Approximate Matching of Run-Length Compressed Strings

Veli Mäkinen[1]*, Gonzalo Navarro[2]**, and Esko Ukkonen[1]*

[1] Department of Computer Science, P.O Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, Finland.
{vmakinen,ukkonen}@cs.helsinki.fi
[2] Department of Computer Science, University of Chile, Blanco Encalada 2120,
Santiago, Chile. gnavarro@dcc.uchile.cl

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. This approach gives also 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, both for LCS and Levenshtein models. 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.

1 Introduction

The problem of *compressed pattern matching* is, given a compressed text T and a (possibly compressed) pattern P, 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 exactly pattern P. We can loosen the requirement of exact occurrences to *approximate occurrences* by introducing a distance function to measure the similarity between P and a substring of 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*, where the minimum amount of character insertions, deletions, and replacements, that are needed to make the two strings equal, is calculated. 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

* Supported by the Academy of Finland under grant 22584.
** Supported in part by Fondecyt grants 1-990627 and 1-000929.

Benson [1], e.g. [2, 8, 10, 9]. The only works addressing the approximate variant of the problem have been [11, 13, 15], on Ziv-Lempel [20].

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 detect distortions. Even a one-dimensional compressed approximate matching algorithm would be useful to speed up existing two-dimensional approximate matching algorithms, e.g. [5].

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 [6]. They considered the version of edit distance without the replacement operation, that 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 [7]. A major improvement over the previous results was due to Apostolico, Landau, and Skiena [3]; they first gave a basic $O(m'n'(m' + n'))$ algorithm, and further improved it to $O(m'n' \log(m'n'))$. Mitchell [14] 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 to the amount of equal letter boxes, see the definition in Sect. 2.2). All these algorithms were limited to the LCS distance, although, Mitchell's method [14] could be applied when different costs are assigned to the insertion and deletion operations. It still remain 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 matching run-length encoded strings under *Levenshtein* distance [12]. 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 [7]; 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 [4].

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'. This algorithm works for both Levenshtein and LCS distance models.

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 [17], 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, that do not however affect the worst case, but do have 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.

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 $1 \leq i_1 < i_2 \cdots < i_k \leq m$.

The *edit distance* 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 [12, 19, 16]. 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. For *Levenshtein distance* (denoted by $D_L(A, B)$) [12], we assign costs $w(a \to a) = 0$, $w(a \to b) = 1$, $w(a \to \lambda) = 1$, and $w(\lambda \to a) = 1$, for all $a, b \in \Sigma$, $a \neq b$. If substitutions are forbidden, i.e. $w(a \to b) = \infty$, we get the distance $D_{ID}(A, B)$.

Distance $D_L(A, B)$ can be calculated by using dynamic programming [16]; evaluate an $(m+1) \times (n+1)$ matrix (d_{ij}), $0 \leq i \leq m$, $0 \leq j \leq n$, using the recurrence

$$\begin{aligned} d_{i,0} &= i, \qquad 0 \leq i \leq m, \\ d_{0,j} &= j, \qquad 0 \leq j \leq n, \\ d_{i,j} &= min(\textbf{if } a_i = b_j \textbf{ then } d_{i-1,j-1} \textbf{ else } d_{i-1,j-1} + 1, \\ &\qquad d_{i-1,j} + 1, d_{i,j-1} + 1), \qquad \text{otherwise.} \end{aligned} \tag{1}$$

The matrix (d_{ij}) can be evaluated row-by-row or column-by-column in $O(mn)$ time, and the value d_{mn} equals $D_L(A, B)$.

A similar method can be used to calculate the distance $D_{ID}(A, B)$. Now, the recurrence is

$$\begin{aligned} d_{i,0} &= i, \qquad 0 \leq i \leq m, \\ d_{0,j} &= j, \qquad 0 \leq j \leq n, \\ d_{i,j} &= min(\textbf{if } a_i = b_j \textbf{ then } d_{i-1,j-1} \textbf{ else } \infty, \\ &\qquad d_{i-1,j} + 1, d_{i,j-1} + 1), \qquad \text{otherwise.} \end{aligned} \tag{2}$$

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 $2 * |LCS(A,B)| = m + n - D_{ID}(A,B)$.

