.

Key Words. Compressed pattern matching, run-length encoding, Levenshtein distance, longest common subsequence, weighted edit distance.

---

*Appears in *Algorithmica* 35:4(347-369), 2003.

†Department of Computer Science, P.O. Box 26 (Teollisuuskatu 23) FIN-00014 University of Helsinki, Finland. {vmakinen,ukkonen}@cs.helsinki.fi. Supported by the Academy of Finland under grant 22584.

‡Center for Web Research, Department of Computer Science, University of Chile, Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl. Supported by Millenium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile.

1 Introduction

The problem of *compressed pattern matching* is, given a compressed text T and a (possibly compressed) pattern P, to find all occurrences of P in T without decompressing T (and P). The goal is to search faster than by using the basic scheme: decompression followed by a search.

In the basic approach, we are interested in reporting only the *exact* occurrences, i.e. the locations of the substrings of T that match pattern P exactly. We can loosen the requirement of exact occurrences to *approximate occurrences* by introducing a distance function to measure the similarity between P and its occurrence in T. Now, we want to find all the approximate occurrences of P in T, where the distance between P and a substring of T is at most a given error threshold k. Often a suitable distance measure between two strings is the *edit distance*, defined as the minimum amount of character insertions, deletions and substitutions that are needed to make the two strings equal. For this distance we are interested in $k < |P|$ errors.

Many studies have been made around the subject of compressed pattern matching over different compression formats, starting with the work of Amir and Benson [1], e.g. [2, 10, 17, 16]. The only works addressing the approximate variant of the problem have been [14, 19, 22], on Ziv-Lempel [27].

Our focus is approximate matching over *run-length encoded* strings. In run-length encoding, a string that consists of repetitions of letters is compressed by encoding each repetition as a pair ("letter","length of the repetition"). For example, string *aaabbbbccaab* is encoded as a sequence $(a,3)(b,4)(c,2)(a,2)(b,1)$. This technique is widely used, especially in image compression, where repetitions of pixel values are common. This is particularly interesting for fax transmissions and bilevel images. Approximate matching on images can be a useful tool to handle distortions. Even a one-dimensional compressed approximate matching algorithm would be useful to speed up existing two-dimensional approximate matching algorithms, e.g. [15, 6].

Exact pattern matching over run-length encoded text can be done optimally in $O(m'+n')$ time, where m' and n' are the compressed sizes of the pattern and the text [1].

Approximate pattern matching over run-length encoded text has not been considered before this study, but there has been work on the distance calculation, namely, given two strings of length m and n that are run-length compressed to lengths m' and n', calculate their distance using the compressed representations of the strings. This problem was first posed by Bunke and Csirik [7]. They considered the version of edit distance without the replacement operation, which is related to the problem of calculating the longest common subsequence (LCS) of two strings. They gave an $O(m'n')$ time algorithm for a special case of the problem, where all run-lengths are of equal size. Later, they gave an $O(m'n + n'm)$ time algorithm for the general case [8]. A major improvement over the previous results was due to Apostolico, Landau, and Skiena [4]. They first gave a basic $O(m'n'(m' + n'))$ algorithm, and further improved it to $O(m'n' \log(m'n'))$. Mitchell [20] gave an algorithm with the same time complexity in the worst case, but faster with some inputs. Its time complexity is $O((p + m' + n') \log(p + m' + n'))$, where p is the amount of pairs of compressed characters that match (p equals the amount of equal letter boxes, see the definition in Section 2.2). All these algorithms were limited to the LCS distance, although Mitchell's method [20] could be applied when different costs are assigned to the insertion and deletion operations. It still remained an open question (as posed by Bunke and Csirik) whether similar improvements could be found for a more general set of edit operations and their costs.

We give an algorithm for computing the *Levenshtein* distance [18] between two strings. In the Levenshtein distance a unit cost is assigned to each of the three edit operations. The algorithm is an extension of the $O(m'n + n'm)$ algorithm of Bunke and Csirik [8]. We keep the same cost but generalize the algorithm to handle a more complex distance model. Independently from our work, Arbell, Landau, and Mitchell have found a similar algorithm [5].

We manage to extend the $O(m'n + n'm)$ algorithm also to a weighted edit distance model, where the costs for the three operations can be chosen arbitrarily.

We modify our algorithm to work in a context of approximate pattern matching, and achieve $O(mm'n')$ time for searching a pattern of length m that is run-length compressed to length m', in a run-length compressed text of length n'.

We also study the LCS calculation. First, we give a greedy algorithm for the LCS that works in $O(m'n'(m'+n'))$ time. Adapting the well known diagonal method [24], we are able to improve the greedy method to work in $O(d^2 min(n', m'))$ time, where d is the edit distance between the two strings (under insertions and deletions with the unit cost model).

Then we present improvements for the greedy method for the LCS, which do not however affect the worst case, but do have an effect on the average case. We end up conjecturing that our improved algorithm is $O(m'n')$ time on average. As we are unable to prove it, we provide instead experimental evidence to support the conjecture.

This paper is an extended version of a conference paper [21]. The weighted edit distance computation was developed after the conference version. Motivated by our open question in that paper, Crochemore, Landau, and Ziv-Ukelson [9] noticed that their sub-quadratic sequence alignment algorithm for unrestricted cost matrices could be generalized to this problem; they obtained the same $O(m'n + n'm)$ bound using completely different techniques from ours.

2 Edit Distance on Run-Length Compressed Strings

2.1 Edit Distance

Let Σ be a finite set of symbols, called an *alphabet*. A *string* A of length $|A| = m$ is a sequence of symbols in Σ, denoted by $A = A_{1\ldots m} = a_1 a_2 \ldots a_m$, where $a_i \in \Sigma$ for every i. If $|A| = 0$, then $A = \lambda$ is an empty string. A *subsequence* of A is any sequence $a_{i_1} a_{i_2} \ldots a_{i_k}$, where $1 \le i_1 < i_2 \cdots < i_k \le m$.

The *edit distance* $D(A, B)$ can be used to measure the similarity between two strings $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$ by calculating the minimum cost of edit operations that are needed to convert A into B [18, 26, 23]. The usual edit operations are *substitution* (convert a_i into b_j, denoted by $a_i \to b_j$), *insertion* ($\lambda \to b_j$), and *deletion* ($a_i \to \lambda$). Different costs for edit operations can be given depending on the letters involved. We define a nonnegative function δ that assigns a cost to each of the above operations. The cost to convert A into B must be a distance, which holds whenever δ is strictly positive ($\delta(x \to y) = 0 \Leftrightarrow x = y$) symmetric ($\delta(x \to y) = \delta(y \to x)$) and satisfies the triangle inequality ($\delta(x \to y) + \delta(y \to z) \geq \delta(x \to z)$) for every $x, y, z \in \Sigma \cup \{\lambda\}$.

For the *Levenshtein distance* (denoted by $D_L(A, B)$) [18], we assign costs $\delta(a \to a) = 0$, $\delta(a \to b) = 1$, $\delta(a \to \lambda) = 1$, and $\delta(\lambda \to a) = 1$, for all $a, b \in \Sigma$, $a \neq b$. If substitutions are forbidden, i.e. $\delta(a \to b) = \infty$, we get the distance $D_{ID}(A, B)$.

In general, the edit distance $D(A, B)$ with arbitrary δ costs can be calculated by using dynamic programming [23]; evaluating an $(m+1) \times (n+1)$ matrix (d_{ij}), $0 \leq i \leq m$, $0 \leq j \leq n$, using the initial value $d_{0,0} = 0$ and the recurrence

$$d_{i,j} = min(d_{i-1,j} + \delta(a_i \to \lambda), d_{i,j-1} + \delta(\lambda \to b_j), d_{i-1,j-1} + \delta(a_i, b_j)) \qquad (1)$$

where d is assumed to take the value ∞ when accessed outside its bounds. The matrix (d_{ij}) can be evaluated row-by-row or column-by-column in $O(mn)$ time, and the value d_{mn} equals $D(A, B)$.

The distance $D_L(A, B)$ is obtained as a particular case using recurrence:

$$d_{i,j} = min(d_{i-1,j} + 1, d_{i,j-1} + 1, d_{i-1,j-1} + \textbf{if } a_i = b_j \textbf{ then } 0 \textbf{ else } 1). \qquad (2)$$

The recurrence for $D_{ID}(A, B)$ is

$$d_{i,j} = min(d_{i-1,j} + 1, d_{i,j-1} + 1, d_{i-1,j-1} + \textbf{if } a_i = b_j \textbf{ then } 0 \textbf{ else } \infty). \qquad (3)$$

The problem of calculating the *longest common subsequence* of strings A and B (denoted by $LCS(A, B)$), is related to the distance $D_{ID}(A, B)$. It is easy to see that $2 * |LCS(A, B)| = m + n - D_{ID}(A, B)$. Also, the sequence $LCS(A, B)$ can be extracted using recurrence (3) if the *optimal path* is stored in the matrix: In each cell d_{ij} a link is stored to the cell that gives the minimum value in the recurrence (3). Now, $LCS(A, B)$ is the concatenation of symbols a_i (or alternatively b_j) that correspond to cells d_{ij} in the optimal path from d_{00} to d_{mn} that have diagonal-wise links. (In fact there may be more than one optimal path yielding different LCSs of the same length.)