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 the strings A and B are run-length encoded. In both methods, we use the following notation to divide the matrix (d_{ij}) into submatrices (see Fig. 1).

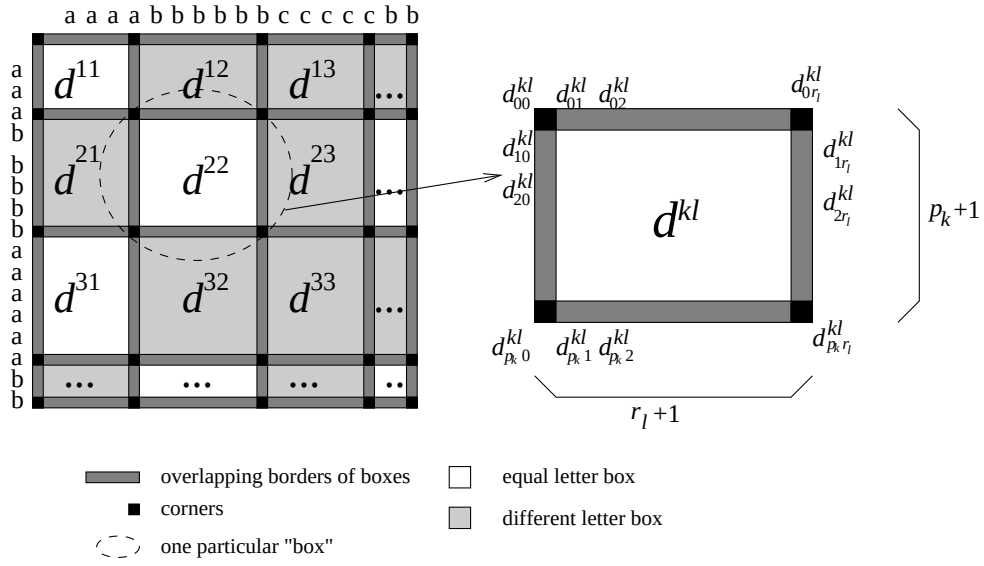

Fig. 1. A dynamic programmig matrix split into run-length blocks.

Let $A' = (a_{i_1}, p_1)(a_{i_2}, p_2) \ldots (a_{i_{m'}}, p_{m'})$ and $B' = (b_{j_1}, r_1), (b_{j_2}, r_2) \ldots (b_{j_{n'}}, p_{n'})$ 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 separate the edit distance matrix (d_{ij}) into submatrices. To ease the notation later on, we define the submatrices so that they overlap on the

borders. Formally, each pair of runs $(a_{i_k}, p_k), (b_{j_\ell}, r_\ell)$ defines a $(p_k+1) \times (r_\ell+1)$ submatrix $(d^{k,\ell}_{s,t})$ such that

$$d^{k,\ell}_{s,t} = d_{i_k+s-1, j_\ell+t-1}, \qquad 0 \le s \le p_k, 0 \le t \le r_\ell. \tag{3}$$

We will call submatrices $(d^{k,\ell}_{s,t})$ *boxes*. If a pair of runs corresponding to a box contain equal letters (i.e. $a_{i_k} = b_{j_\ell}$), then $(d^{k,\ell}_{s,t})$ 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.

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

Bunke and Csirik [7] 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 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 [4] 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 Fig. 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 The 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 [7]) *The recurrences (1) and (2) can be replaced by*

$$d^{k,\ell}_{s,t} = \textbf{if } s \le t \textbf{ then } d^{k,\ell}_{0,t-s} \textbf{ else } d^{k,\ell}_{s-t,0}, \tag{4}$$

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 (1) and (2) 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 (1) can be replaced by*

$$d^{k,\ell}_{s,t} = 1 + min\ (\ t-1 + min_{max(0,s-t)\le q\le s}\ d^{k,\ell}_{q,0}\ ,\\ s-1 + min_{max(0,t-s)\le q\le t}\ d^{k,\ell}_{0,q}), \tag{5}$$

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 $s+t=2$ the formula (5) 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 (1). 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 (1), and using the induction hypothesis we get

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

where we have used the property that consecutive cells in the (d_{ij}) matrix differ at most by 1 [18]. 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

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

which is the particularization of formula (5) for $s=1$. □

Formula (5) 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.