2.2 Dividing the Edit Distance Matrix into Boxes

A *run-length* encoding of the string $A = a_1 a_2 \ldots a_m$ is $A' = (a_1, p_1)(a_{p_1+1}, p_2)(a_{p_1+p_2+1}, p_3) \ldots (a_{m-p_{m'}+1}, p_{m'}) = (a_{i_1}, p_1)(a_{i_2}, p_2) \ldots (a_{i_{m'}}, p_{m'})$, where (a_{i_k}, p_k) denotes a sequence $\alpha_k = a_{i_k} a_{i_k} \ldots a_{i_k} = a_{i_k}^{p_k}$ of length $|\alpha_k| = p_k$. We also call (a_{i_k}, p_k) a *run* of a_{i_k}. String A is *optimally run-length encoded* if $a_{i_k} \neq a_{i_{k+1}}$ for all $1 \leq k < m'$.

In the next sections, we will show how to speed up the evaluation of values d_{mn} for both distances $D_L(A, B)$ and $D_{ID}(A, B)$ when both strings A and B are run-length encoded. We will generalize to $D(A, B)$ as well. In all the methods, we use the following notation to divide the matrix (d_{ij}) into sub-matrices (see Figure 1).

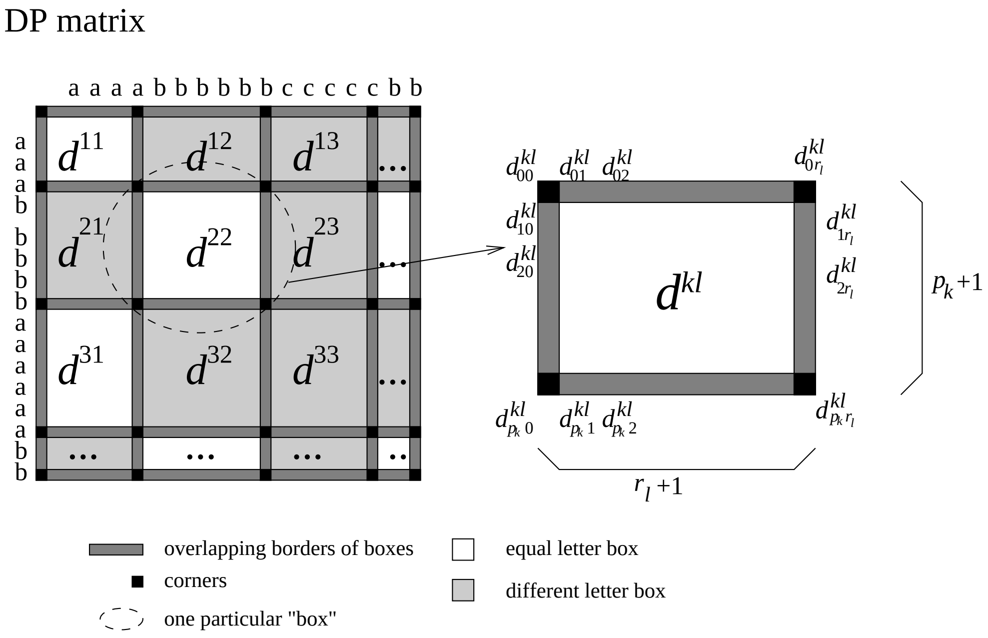

Figure 1: A dynamic programming matrix split into run-length blocks.

Let $A' = (a_{i_1}, p_1)(a_{i_2}, p_2) \ldots (a_{i_{m'}}, p_{m'})$ and $B' = (b_{j_1}, r_1), (b_{j_2}, r_2) \ldots (b_{j_{n'}}, r_{n'})$ be the run-length encoded representations of strings A and B. The rows and columns that correspond to the ends of runs in A and B divide the edit distance matrix (d_{ij}) into sub-matrices. To ease the notation later on, we define the sub-matrices so that they overlap on the borders. Formally, each pair of runs $(a_{i_k}, p_k), (b_{j_\ell}, r_\ell)$ defines a $(p_k + 1) \times (r_\ell + 1)$ sub-matrix $(d_{s,t}^{k,\ell})$ such that

$$d_{s,t}^{k,\ell} = d_{i_k+s-1, j_\ell+t-1}, \qquad 0 \leq s \leq p_k, 0 \leq t \leq r_\ell. \tag{4}$$

We will call sub-matrices $(d_{s,t}^{k,\ell})$ *boxes*. If a pair of runs corresponding to a box contains equal letters (i.e. $a_{i_k} = b_{j_\ell}$), then $(d_{s,t}^{k,\ell})$ is called an *equal letter box*. Otherwise we call $(d^{k,\ell}_{s,t})$ a *different letter box.* Adjacent boxes can form *runs of different letter boxes* along rows and columns. We assume that both strings are optimally run-length encoded, and hence runs of equal letter boxes can not occur. (If the strings are not optimally encoded, they can easily be converted into optimally encoded in $O(m'+n')$ time by joining adjacent runs of equal letters. This cost is negligible compared to those of our algorithms.)

3 An $O(mn'+m'n)$ Algorithm for the Levenshtein Distance

Bunke and Csirik [8] gave an $O(mn'+m'n)$ time algorithm for computing the LCS between two strings of lengths n and m run-length compressed to n' and m'. They pose it as an open problem extending their algorithm to the Levenshtein distance. This is what we do in this section, without increasing the complexity to compute the new distance D_L. Arbell, Landau, and Mitchell [5] have independently found a similar algorithm. Their solution is also based on the same idea of extending the $O(mn'+m'n)$ LCS algorithm to the Levenshtein distance.

Compared to the LCS-related distance D_{ID}, the Levenshtein distance D_L permits an additional character substitution operation, at cost 1. We compute $D_L(A,B)$ by filling all the borders of all the boxes $(d^{k,\ell}_{s,t})$ (see Figure 1). We manage to fill each cell in constant time, which adds up the promised $O(mn'+m'n)$ complexity. The space complexity can be made $O(n+m)$ by processing the matrix row-wise or column-wise.

3.1 Basic Algorithm

We start with two lemmas that characterize the relationships between the border values in the boxes $(d^{k,\ell}_{s,t})$. First, we consider the equal letter boxes:

Lemma 1 (Bunke and Csirik [8]) *The recurrences (2) and (3) can be replaced by*

$$d^{k,\ell}_{s,t} = \textbf{if}\ \ s \le t\ \ \textbf{then}\ \ d^{k,\ell}_{0,t-s}\ \ \textbf{else}\ \ d^{k,\ell}_{s-t,0}, \tag{5}$$

where $1 \le s \le p_k$ and $1 \le t \le r_\ell$, for values $d^{k,\ell}_{s,t}$ in an equal letter box.

Note that Lemma 1 holds for both Levenshtein and LCS distance models, because formulas (2) and (3) are equal when $a_i = b_j$. Since we are computing all the cells in the borders of the boxes, Lemma 1 permits computing new box borders in constant time using those of previous boxes.

The difficult part lies in the different letter boxes.

Lemma 2 *The recurrence (2) can be replaced by*

$$d^{k,\ell}_{s,t} = 1 + min\,(\quad t-1 + min_{max(0,s-t) \le q \le s}\ d^{k,\ell}_{q,0}\ , \quad s-1 + min_{max(0,t-s) \le q \le t}\ d^{k,\ell}_{0,q}\ ), \tag{6}$$

where $1 \le s \le p_k$ and $1 \le t \le r_\ell$, for values $d^{k,\ell}_{s,t}$ in a different letter box.

Proof. We use induction on $s+t$. If $s+t=2$ then formula (6) becomes $d^{k,\ell}_{1,1} = 1 + min(d^{k,\ell}_{0,0}, d^{k,\ell}_{1,0}, d^{k,\ell}_{0,1})$, which matches recurrence (2). In the inductive case we have

$$d^{k,\ell}_{s,t} = 1 + min(d^{k,\ell}_{s-1,t-1}, d^{k,\ell}_{s-1,t}, d^{k,\ell}_{s,t-1})$$

by recurrence (2), and using the induction hypothesis we get

$$\begin{aligned} d^{k,\ell}_{s,t} = 2 + min\,(\quad & min\,(\quad t-2 + min_{max(0,s-t) \le q \le s-1}\ d^{k,\ell}_{q,0}\ , \\ & \qquad\quad s-2 + min_{max(0,t-s) \le q \le t-1}\ d^{k,\ell}_{0,q}\ )\ , \\ & min\,(\quad t-1 + min_{max(0,s-1-t) \le q \le s-1}\ d^{k,\ell}_{q,0}\ , \\ & \qquad\quad s-2 + min_{max(0,t-s+1) \le q \le t}\ d^{k,\ell}_{0,q}\ )\ , \\ & min\,(\quad t-2 + min_{max(0,s-t+1) \le q \le s}\ d^{k,\ell}_{q,0}\ , \\ & \qquad\quad s-1 + min_{max(0,t-1-s) \le q \le t-1}\ d^{k,\ell}_{0,q}\ )\ ) \end{aligned}$$

$$d^{k,\ell}_{s,t} = 1 + min\,(\ t-1 + min_{max(0,s-t) \le q \le s}\ d^{k,\ell}_{q,0}\ ,\ s-1 + min_{max(0,t-s) \le q \le t}\ d^{k,\ell}_{0,q}\ ),$$

where we have used the property that consecutive cells in the (d_{ij}) matrix differ at most by 1 [25]. Note that we have assumed $s > 1$ and $t > 1$. The particular cases $s = 1$

or $t = 1$ are easily derived as well, for example for $s = 1$ and $t > 1$ we have

$$
\begin{aligned}
d_{1,t}^{k,\ell} &= 1 + min(d_{0,t-1}^{k,\ell}, d_{0,t}^{k,\ell}, d_{1,t-1}^{k,\ell}) \\
&= 1 + min\Big(d_{0,t-1}^{k,\ell}, d_{0,t}^{k,\ell}, \\
&\qquad 1 + min\left(t-2 + min_{max(0,2-t) \le q \le 1}\ d_{q,0}^{k,\ell}, min_{max(0,t-2) \le q \le t-1}\ d_{0,q}^{k,\ell}\right) \Big) \\
&= 1 + min\left(d_{0,t-1}^{k,\ell}, d_{0,t}^{k,\ell}, t-1 + min(d_{0,0}^{k,\ell}, d_{1,0}^{k,\ell}), 1 + min(d_{0,t-2}^{k,\ell}, d_{0,t-1}^{k,\ell})\right) \\
&= 1 + min\left(t-1 + min(d_{0,0}^{k,\ell}, d_{1,0}^{k,\ell}), min(d_{0,t-1}^{k,\ell}, d_{0,t}^{k,\ell})\right),
\end{aligned}
$$

which is the particularization of formula (6) for $s = 1$. □

Formula (6) relates the values at the right and bottom borders of a box to its left and top borders. Yet it is not enough to compute the cells in constant time. Although we cannot compute one cell in $O(1)$ time, we can compute all the p_k (or r_ℓ) cells in overall $O(p_k)$ (or $O(r_\ell)$) time.

Figure 2 shows the algorithm. We use a data structure (which in the pseudo-code is represented just as a set M_*) able to handle a multiset of elements starting with a single element, adding and deleting elements, and delivering its minimum value at any time. It will be used to maintain and update the minima $min_{max(0,s-t) \le q \le s}\ d_{q,0}^{k,\ell}$ and $min_{max(0,t-s) \le q \le t}\ d_{0,q}^{k,\ell}$, used in formula (6). We see later that in our particular case all those operations can be performed in constant time.

In the code we use $dr_s^{k,\ell} = d_{s,r_\ell}^{k,\ell}$ for the rightmost column and $db_t^{k,\ell} = d_{p_k,t}^{k,\ell}$ for the bottom row. Their update formulas are derived from formula (6):

$$
\begin{aligned}
dr_s^{k,\ell} = \ & 1 + min(\ r_\ell - 1 + min_{max(0,s-r_\ell) \le q \le s}\ dr_q^{k,\ell-1}\ , \\
& \qquad\qquad s - 1 + min_{max(0,r_\ell-s) \le q \le r_\ell}\ db_q^{k-1,\ell}\ ), \\
db_t^{k,\ell} = \ & 1 + min(\ t - 1 + min_{max(0,p_k-t) \le q \le p_k}\ dr_q^{k,\ell-1}\ , \\
& \qquad\qquad p_k - 1 + min_{max(0,t-p_k) \le q \le t}\ db_q^{k-1,\ell}\ ).
\end{aligned}
$$

The whole algorithm can be made $O(n+m)$ space by noting that in a column-wise traversal we need, when computing cell (kl), to store only $dr^{k-1,\ell}$ and $db^{k,\ell-1}$, so the space is that for storing one complete column (m) and a row whose width is one box (at most n). Our multiset data structure does not increase this space complexity. Hence

Levenshtein $(A' = (a_{i_1}, p_1)(a_{i_2}, p_2) \ldots (a_{i_{m'}}, p_{m'}), B' = (b_{j_1}, r_1)(b_{j_2}, r_2) \ldots (b_{j_{n'}}, r_{n'}))$

    `/* We fill the topmost row and leftmost column first */`
$dr_0^{0,0} \leftarrow 0, \quad db_0^{0,0} \leftarrow 0$
for $k \in 1 \ldots m'$ **do**
  for $s \in 0 \ldots p_k$ **do** $dr_s^{k,0} \leftarrow dr_{p_{k-1}}^{k-1,0} + s$
  $db_0^{k,0} \leftarrow dr_{p_k}^{k,0}$
for $\ell \in 0 \ldots n'$ **do**
  for $t \in 0 \ldots r_\ell$ **do** $db_t^{0,\ell} \leftarrow db_{r_{\ell-1}}^{0,\ell-1} + t$
  $dr_0^{0,\ell} \leftarrow db_{r_\ell}^{0,\ell}$
    `/* now we fill the rest of the matrix */`
for $\ell \in 1 \ldots m'$ **do** `/* column-wise traversal */`
  for $k \in 1 \ldots n'$ **do**
    if $a_k = b_\ell$ **then** `/* equal letter box */`
      for $s \in 1 \ldots p_k$ **do**
        if $s \leq r_\ell$ **then** $dr_s^{k,\ell} \leftarrow db_{r_\ell - s}^{k-1,\ell}$ **else** $dr_s^{k,\ell} \leftarrow dr_{s-r_\ell}^{k,\ell-1}$
      for $t \in 1 \ldots r_\ell$ **do**
        if $t \leq p_k$ **then** $db_t^{k,\ell} \leftarrow dr_{p_k - t}^{k,\ell-1}$ **else** $db_t^{k,\ell} \leftarrow db_{t-p_k}^{k-1,\ell}$
    else `/* different letter box */`
      $M_l \leftarrow \{dr_0^{k,\ell-1}\}, \quad M_t \leftarrow \{db_{r_\ell}^{k-1,\ell}\}$
      $dr_0^{k,\ell} \leftarrow dr_{p_{k-1}}^{k-1,\ell}$
      for $s \in 1 \ldots p_k$ **do**
        $M_l \leftarrow M_l \cup \{dr_s^{k,\ell-1}\}$
        if $s > r_\ell$ **then** $M_l \leftarrow M_l - \{dr_{s-r_\ell-1}^{k,\ell-1}\}$
        if $r_\ell \geq s$ **then** $M_t \leftarrow M_t \cup \{db_{r_\ell - s}^{k-1,\ell}\}$
        $dr_s^{k,\ell} \leftarrow 1 + min(r_\ell - 1 + min(M_l), s - 1 + min(M_t))$
      $M_l \leftarrow \{dr_{p_k}^{k,\ell-1}\}, \quad M_t \leftarrow \{db_0^{k-1,\ell}\}$
      $db_0^{k,\ell} \leftarrow db_{r_{\ell-1}}^{k,\ell-1}$
      for $t \in 1 \ldots r_\ell$ **do**
        $M_t \leftarrow M_t \cup \{db_t^{k-1,\ell}\}$
        if $t > p_k$ **then** $M_t \leftarrow M_t - \{db_{t-p_k-1}^{k-1,\ell}\}$
        if $p_k \geq t$ **then** $M_l \leftarrow M_l \cup \{dr_{p_k - t}^{k,\ell-1}\}$
        $db_t^{k,\ell} \leftarrow 1 + min(t - 1 + min(M_l), p_k - 1 + min(M_t))$
return $dr_{p_{m'}}^{m'n'}$ `/* or` $db_{r_{n'}}^{m'n'}$ `*/`

Figure 2: The $O(m'n + n'm)$ time algorithm to compute the Levenshtein distance between A and B, coded as run-length sequences of pairs $(letter, run\_length)$.

we have

Theorem 3 *Given strings A and B of lengths m and n that are run-length encoded to lengths m' and n', there is an algorithm to calculate $D_L(A, B)$ in $O(m'n + n'm)$ time and $O(m + n)$ space in the worst case.*

This is a good point to give some intuition on the method. Figure 3 illustrates three different points along our computation of a different letter box. In principle, to fill the cell (s, t), we would need to consider all the cells $(0 \ldots s, t)$ and $(s, 0 \ldots t)$. However, we have shown in Lemma 2 that some of these cells cannot influence the final value of the cell (s, t). The reason is as follows. The cells in the grayed areas need to reach cell (s, t) through a path of vertical, horizontal and diagonal moves, which correspond to insertions, deletions and substitutions. Every such move costs 1, so the final cost is $s - 1$ for every cell in the top grayed area and $t - 1$ for every cell in the left grayed area. These costs are added to the original costs of the grayed cells. Note that the optimal paths use the diagonal moves as much as possible. The reason that permits not considering some of the top and left cells is that their shortest paths to (s, t) are longer than those of grayed cells, by an amount that equals their distance to the closest grayed cell. Since neighboring cells differ by at most 1, a non-grayed area can never compensate its farther distance to (s, t) with a smaller cell value. Finally, those grayed areas grow by one cell at a time and we manage to maintain their minimum value in constant time.

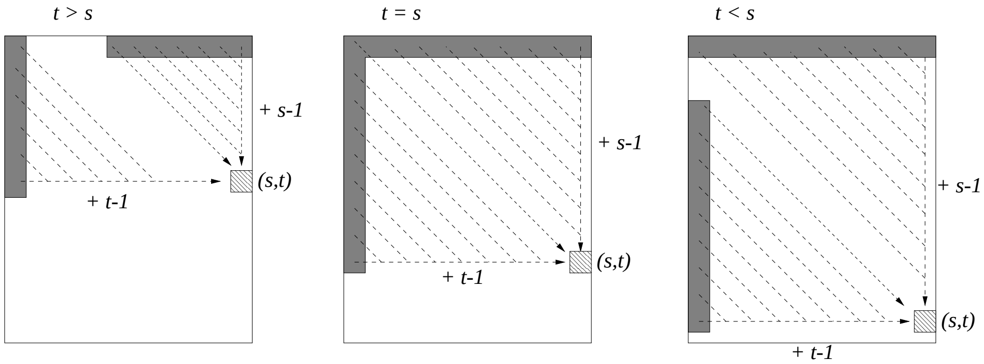

Figure 3: Different cases along the computation of a different letter box.

3.2 Multiset Data Structure

What is left is to describe our data structure to handle a multiset of natural numbers. We exploit the fact that consecutive cells in (d_{ij}) differ by at most 1 [25]. Our data structure represents the multiset S as a triple $(min(S), max(S), V_{min(S)...max(S)} \rightarrow \mathbb{N})$. That is, we store the minimum and maximum value of the multiset and a vector of counters V, which stores at V_i the number of elements equal to i in S. Given the property that consecutive cells differ by at most 1, we have that no value V_i is equal to zero. This is proved in the following lemma.