Fig. 2 shows the algorithm. We use a data structure (which in the pseudocode 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 $min_{max(0,s-t)\le q\le s}\ d^{k,\ell}_{q,0}$ and $min_{max(0,t-s)\le q\le t}\ d^{k,\ell}_{0,q}$, used in the formula (5). We see later that in our particular application all those operations can be performed in constant time.

In the code we use $dr^{k,\ell}_s = d^{k,\ell}_{s,r_\ell}$ for the rightmost column and $db^{k,\ell}_t = d^{k,\ell}_{p_k,t}$ for the bottom row. Their update formulas are derived from the formula (5):

$$\begin{aligned} dr^{k,\ell}_s &= 1 + min(\ r_\ell - 1 + min_{max(0,s-r_\ell)\le q\le s}\ dr^{k,\ell-1}_q\ , \\ &\qquad\qquad s - 1 + min_{max(0,r_\ell-s)\le q\le r_\ell}\ db^{k-1,\ell}_q), \\ db^{k,\ell}_t &= 1 + min(\ t - 1 + min_{max(0,p_k-t)\le q\le p_k}\ dr^{k,\ell-1}_q\ , \\ &\qquad\qquad p_k - 1 + min_{max(0,t-p_k)\le q\le t}\ db^{k-1,\ell}_q). \end{aligned}$$

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 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.*

□

3.2 The 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 [18]. Our data structure represents the multiset S as a triple $(min(S), max(S), V_{min(S)\ldots max(S)} \to \mathbb{N})$. 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 $min(S) \le i \le max(S)$ 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 now suppose that $V_i = 0$ for an intermediate value. 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 between $min(S)$ and $max(S)$ exists in a path that goes from $d_{i,j}$ to $d_{i',j'}$. □

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

1. `/* We fill the topmost row and leftmost column first */`
2. $dr_0^{0,0} \leftarrow 0, \quad db_0^{0,0} \leftarrow 0$
3. **For** $k \in 1\ldots m'$ **Do**
4. **For** $s \in 0\ldots p_k$ **Do** $dr_s^{k,0} \leftarrow dr_{p_{k-1}}^{k-1,0} + s$
5. $db_0^{k,0} \leftarrow dr_{p_k}^{k,0}$
6. **For** $\ell \in 0\ldots n'$ **Do**
7. **For** $t \in 0\ldots r_\ell$ **Do** $db_t^{0,\ell} \leftarrow db_{r_{\ell-1}}^{0,\ell-1} + t$
8. $dr_0^{0,\ell} \leftarrow db_{r_\ell}^{0,\ell}$
9. `/* Now we fill the rest of the matrix */`
10. **For** $\ell \in 1\ldots m'$ **Do** `/* column-wise traversal */`
11. **For** $k \in 1\ldots n'$ **Do**
12. **If** $a_k = b_\ell$ **Then** `/* equal letter box */`
13. **For** $s \in 1\ldots p_k$ **Do**
14. **If** $s \leq r_\ell$ **Then** $dr_s^{k,\ell} \leftarrow db_{r_\ell - s}^{k-1,\ell}$ **Else** $dr_s^{k,\ell} \leftarrow dr_{s-r_\ell}^{k,\ell-1}$
15. **For** $t \in 1\ldots r_\ell$ **Do**
16. **If** $p_k \leq t$ **Then** $db_t^{k,\ell} \leftarrow db_{t-p_k}^{k-1,\ell}$ **Else** $db_t^{k,\ell} \leftarrow dr_{p_k-t}^{k,\ell-1}$
17. **Else** `/* different letter box */`
18. $M_r \leftarrow \{dr_0^{k,\ell-1}\}, \quad M_b \leftarrow \{db_{r_\ell}^{k-1,\ell}\}$
19. $dr_0^{k,\ell} \leftarrow dr_{p_{k-1}}^{k-1,\ell}$
20. **For** $s \in 1\ldots p_k$ **Do**
21. $M_r \leftarrow M_r \cup \{dr_s^{k,\ell-1}\}$
22. **If** $s > r_\ell$ **Then** $M_r \leftarrow M_r - \{dr_{s-r_\ell-1}^{k,\ell-1}\}$
23. **If** $r_\ell \geq s$ **Then** $M_b \leftarrow M_b \cup \{db_{r_\ell - s}^{k-1,\ell}\}$
24. $dr_s^{k,\ell} \leftarrow 1 + min(r_\ell - 1 + min(M_r), s - 1 + min(M_b))$
25. $M_r \leftarrow \{dr_{p_k}^{k,\ell-1}\}, \quad M_b \leftarrow \{db_0^{k-1,\ell}\}$
26. $db_0^{k,\ell} \leftarrow db_{r_{\ell-1}}^{k,\ell-1}$
27. **For** $t \in 1\ldots r_\ell$ **Do**
28. **If** $p_k \geq t$ **Then** $M_r \leftarrow M_r \cup \{dr_{p_k-t}^{k,\ell-1}\}$
29. $M_b \leftarrow M_b \cup \{db_t^{k-1,\ell}\}$
30. **If** $t > p_k$ **Then** $M_b \leftarrow M_b - \{db_{t-p_k-1}^{k-1,\ell}\}$
31. $db_t^{k,\ell} \leftarrow 1 + min(t - 1 + min(M_r), p_k - 1 + min(M_b))$
32. **Return** $dr_{p_{m'}}^{m'n'}$ `/* or` $db_{r_{n'}}^{m'n'}$ `*/`