Lemma 4 *No value V_i for $min(S) \leq i \leq max(S)$ is equal to zero when S is a set of consecutive values in (d_{ij}) (i.e., S contains a contiguous part of a row or a column of the matrix (d_{ij})).*

Proof. The lemma is trivially true for the extremes $i = min(S)$ and $i = max(S)$. Let us assume that the value $min(S)$ is achieved at cell $d_{i,j}$ and that the value $max(S)$ is achieved at cell $d_{i',j'}$. Since all the intermediate cell values are also in S by hypothesis, and consecutive cells differ by at most 1, it follows that any value x between $min(S)$ and $max(S)$ exists in a path that goes from $d_{i,j}$ to $d_{i',j'}$. Hence $V_x > 0$. □

Figure 4 shows the detailed algorithms. When we initialize the data structure with the single element $S = \{x\}$ we represent the situation as $(x, x, V_x = 1)$. When we have to add an element y to S, we check whether y is outside the range $min(S) \ldots max(S)$, and in that case we extend the range. In any case we increment V_y. Note that the domain is never extended by more than one cell, as there cannot appear empty cells in between by Lemma 4. When we have to remove an element z from S we simply decrement V_z. If V_z becomes zero, Lemma 4 implies that this is because z is either the minimum or the maximum of the set. So we reduce the domain of V by one. Finally, obtaining $min(S)$ is trivial as we have it already precomputed.

It is easy to see that all the operations take constant time. As a practical matter, we note that it is a good idea to keep V in a circular array so that it can grow and shrink by any extreme. Its maximum size corresponds to p_k (for M_l) or r_ℓ (for M_t), which are known at the time of **Create**.

```
Create (x)
    return (x, x, V_x = 1)

Add ((minS, maxS, V), y)
    if y < minS then
        minS ← y
        add new first cell V_y = 0
    else if y > maxS then
        maxS ← y
        add new last cell V_y = 0
    V_y ← V_y + 1
    return (minS, maxS, V)

Remove ((minS, maxS, V), z)
    V_z ← V_z − 1
    if V_z = 0 then
        if z = minS then
            remove first cell from V
            minS ← minS + 1
        else /* z = maxS */
            remove last cell from V
            maxS ← maxS − 1
    return (minS, maxS, V)

Min ((minS, maxS, V))
    return minS
```

Figure 4: The multiset data structure implementation.

4 Extending the Algorithm to Weighted Edit Distance

In this section we show that the algorithm of Section 3 can be extended to handle an arbitrary cost function δ so that the algorithm stays in $O(mn' + m'n)$ time.

The key fact is that, inside a given box (k, ℓ), the letters in A and B are the same all the time, namely a_{i_k} and b_{j_ℓ}. Hence, there are only three different costs involved:

$$\begin{array}{rrcl} \text{(insertion)} & C_i & = & \delta(\lambda \rightarrow b_{j_\ell}) \\ \text{(deletion)} & C_d & = & \delta(a_{i_k} \rightarrow \lambda) \\ \text{(substitution)} & C_s & = & \delta(a_{i_k} \rightarrow b_{j_\ell}) \end{array} \qquad (7)$$

where, since the triangle inequality holds, $C_s \leq C_i + C_d$.

We will not differentiate between equal and different letter boxes in this section, since $C_s \leq C_i + C_d$ in both cases. Also, to simplify the exposition, we assume w.l.o.g. that the costs C_i and C_d are the same in all boxes (if this is not the case, we can redefine boxes not to overlap at borders and use the basic recurrence to compute the values in the left and top borders inside the current box from the borders of preceding boxes).

Several problems have to be dealt with. We first consider how to compute the path costs and how to determine the relevant cells, then how to update the path costs in constant time, and finally how to handle our multiset under a more general scenario.

4.1 Determining Relevant Cells and Path Costs

The following lemma shows that the path costs can still be computed in constant time and that the cells that are relevant to the computation of $d_{s,t}^{k,\ell}$ are exactly the same as for the Levenshtein distance.

Lemma 5 *If δ is the cost function, then it holds that*

$$\begin{aligned} d_{s,t}^{k,\ell} = min \quad (& \quad path(s-q,t) + min_{max(0,s-t) \leq q \leq s} \; d_{q,0}^{k,\ell} \;, \\ & \quad path(s,t-q) + min_{max(0,t-s) \leq q \leq t} \; d_{0,q}^{k,\ell} \quad), \end{aligned} \tag{8}$$

where $1 \leq s \leq p_k$ and $1 \leq t \leq r_\ell$, for values $d_{s,t}^{k,\ell}$. The function path is defined as

$$path(d,r) \quad = \quad C_s min(d,r) + C_d max(d-r,0) + C_i max(r-d,0)$$

where C_d, C_i and C_s are as defined in Eq. (7).

Proof. In order to determine the cost of a path from (s',t') to (s,t), we observe that the optimal path uses as many diagonal moves as possible, so its cost is $(t-t')C_s + ((s-s')-(t-t'))C_d$ if $s-s' \geq t-t'$, and $(s-s')C_s + ((t-t')-(s-s'))C_i$ otherwise. The formula for *path* is easily derived from this observation.

It remains to show that the cells not considered in the minimization are not necessary. Let us first assume that $s > t$ and consider including the previous cell in the first row of the minimization formula (8). Let us call $q_0 = s-t$ the row index of the first relevant cell and $q' = s-t-1 \geq 0$ that of the previous cell. Because of the formula to compute the extended edit distance (1), $d^{k,\ell}_{q_0,0} \leq d^{k,\ell}_{q',0} + \delta(a_{i_k} \to \lambda)$, so $d^{k,\ell}_{q',0} \geq d^{k,\ell}_{q_0,0} - C_d$. On the other hand, $path(s-q',t) = C_s t + C_d = path(s-q_0,t) + C_d$. Therefore, $path(s-q',t) + d^{k,\ell}_{q',0} \geq path(s-q_0,t) + d^{k,\ell}_{q_0,0}$, which means that adding the previous cell $(q',0)$ in the minimization does not change the final value. The argument can be inductively repeated with any previous cell. The case $s \leq t$ is trivial since there are no previous cells. This proof applies equally to the second row of the formula (8) for the cells on the top. $\square$

4.2 Updating Path Costs

In the Levenshtein distance all the paths arriving to cell (s,t) cost the same ($t-1$ or $s-1$ depending on the case). So we maintain a set of cell values, take the minimum, and add the invariant path cost to them. Under weighted edit distance, we need to add the path cost to the cell values before looking for the minimum. Hence we will store in our multiset the cell values with the path costs added.

The problem is that all the path costs change as we move from (s,t) to $(s+1,t)$ or $(s,t+1)$ in the algorithm of Figure 2. To avoid the need of updating all the values of the multiset, we will store a different, invariant value that does not alter the order relationship between cell costs (so the minimum will be the same, and after it is chosen we will compute its real value in constant time).

Let us consider the first row of formula (8), that is, the values stored in the multiset M_l, as M_t is analogous. The area of relevant cells in the leftmost column is $max(0, s-t) \ldots s$. As in the algorithm in Figure 2, we need to consider two cases: (i) the area is extended from $p_k \ldots p_k$ to $max(0, p_k - r_\ell) \ldots p_k$ by increasing t from 0 to r_ℓ; (ii) the area is extended from $0 \ldots 0$ to $max(0, p_k - r_\ell) \ldots p_k$ by increasing s from 0 to p_k. Let us focus on (i). As t is increased by 1, all the previous $path$ costs must be incremented by C_i because they use one more horizontal move (insertion). Instead of incrementing all previous values by C_i, we can subtract an amount from each new value that is included in the set of relevant values, so that the values in the set remain comparable. This amount is $t \times C_i$ for the $(t+1)$-th element that is included in the set. Since the new value that is included in the set uses only diagonal moves (substitutions), the absolute value that is stored for the $(t+1)$-th cell included in the multiset is $d^{k,\ell}_{p_k-t,0}+t\times C_s-t\times C_i$. When calculating the value of cell $d^{k,\ell}_{p_k,t}$, we can look for the minimum as before, and after it is found, we can retrieve its original value by adding $t \times C_i$.

Let us now consider case (ii). As we move from s to $s+1$, all the previous paths gain one diagonal move (substitution) and lose one horizontal move (insertion), so this time the amount to subtract is $s \times (C_s - C_i)$ at the $(s+1)$-th step. This is also the value to add to the minimum after it is retrieved. Again, we can see that the path of the new cell that is included uses only horizontal moves (insertions), so the absolute value to store for it is $d^{k,\ell}_{s,0} + (r_\ell + s) \times C_i - s \times C_s$.

A related problem is how to determine the value of the cells that have to be removed from the multiset, since we have stored it with the path cost added and the invariant value subtracted. This corresponds to case (ii) only. As the amount by which all had been shifted at the $(x+1)$-th step was $x \times (C_s - C_i)$, and the element removed is $(x,0) := (s-r_\ell, 0)$, we remove from the multiset the value $d^{k,\ell}_{x,0}+(r_\ell+x)\times C_i - x\times C_s = d^{k,\ell}_{s-r_\ell,0} + s \times C_i - (s - r_\ell) \times C_s$.

4.3 Managing the Multiset

Now we are faced with the final problem: how to handle the multiset operations in constant time. The differences between consecutive cells need not be in the set $\{-1, 0, +1\}$, so the preconditions for our previous multiset implementation do not hold anymore. We show that we can still implement the multiset in constant time per operation.

Let us first consider the case where δ gives integer values. The maximum difference between two consecutive cell values is $v^* = \max(C_i, C_d) = O(1)$. Since now we add the path values to the cell costs before inserting them in the multiset, the difference between consecutive values can be as large as $2v^*$, which is still constant. Hence we can implement the multiset with an array V of counters as before. Unlike the Levenshtein case, V will have zero entries, but there will never be more than $2v^* - 1$ consecutive zero entries. Therefore, inserting a new entry may force us to initialize up to $2v^*$ cells (instead of only 1 as in lines 4 and 7 of Figure 4), and removing an entry may force us to remove up to $2v^*$ cells before finding the next nonzero entry (instead of only 1 as in lines 13 and 16 of Figure 4). All this costs $O(v^*)$, which is constant. Note also that it is not true any longer, as assumed in line 15 of Figure 4, that cells that become zero must necessarily lie at the limit of the multiset.

If the δ function delivers real values, the above solution does not work. Implementing the set as a priority queue adds an $O(\log max(m, n))$ factor to the time. However, we can use min-deques [13] instead, which allow handling a queue of elements by adding and removing elements from both ends, as well as taking the minimum over the queue. All these operations can be performed in $O(1)$ worst-case amortized time by using simple techniques, and $O(1)$ worst case time per operation with more sophisticated ones. Hence the total time stays the same.

An illustration of the algorithm is shown in Fig. 5. The example shows how to derive the values of the last row by using the values of the leftmost column. Note that the values in the first row should also be taken into account, but for brevity we only consider the leftmost column. A transition from value $t = 3$ to value $t = 4$ is shown in the example. First, value 3 is included in the set M_l after adding the path value $t \times C_s = 4$ and subtracting the value $t \times C_i = 8$. The minimum of the set M_l is now -1, and the value $t \times C_i = 8$ is added to get the value of X.

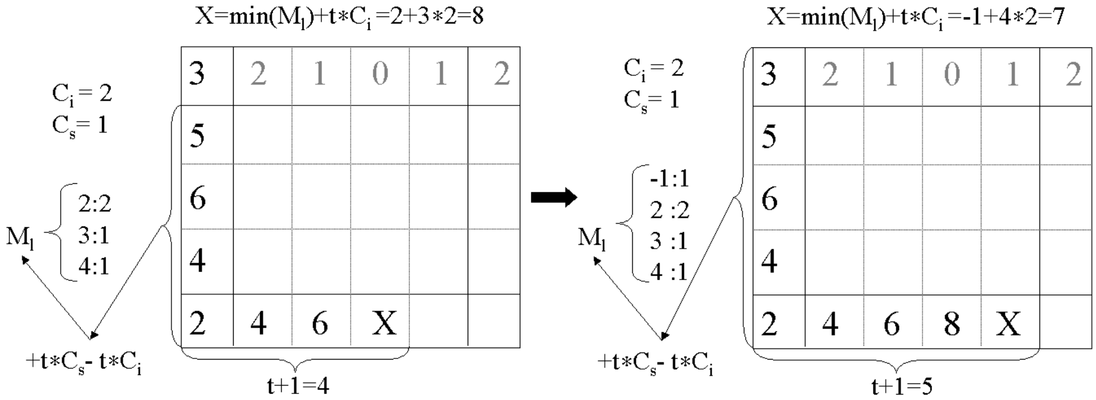

Figure 5: An example of the evaluation of the weighted edit distance.

5 Approximate Searching

Let us now consider a problem related to computing the LCS or the edit distance. Assume that string A is a short pattern and string B is a long text (so m is much smaller than n), and that we are given a threshold parameter k. We are interested in reporting all the "approximate occurrences" of A in B, that is, all the positions of text substrings which are at distance k or less from the pattern A. In order to ensure a linear size output, we content ourselves with reporting the ending positions of the occurrences (which we call "matches").

The classical algorithm to find all the matches [23] computes a matrix exactly like those of recurrences (3) and (2), with the only difference that $d_{0,j} = 0$. This permits the occurrences to start at any text position. The last row of the matrix, $d_{m,j}$, is examined and every text position j such that $d_{m,j} \leq k$ is reported as a match.

Our goal now is to devise a more efficient algorithm when pattern and text are run-length compressed. A trivial $O(m^2n' + R)$ algorithm (where R is the size of the output) is obtained as follows. We start filling the matrix only at beginnings of text runs, and complete the first m columns only (at $O(m^2)$ cost). The rest of the columns of the run are equal to the m-th because no optimal path can span more than m columns under the LCS or Levenshtein models (m deletions are enough to convert an empty substring of the text into the pattern). We later examine the last row of the matrix and report every text position with value $\leq k$. If the run is longer than m, then we have not produced the whole last row but only the first m cells of it. In this case we report the positions $m + 1 \ldots r_\ell$ of the ℓ-th run if and only if the position m was reported.

We improve now the trivial algorithm. A first attempt is to apply our algorithms directly using the new base value $d_{0,j} = 0$. This change does not present complications.

Let us first concentrate on the Levenshtein distance. Our algorithm obtains $O(m'n+ n'm)$ time, which may or may not be better than the trivial approach. The problem is that $O(m'n)$ may be too much in comparison to $O(m^2n')$, especially if n is much larger than m. We seek for an algorithm proportional to the compressed text size. We divide the text runs in *short* (of length at most m) and *long* (longer than m) runs. We apply our Levenshtein algorithm on the text runs, filling the matrix column-wise. If we have a short run (b_{j_ℓ}, r_ℓ), $r_\ell \leq m$, we compute all the $m'+1$ horizontal borders plus its final vertical border (which becomes the initial border of the next column). The time to achieve this is $O(m'r_\ell + m)$. For an additional $O(r_\ell)$ cost we examine all the cells of the last row and report all the text positions $j_\ell + t$ such that $d^{m',\ell}_{p_{m'},t} \leq k$.

If we have a long run (b_{j_ℓ}, r_ℓ), $r_\ell > m$, we limit its length to m and apply the same algorithm, at $O(m'm + m + m)$ cost. The columns $m+1 \ldots r_\ell$ of that run are equal to the m-th, so we just need to examine the last row of the m-th column, and report all the text positions up to the end of the run, $j_\ell + m + 1 \ldots j_\ell + r_\ell$, if $d^{m',\ell}_{p_{m'},m} \leq k$.

This algorithm takes $O(n'm'm + R)$ time in the worst case, where R is the number of occurrences reported. The space requirement is that to compute one text run limited to length m, i.e. $O(m'm)$. For the LCS model we have the same upper bound of m, so we achieve the same complexity. Our $O(m'n'(m'+n'))$ algorithm to be presented in Section 6 does not yield a good complexity here.

Note that if we are allowed to represent the occurrences as a sequence of *runs* of consecutive text positions (all of which match), then the extra term R of the search cost disappears.

Theorem 6 *Given a pattern A and a text B of lengths m and n that are run-length encoded to lengths m' and n', there is an algorithm to find all the ending points of the approximate occurrences of A in B, either under the LCS or Levenshtein model, in $O(m'mn')$ time and $O(m'm)$ space in the worst case.*

The above result generalizes easily for the case of weighted edit distance using the methods from Section 4. The limit between short and long runs depends in this case on the δ values, but it is still $O(m)$. For integer-valued costs for the edit operations, we have the same bound as before, $O(m'mn')$.

6 Improving a Greedy Algorithm for the LCS

The idea in our algorithm for the Levenshtein distance D_L in Section 3 was to fill all the borders of all the boxes ($d^{k,\ell}_{s,t}$). The natural way to reduce the complexity would be to fill only the corners of the boxes (see Figure 1). For the D_L distance this seems difficult to obtain, but for the D_{ID} distance there is an obvious greedy algorithm that achieves this goal: in different letter boxes, we can calculate the corner values in constant time, and in equal letter boxes we can trace an optimal path to a corner in $O(m' + n')$ time. Thus, we can calculate all the corner values in $O(m'n'(m' + n'))$ time[1].

It turns out that we can improve the greedy algorithm significantly by fairly simple means. We notice that the diagonal method of [24] can be applied, and yields an $O(d^2 min(n'm'))$ algorithm, where $d = D_{ID}(A, B)$. We also give other improvements that do not affect the worst case, but are significant in the average case and in practice. We end the section conjecturing that our improved algorithm runs in $O(m'n')$ time on the average. As we are unable to prove this conjecture, we provide experimental evidence to support it.

6.1 Greedy Algorithm for the LCS

Calculating the corner value $d^{k,\ell}_{p_k,r_\ell}$ in a different letter box is easy, because it can be retrieved from the values $d^{k,\ell}_{0,r_\ell} = d^{k-1,\ell}_{p_{k-1},r_\ell}$ and $d^{k,\ell}_{p_k,0} = d^{k,\ell-1}_{p_k,r_{\ell-1}}$, which are calculated earlier during the dynamic programming. This follows from the lemma:

Lemma 7 (Bunke and Csirik [8]) *The recurrence (3) can be replaced by the recurrence*

$$d^{k,\ell}_{s,t} = min(d^{k,\ell}_{s,0} + t, d^{k,\ell}_{0,t} + s), \tag{9}$$

where $1 \le s \le p_k$ and $1 \le t \le r_\ell$, for values $d^{k,\ell}_{s,t}$ in a different letter box.

[1] Apostolico et. al. [4] also gave a basic $O(m'n'(m' + n'))$ algorithm for the LCS, which they then improved to $O(m'n' \log(m'n'))$. Their basic algorithm differs from our greedy algorithm in that they were using the recurrence for computing the LCS directly, and we are computing the distance D_{ID}. Furthermore, they traced a specific optimal path (which was the property that they could use to achieve the $O(m'n' \log(m'n'))$ algorithm).