Fig. 2. The $O(m'n + n'm)$ time algorithm to compute the Levenshtein distance between A and B, coded as a run-length sequence of pairs $(letter, run_length)$.

Fig. 3 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 extension is never 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, the operation $min(S)$ is trivial as we have it already precomputed.

```
Create (x)
 1.     Return (x, x, V_x = 1)

Add ((minS, maxS, V), y)
 2.     If y < minS Then
 3.         minS ← y
 4.         add new first cell V_y = 0
 5.     Else If y > maxS Then
 6.         maxS ← y
 7.         add new last cell V_y = 0
 8.     V_y ← V_y + 1
 9.     Return (minS, maxS, V)

Remove ((minS, maxS, V), z)
 10.    V_z ← V_z − 1
 11.    If V_z = 0 Then
 12.        If z = minS Then
 13.            remove first cell from V
 14.            minS ← minS + 1
 15.        Else /* z = maxS */
 16.            remove last cell from V
 17.            maxS ← maxS − 1
 18.    Return (minS, maxS, V)

Min ((minS, maxS, V))
 19.    Return minS
```

Fig. 3. The multiset data structure implementation.

It is easily seen 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_r) or r_ℓ (for M_b), which are known at the time of **Create**.

4 Approximate Searching

Let us now consider a problem related to computing the LCS or the Levenshtein 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 [16] computes a matrix exactly like those of recurrences (2) and (1), 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_{mj} 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 $2m$ columns only (at $O(m^2)$ cost). The rest of the columns of the run are equal to the $2m$-th because no optimal path can be longer than $2m - 1$ under the LCS or Levenshtein models. We later examine the last row of the matrix and report every text position with value $\leq k$. If the run is longer than $2m$, then we have not produced the whole last row but only the first $2m$ cells of it. In this case we report the positions $2m + 1 \ldots r_\ell$ of the ℓ-th run if and only if the position $2m$ 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 $2m$) and *long* (longer than $2m$) runs. We apply our Levenshtein algorithm on the text runs, filling the matrix column-wise. If we have a short run (a_{i_ℓ}, r_ℓ), $r_\ell \leq 2m$, 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 $i_\ell + t$ such that $d^{m',\ell}_{p_{m'},t} \leq k$.

If we have a long run (a_{i_ℓ}, r_ℓ), $r_\ell > 2m$, we limit its length to $2m$ and apply the same algorithm, at $O(m'm + m + m)$ cost. The columns $2m + 1 \ldots r_\ell$ of that run are equal to the $2m$-th, so we just need to examine the last row of the $2m$-th column, and report all the text positions up to the end of the run, $i_\ell + 2m + 1 \ldots i_\ell + p_k$, if $d^{m',\ell}_{p_{m'},2m} \leq k$.