In contrast to the D_L distance, the difficult part in D_{ID} distance lies in equal letter boxes. As noted earlier, Lemma 1 also applies for the D_{ID} distance. From Lemma 1 we can see that the corner values are retrieved along the diagonal, and those values may not have been calculated earlier. However, if $p_k = r_\ell$ in all equal letter boxes, then each corner $d^{k,\ell}_{p_k,r_\ell}$ can be calculated in constant time. This gives an $O(m'n')$ algorithm for a (very) special case, as previously noted in [7].

What follows is an algorithm to retrieve the value $d^{k,\ell}_{p_k,r_\ell}$ in an equal letter box in $O(m'+n')$ time. The idea is to trace an optimal path to the cell $d^{k,\ell}_{p_k,r_\ell}$. This can be done by using Lemmas 1 and 7 recursively. Assume that $d^{k,\ell}_{p_k,r_\ell} = d^{k,\ell}_{0,r_\ell-p_k}$ by Lemma 1 (case $d^{k,\ell}_{p_k,r_\ell} = d^{k,\ell}_{p_k-r_\ell,0}$ is symmetric). If $k=1$, then the value $d^{1,\ell}_{0,r_\ell-p_k}$ corresponds to a value in the first row (0) of the matrix (d_{ij}) which is known. Otherwise, the box $(d^{k-1,\ell}_{s,t})$ is a different letter box, and using the definition of overlapping boxes and Lemma 7 it holds

$$d^{k,\ell}_{0,r_\ell-p_k} = d^{k-1,\ell}_{p_{k-1},r_\ell-p_k} = min(d^{k-1,\ell}_{p_{k-1},0} + r_\ell - p_k, d^{k-1,\ell}_{0,r_\ell-p_k} + p_{k-1}).$$

Now, the value $d^{k-1,\ell}_{p_{k-1},0}$ is calculated during the dynamic programming, so we can continue tracing value $d^{k-1,\ell}_{0,r_\ell-p_k}$ using Lemmas 1 and 7 recursively until we meet a value that has already been calculated during dynamic programming (including the first row and the first column of the matrix (d_{ij})). The recursion never branches, because Lemma 1 defines explicitly the next value to trace, and one of the two values (from which the minimum is taken over in Lemma 7) is always known (that is because we enter the different letter boxes at the borders, and therefore the other value is from a corner that is calculated during the dynamic programming). We call the path described by the recursion a *tracing path*.

Tracing the value $d^{k,\ell}_{p_k,r_\ell}$ in an equal letter box may take $O(m'+n')$ time, because we are skipping one box at a time, and there are at most $m'+n'$ boxes in the tracing path. Therefore, we get an $O(m'n'(m'+n'))$ algorithm to calculate $D_{ID}(A,B)$. A worst case example that actually achieves the bound is $A = a^n$ and $B = (ab)^{n/2}$.

The space requirement of the algorithm is $O(m'n')$, because we need to store only the corner value in each box, and the $O(m'+n')$ space for the stack is not needed because the recursion does not branch.

We also achieve the $O(m'n + n'm)$ bound, because the corner values $d^{k,\ell}_{p_k,r_\ell}$ of equal letter boxes define distinct tracing paths, and therefore each cell in the borders of the boxes can be visited only once. To see this observe that each border cell reached by a tracing path uniquely determines the border cell it comes from along the tracing path, and therefore no two different paths can meet in a border cell. The only exception is a corner cell, but in this case all the tracing paths end there immediately.

Theorem 8 *Given strings A and B of lengths m and n that are run-length encoded to lengths m' and n', there is an algorithm to calculate $D_{ID}(A, B)$ in $O(min(m'n'(m' + n'), m'n + n'm))$ time and $O(m'n')$ space.*

6.2 Diagonal Algorithm

The diagonal method [24] provides an $O(d\,min(m, n))$ algorithm for calculating the distance $d = D_{ID}(A, B)$ (or D_L as well) between strings A and B of length m and n, respectively. The idea is the following: the value $d_{m,n} = D_{ID}(A, B)$ in the $(d_{i,j})$ matrix of recurrence (3) defines a diagonal band, where the optimal path must lie. Thus, if we want to check whether $D_{ID} < k$, we can limit the calculation to the diagonal band defined by value k (consisting of $O(k)$ diagonals). Starting with $k = |n - m| + 1$, we can double the value k and run in each step the recurrence (3) on the increasing diagonal band. As soon as $d_{m,n} < k$, we have found $D_{ID}(A, B) = d_{m,n}$, and we can stop the doubling. The total number of diagonals evaluated is at most $2\,D_{ID}(A, B)$, and there are at most $min(m, n)$ cells in each diagonal. Therefore, the total cost of the algorithm is $O(d\,min(m, n))$, where $d = D_{ID}(A, B)$.

We can use the diagonal method with our greedy algorithm as follows. We calculate only the corner values that are inside the diagonal band defined by value k in the above doubling algorithm. The corner values in equal letter boxes inside the diagonal band can be retrieved in $O(k)$ time. That is because we can limit the length of the tracing paths with the value $2k + 1$ (between two equal letter boxes there is a different letter box that contributes at least 1 to the value that we are tracing, and we are not interested in corner values that are greater than k). Therefore, we get the total cost $O(d^2 min(m', n'))$, where $d = D_{ID}(A, B)$.

6.3 Faster on Average

There are some practical refinements for the greedy algorithm that do not improve its worst case behavior, but do have an impact on its average case.

Skipping runs of different letter boxes in tracing paths. Consider two consecutive different letter boxes $(d^{k,\ell}_{s,t})$ and $(d^{k+1,\ell}_{s,t})$. By Lemma 7 it holds for the values $1 \leq t \ \leq r_\ell$,

$$\begin{aligned}
d^{k+1,\ell}_{p_{k+1},t} &= min\left(d^{k+1,\ell}_{0t} + p_{k+1}, d^{k+1,\ell}_{p_{k+1},0} + t\right) \\
&= min\left(d^{k,\ell}_{p_k,t} + p_{k+1}, d^{k+1,\ell}_{p_{k+1},0} + t\right) \\
&= min\left(d^{k,\ell}_{0t} + p_k + p_{k+1}, d^{k,\ell}_{p_k,0} + p_{k+1} + t, d^{k+1,\ell}_{p_{k+1},0} + t\right) \\
&= min\left(d^{k,\ell}_{0t} + p_k + p_{k+1}, d^{k+1,\ell}_{p_{k+1},0} + t\right).
\end{aligned}$$

The above result can be extended to the following lemma by using induction:

Lemma 9 *Let $((d^{k',\ell}_{s,t}), (d^{k'+1,\ell}_{s,t}), \ldots, (d^{k,\ell}_{s,t}))$ and $((d^{k,\ell'}_{s,t}), (d^{k,\ell'+1}_{s,t}), \ldots, (d^{k,\ell}_{s,t}))$ be vertical and horizontal runs of different letter boxes. When $1 \leq t \leq r_\ell$ and $1 \leq s \leq p_k$, the recurrence (4) can be replaced by the recurrences*

$$\begin{aligned}
d^{k,\ell}_{p_k,t} &= min\left(d^{k,\ell}_{p_k,0} + t, d^{k',\ell}_{0,t} + \sum_{s=k'}^{k} p_s\right) & 1 \leq t \leq r_\ell, \\
d^{k,\ell}_{s,r_\ell} &= min\left(d^{k,\ell}_{0,r_\ell} + s, d^{k,\ell'}_{s,0} + \sum_{t=\ell'}^{\ell} r_t\right) & 1 \leq s \leq p_k.
\end{aligned}$$

Now it is obvious how to speed up the retrieval of values $d^{k,\ell}_{p_k,r_\ell}$ in the equal letter boxes. During dynamic programming, we can maintain pointers in each different letter box to the last equal letter box encountered in the direction of the row and the column. When we enter a different letter box while tracing the value of $d^{k,\ell}_{p_k,r_\ell}$ in an equal letter box, we can use Lemma 9 to calculate the minimum over the run of different letter boxes at once, and continue on tracing from the equal letter box preceding the run of different letter boxes. (Note that in order to use the summations of Lemma 9 we should better store the cumulative i_k and j_ℓ values instead of p_k and r_ℓ.) Therefore we get the following result:

Theorem 10 *Given strings A and B of lengths m and n that are run-length encoded to lengths m' and n', such that the letters of the runs are independently and uniformly distributed over an alphabet of size $|\Sigma|$, there is an algorithm to calculate $D_{ID}(A,B)$ in $O(m'n'(1+(m'+n')/|\Sigma|^2))$ time on the average.*

Proof. The first part of the cost, $O(m'n')$ comes from the constant time computation of all the different letter boxes.

On the other hand, there are on the average $O(m'n'/|\Sigma|)$ equal letter boxes. This can be seen as follows: Consider the box model of Figure 1. The equal letter boxes in a row of the matrix correspond to the same character, say $\sigma \in \Sigma$. Let X_j be a random variable to denote the amount of different letter boxes between the jth and $(j-1)$th equal letter box in a row (without the lack of generality, we may assume that a row starts and ends with an equal letter box). It is an easy exercise to see that the expected value of each X_j is $|\Sigma| - 1$. We can use this to estimate the number of equal letter boxes in a row, denoted by ϕ, because we can write

$$\sum_{j=1}^{\phi} X_j + 1 < n'. \tag{10}$$

We are interested in the first value of ϕ such that (10) does not hold. Using a result from *renewal theory*, the expected value of such a ϕ is $O(n'/|\Sigma|)$ (see p. 359 in [11]; the result requires that the variables X_j are independent, which is our case). Using the linearity of expectation, the expected number of equal letter boxes in the whole matrix is just the sum of the equal letter boxes in all rows, that is $O(m'n'/|\Sigma|)$.

To get the claimed bound $O(m'n'(1+(m'+n')/|\Sigma|^2))$, it remains to show that the expected amount of calculation in an equal letter box is $O((m'+n')/|\Sigma|)$. This is the amount of equal letter boxes visited by a tracing path. We can use a similar argument as when calculating the amount of equal letter boxes in a row. Let X_j be a random variable to denote the amount of different letter boxes between jth and $(j-1)$th equal letter box in a tracing path (again, we may assume that a tracing path starts and ends with an equal letter box). Notice that the string that is the concatenation of the characters in a tracing path has similar distribution as the strings A and B. Thus the expected value of each X_j is $|\Sigma| - 1$. As a tracing path can visit at most $m' + n'$ boxes, we can write

$$\sum_{j=1}^{\phi} X_j + 1 < m' + n', \tag{11}$$

where ϕ is the number of equal letter boxes in a tracing path. As previously, the expected value of the first ϕ such that (11) does not hold is $O((n' + m')/|\Sigma|)$. $\Box$

Using bridges to prune tracing paths. The second improvement to the greedy algorithm is to limit the length of the tracing paths. In the greedy algorithm the tracing is continued until a value is reached that has been calculated during the dynamic programming. However, there are more known values than those that have been explicitly calculated. Consider value $d^{k,\ell}_{p_k,t}$, $1 \le t \le r_\ell$ (or symmetrically $d^{k,\ell}_{s,r_\ell}$, $1 \le s \le p_k$) in the border of a different letter box. If $d^{k,\ell}_{p_k,r_\ell} = d^{k,\ell}_{p_k,0} + r_\ell$ then it must hold that $d^{k,\ell}_{p_k,t} = d^{k,\ell}_{p_k,0} + t$, otherwise we get a contradiction: $d^{k,\ell}_{p_k,r_\ell} < d^{k,\ell}_{p_k,0} + r_\ell$.

We call the above situation a horizontal (vertical) *bridge*. Note that from Lemma 7 it follows that there is either a vertical or a horizontal bridge in each different letter box. When we enter a different letter box in the recursion, we can check whether the bridge property holds at the border we entered, using the corner values that are calculated during the dynamic programming. Thus, we can stop the recursion at the first bridge encountered. To combine this improvement with the algorithm that skips runs of different letter boxes, we need Lemma 11 below that states that the bridges propagate along runs of different letter boxes. Therefore we only need to check whether the last different letter box has a bridge to decide whether we have to skip to the next equal letter box. The resulting algorithm is given in pseudo-code in Figure 6. An illustration of the algorithm is shown in Figure 7.

Lemma 11 *Let $((d^{k',\ell}_{s,t}), (d^{k'+1,\ell}_{s,t}), \ldots, (d^{k,\ell}_{s,t}))$ be a vertical run of different letter boxes. If there is a horizontal bridge $d^{k',\ell}_{p_{k'},r_\ell} = d^{k',\ell}_{p_{k'},0} + r_\ell$ then there is a horizontal bridge*

$d^{k'',\ell}_{p_{k''},r_\ell} = d^{k'',\ell}_{p_{k''},0} + r_\ell$ *for all* $k' < k'' \leq k$. *The symmetric result holds for horizontal runs of different letter boxes.*

Proof. We use the counter-argument that $d^{k'',\ell}_{p_{k''},r_\ell} = d^{k'',\ell}_{p_{k''},0} + r_\ell$ does not hold for some $k' < k'' \leq k$. Then by Lemma 9 and by the bridge assumption it holds

$$d^{k'',\ell}_{p_{k''},r_\ell} = d^{k'+1,\ell}_{0,r_\ell} + \sum_{s=k'+1}^{k''} p_s = d^{k'+1,l}_{0,0} + r_\ell + \sum_{s=k'+1}^{k''} p_s.$$

On the other hand, using the counter-argument and the fact that consecutive cells in the (d_{ij}) matrix differ at most by 1 [25], we get

$$d^{k'',\ell}_{p_{k''},r_\ell} < d^{k'',\ell}_{p_{k''},0} + r_\ell \leq d^{k'+1,\ell}_{0,0} + \left(\sum_{s=k'+1}^{k''} p_s \right) + r_\ell,$$

which is a contradiction and so the original proposition holds. □

Lemma 11 has a corollary: if the last different letter box in a run does not have a horizontal (vertical) bridge, then none of the boxes in the same run have a horizontal (vertical) bridge and, on the other hand, all the boxes in the same run must have a vertical (horizontal) bridge.

Now, if two tracing paths cross inside a box (or run thereof), then one of them necessarily meets a bridge. In the average case, there are a lot of crossings of the tracing paths and the total cost for tracing the values in equal letter boxes decreases significantly.

Another way to consider the average length of a tracing path is to think that every time a tracing path enters a different letter box, it has some probability to hit a bridge. If the bridges were placed randomly in the different letter boxes, then the probability to hit a bridge would be $\frac{1}{2}$. This would give immediately a constant expected length for a tracing path. However, the placing of the bridges depends on the computation of recurrence (3), and this makes the probabilistic reasoning much more complex. We are still confident that the following conjecture holds, although we have not been able to prove it.

Conjecture 12 *Let* A *and* B *be strings that are run-length encoded to lengths* m' *and* n'*, such that the lengths of the runs are equally distributed in both strings. Under these*

LCS $(A' = (a_{i_1}, p_1)(a_{i_2}, p_2) \ldots (a_{i_{m'}}, p_{m'}), B' = (b_{j_1}, r_1)(b_{j_2}, r_2) \ldots (b_{j_{n'}}, r_{n'}))$

/* We use structure $d^{k,\ell}$ to denote a box $(d^{k,\ell}_{s,t})$ as follows: */
/* $d^{k,\ell}.corner := d^{k,\ell}_{p_k, r_\ell}$ */
/* $d^{k,\ell}.jumptop :=$ location of the next equal letter box above */
/* $d^{k,\ell}.jumpleft :=$ location of the next equal letter box in the left */
/* $d^{k,\ell}.sumtop :=$ **if** $a_{i_k} \neq b_{j_\ell}$ **then** $\sum_{t=d^{k,\ell}.jumptop+1}^{k} p_t$ */
/* $d^{k,\ell}.sumleft :=$ **if** $a_{i_k} \neq b_{j_\ell}$ **then** $\sum_{t=d^{k,\ell}.jumpleft+1}^{\ell} r_t$ */
/* Initialize first row and column (let $a_{i_0} = b_{j_0} = \epsilon, p_0 = r_0 = 1$) */
$d^{00}.corner \leftarrow 0$
for $k \in 1 \ldots n'$ **do** $d^{k,0}.corner \leftarrow d^{k-1,0}.corner + r_{k-1}$
for $\ell \in 1 \ldots m'$ **do** $d^{0,\ell}.corner \leftarrow d^{0,\ell-1}.corner + p_{\ell-1}$
compute all the values $d^{k,\ell}.(jumptop, jumpleft, sumtop, sumleft)$
`/* now we fill the rest of the corner values */`
for $k \in 1 \ldots m'$ **do**
  for $\ell \in 1 \ldots n'$ **do**
    $(bridge, k', \ell', p, r, sum, d^{k,\ell}.corner) \leftarrow (false, k, \ell, p_k, r_\ell, 0, \infty)$
    if $a_{i_k} \neq b_{j_\ell}$ **then** `/* different letter box */`
      $d^{k,\ell}.corner \leftarrow min(d^{k-1,\ell}.corner + a_{i_k}, d^{k,\ell-1}.corner + b_{j_\ell})$
    else while $bridge = false$ **do**
      `/* equal letter box, trace` $d^{k,\ell}.corner$ `*/`
      if $p = r$ **then** `/* straight from the diagonal */`
        $d^{k,\ell}.corner \leftarrow min(d^{k,\ell}.corner, sum + d^{k'-1,\ell'-1}.corner)$
        $bridge \leftarrow true$
      else if $p < r$ **then** `/* diagonal up */`
        $(r, k') \leftarrow (r - p, k' - 1)$
        $d^{k,\ell}.corner \leftarrow min(d^{k,\ell}.corner, sum + d^{k',\ell'-1}.corner + r)$
        if $d^{k',\ell'}.corner = d^{k',\ell'-1}.corner + r_{\ell'}$ **then** $bridge \leftarrow true$
        else `/* jump to the next equal letter box */`
          $(sum, k') \leftarrow (sum + d^{k',\ell'}.sumtop, d^{k',\ell'}.jumptop)$
          $p \leftarrow p_{k'}$
          if $k' = 0$ **then** `/* first row */`
            $d^{k,\ell}.corner \leftarrow min(d^{k,\ell}.corner,$
              $sum + d^{k',\ell'-1}.corner + r)$
            $bridge \leftarrow true$
      else $\ldots$ `/* diagonal left similarly*/`
return $(m + n - d^{m',n'}.corner)/2$ `/* return the length of the LCS */`

Figure 6: The improved greedy algorithm for computing the LCS between A and B, coded as run-length sequences of pairs $(letter, run\_length)$.

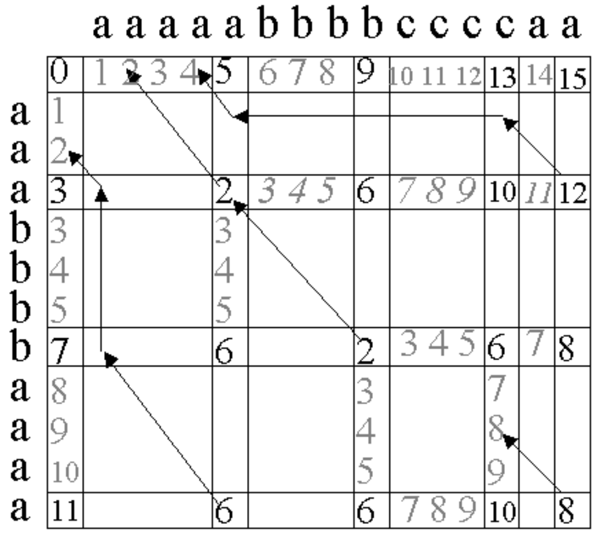

Figure 7: Evaluating the LCS between strings $A = aaabbbbaaaa$ and $B = aaaaabbbbccccaa$ using the algorithm in Figure 6. The gray values denote the bridges, thus these values are not explicitly computed, but they can be deduced from the corner values.

assumptions the expected running time of the algorithm in Figure 6 for calculating $D_{ID}(A, B)$ *is* $O(m'n')$.

6.4 Experimental Results

To test Conjecture 12, we ran the algorithm in Figure 6 with the following settings:

1. $m' = n' = 2000$, $|\Sigma| = 2$, runs in $[1, x]$,
   $x \in \{1, 10, 100, 1000, 10000, 100000, 1000000\}$.

2. $m' = 2000, n' \in \{1, 50, 100, 500, 1000, 1500, 2000\}, |\Sigma| = 2$, runs in $[1, 1000]$.

3. $m' = n' = 2000, |\Sigma| \in \{2, 4, 8, 16, 32, 64, 128, 256\}$, runs in $[1, 1000]$.

4. String A was as in item 1 with runs in $[1, 1000]$. String B was generated by applying k random insertions/deletions on A, where $k \in \{0, 1, 10, 100, 1000, 10000, 100000\}$.

5. Real data: three different black/white images (printed lines from a book draft

Table 1: The average length and the maximum length of a tracing path was measured in different test settings. The values of tests 1-4 are averages over 10-10,000 trials (e.g. on small values of n' in test 2, more trials were needed because of high variance, whereas otherwise the variance was small). Test 5 was deterministic (i.e. the values are from one trial).

	Average length of a tracing path (maximum length)
test X	setting 1, setting 2, ...
test 1	1 (1), 1.71 (18), 1.96 (28), 1.98 (27), 1.98 (32), 1.99 (29), 1.98 (25)
test 2	1.73 (5), 1.77 (10), 1.74 (13), 1.80 (21), 1.90 (30), 1.97 (35), 1.98 (38)
test 3	1.99 (30), 1.77 (20), 1.60 (14), 1.45 (14), 1.33 (9), 1.24 (7), 1.17 (6), 1.13 (6)
test 4	1.71 (9), 1.71 (8), 1.71 (7), 1.71 (10), 1.72 (9), 1.72 (10), 1.72 (12)
test 5	2.00 (35), 2.34 (146), 2.32 (31)

(187×591), technical drawing (160×555), and a signature (141×362)). We ran the LCS algorithm on all pairs of lines in each image.

Table 1 shows the results. Different parameter choices are listed in the order they appear in the above listing (e.g. setting 1 in test 1 corresponds to $x = 1$, setting 2 corresponds to $x = 10$, etc.).

The average length L of a tracing path (i.e. the amount of equal letter boxes visited by a tracing path) was smaller than 2 in tests 1-4 (slightly greater in test 5). That is, the running time was in practice $O(m'n')$ with a very small constant factor. Test 1 showed that when the mean length of the runs increases, then L also increases, but not exceeding 2 ($L \in [1, 1.99]$). In test 2, the worst situation was with $n' = m'$ ($L = 1.98$). We tested the effect of the alphabet in test 3, and the worst was $|\Sigma| = 2$ ($L = 1.99$) and the best was $|\Sigma| = 256$ ($L = 1.13$). Test 4 was used to simulate a typical situation, in which the distance between the strings is small. The amount of errors did not have much influence ($L \in [1.71, 1.72]$). In real data (test 5), there were pairs close to the worst case ($A = a^n, B = (ab)^{n/2}$), and therefore the results were slightly worse than with randomly generated data: $L \in \{2.00, 2.34, 2.31\}$ with the three images. Of course real data does not need to fit the hypothesis of our conjecture.

7 Conclusions

We have presented new algorithms for approximate matching of run-length compressed strings. The previous algorithms [8, 4] permit computing their LCS. We have presented a new LCS algorithm with improved average complexity. We have also extended an LCS algorithm [8] to a more general weighted edit distance model (in particular to the Levenshtein distance) without increasing its complexity. Finally, we have presented an algorithm with nontrivial complexity for approximate searching of a run-length compressed pattern on a run-length compressed text under either model.

A possible application for the edit distance would be the comparison of images. Several models to compare images permitting not only differences in the pixel values but also distortions have been proposed [15, 6]. When considering color images, a natural choice is that the cost to change one pixel by another has a cost related to the absolute difference of their colors. Once a suitable cost for insertions and deletions of pixels is chosen, the problem is how to compute the best alignment between two images or find the places in a large image where a small image pattern aligns best. The algorithms depicted in [15, 6] need $O(n^4)$ time to compare two $n \times n$ images. They also give fast filtration methods to search for patterns inside large images. In several cases, these algorithms resort to one-dimensional weighted edit distance or one-dimensional approximate searching algorithms. These could be significantly improved if the images were run-length compressed prior to the computation and our algorithms were used for those subproblems. Some recent algorithms searching for rotated image patterns inside a large image [12] could be extended as well: their matching model does not permit insertions or deletions of pixels, so they could be integrated with other approaches such as [6]. Again, it would be possible to speed up the comparison process by run-length compressing the image and the pattern, the latter at several rotations.

With respect to the original models, an interesting question is whether an algorithm can be obtained whose cost is just the product of the compressed lengths. Indeed, this seems possible in the average case, as demonstrated by the experiments with our improved algorithm for the LCS. Finally, a combination of a two-dimensional approximate pattern matching algorithm with two-dimensional run-length compression [15, 6, 1, 3] seems interesting.

References

[1] A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proc. 2nd IEEE Data Compression Conference (DCC'92)*, pages 279–288, 1992.

[2] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and Systems Sciences*, 52(2):299–307, 1996.

[3] A. Amir, G. Landau, and D. Sokol. Inplace run-length 2d compressed search. *Proc. 11th Symposium on Discrete Algorithms (SODA'00)*, pages 817–818, 2000.

[4] A. Apostolico, G. Landau, and S. Skiena. Matching for run-length encoded strings. *Journal of Complexity*, 15:4–16, 1999.

[5] O. Arbell, G. Landau, and J. Mitchell. Edit distance of run-length encoded strings. *Information Processing Letters*, 83(6):307–314, 2002.

[6] R. Baeza-Yates and G. Navarro. New models and algorithms for multidimensional approximate pattern matching. *Journal of Discrete Algorithms* 1(1):21–49, 2000. Special issue on Matching Patterns. Hermes Science Publishing.

[7] H. Bunke and J. Csirik. An algorithm for matching run-length coded strings. *Computing*, 50:297–314, 1993.

[8] H. Bunke and J. Csirik. An improved algorithm for computing the edit distance of run-length coded strings. *Information Processing Letters*, 54(2):93–96, 1995.

[9] M. Crochemore, G. M. Landau, and M. Ziv-Ukelson. A Sub-quadratic Sequence Alignment Algorithm for Unrestricted Cost Matrices. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*, pages 679–688, 2002 (the citation is to the revised report 2001-08 at Institut Gaspard-Monge, Université de Marne-la-Vallée).

[10] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed texts. *Algorithmica*, 20:388–404, 1998.

[11] W. Feller. *An Introduction to Probability Theory and Its Applications.* Vol. II, John Wiley, New York, 1966.

[12] K. Fredriksson. Rotation invariant histogram filters for similarity and distance measures between digital images. In *Proc. 7th Symposium on String Processing and Information Retrieval (SPIRE'00)*, IEEE CS Press, pages 105–115.

[13] H. Gajewska and R. Tarjan. Deques with heap order. *Information Processing Letters* 12(4):197–200, 1986.

[14] J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching over Ziv-Lempel compressed text. In *Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM'00)*, LNCS 1848, pages 195–209, 2000. Extended version to appear in the *Journal of Discrete Algorithms.*

[15] K. Krithivasan and R. Sitalakshmi. Efficient two-dimensional pattern matching in the presence of errors. *Information Sciences* 43:169–184, 1987.

[16] T. Kida, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. A unifying framework for compressed pattern matching. In *Proc. 6th Symposium on String Processing and Information Retrieval (SPIRE'99)*, pages 89–96. IEEE CS Press, 1999.

[17] T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. In *Proc. 8th IEEE Data Compression Conference (DCC'98)*, 1998.

[18] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady* 6:707–710, 1966.

[19] T. Matsumoto, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Bit-parallel approach to approximate string matching. In *Proc. 7th Symposium on String Processing and Information Retrieval (SPIRE'00)*, IEEE CS Press, pages 221–228, 2000.

[20] J. Mitchell. A geometric shortest path problem, with application to computing a longest common subsequence in run-length encoded strings. In *Technical Report*, Dept. of Applied Mathematics, SUNY Stony Brook, 1997.

[21] V. Mäkinen, G. Navarro, and E. Ukkonen. Approximate matching of run-length compressed strings. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM'01)*, LNCS 2089, pages 31–49, 2001.

[22] G. Navarro, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Faster Approximate string matching over compressed text. In *Proc. 11th IEEE Data Compression Conference (DCC'01)*, pages 459–468, 2001.

[23] P. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms*, 1(4):359–373, 1980.

[24] E. Ukkonen. Algorithms for approximate string matching. *Information and Control* 64(1–3):100–118, 1985.

[25] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms* 6(1–3):132–137, 1985.

[26] R. Wagner and M. Fisher. The string-to-string correction problem. *Journal of the ACM* 21(1):168–173, 1974.

[27] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.