This algorithm is $O(n'm'm + R)$ time in the worst case, where R is the number of occurrences reported. For the LCS model we have the same upper

bound, so we achieve the same complexity. Our $O(m'n'(m'+n'))$ algorithm does not yield a good complexity here. The space is that to compute one text run limited to length $2m$, i.e. $O(m'm)$.

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 R extra term of the search cost disappears.

Theorem 5 *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.*

□

5 Improving a Greedy Algorithm for the LCS

The idea in our algorithm for the Levenshtein distance D_L in Sect. 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 Fig. 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 [17] can be applied, and achieve an $O(d^2 min(n'm'))$ algorithm. We give also 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 in the average. As we are unable to prove this conjecture, we provide experimental evidence to support it.

5.1 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:

[1] Apostolico et. al. [3] 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 calculating the LCS directly, and we are calculating the distance D_{ID}. Also, they traced a specific optimal path (which was the property that they could use to achieve the $O(m'n'\log(m'n'))$ algorithm).

Lemma 6 (Bunke and Csirik [7]) *The recurrence (2) can be replaced by the recurrence*

$$d_{s,t}^{k,\ell} = min(d_{s,0}^{k,\ell} + t, d_{0,t}^{k,\ell} + s), \tag{6}$$

where $1 \leq s \leq p_k$ *and* $1 \leq t \leq r_\ell$, *for values* $d_{s,t}^{k,\ell}$ *in a different letter box.* □

In contrast to the D_L distance, the difficult part in D_{ID} distance lies in equal letter boxes. As noted earlier, Lemma 1 applies also 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 $d_{p_k,r_\ell}^{k,\ell}$ can be calculated in constant time. This gives an $O(m'n')$ algorithm for a special case, as previously noted in [6].

What follows is an algorithm to retrieve the value $d_{p_k,r_\ell}^{k,\ell}$ in an equal letter box in $O(m' + n')$ time. The idea is to trace an optimal path to the cell $d_{p_k,r_\ell}^{k,\ell}$. This can be done by using lemmas 1 and 6 recursively. Assume that $d_{p_k,r_\ell}^{k,\ell} = d_{0,r_\ell-p_k}^{k,\ell}$ by Lemma 1 (case $d_{p_k,r_\ell}^{k,\ell} = d_{p_k-r_\ell,0}^{k,\ell}$ is symmetric). If $k = 1$, then the value $d_{0,r_\ell-p_k}^{1,\ell}$ corresponds to a value in the first row (0) of the matrix (d_{ij}) which is known. Otherwise, the box $(d_{s,t}^{k-1,\ell})$ is a different letter box, and using the definition of overlapping boxes and Lemma 6 it holds

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

Now, the value $d_{p_{k-1},0}^{k-1,\ell}$ is calculated during the dynamic programming, so we can continue on tracing value $d_{0,r_\ell-p_k}^{k-1,\ell}$ using lemmas 1 and 6 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 6) 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 $d_{p_k,r_\ell}^{k,\ell}$ 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 $d_{p_k,r_\ell}^{k,\ell}$ 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 7 *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.* □

5.2 Diagonal Algorithm

The diagonal method [17] 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, respectivily. The idea is the following: The value $d_{mn} = D_{ID}(A,B)$ in the (d_{ij}) matrix of (2) 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 (2) on the increasing diagonal band. As soon as $d_{mn} < k$, we have found $D_{ID}(A,B) = d_{mn}$, 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 $d = D_{ID}(A,B)$.

5.3 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.

First of all, the runs of different letter boxes can be skipped in the tracing paths.

Consider two consecutive different letter boxes $(d^{k,\ell}_{s,t})$ and $(d^{k+1,\ell}_{s,t})$. By Lemma 6 it holds for the values $1 \le t \le r_\ell$,

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

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

Lemma 8 *Let* $((d_{s,t}^{k',\ell}), (d_{s,t}^{k'+1,\ell}), \ldots, (d_{s,t}^{k,\ell}))$ *and* $((d_{s,t}^{k,\ell'}), (d_{s,t}^{k,\ell'+1}), \ldots, (d_{s,t}^{k,\ell}))$ *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*

$$d_{p_k,t}^{k,\ell} = min\left(d_{p_k,0}^{k,\ell} + t, d_{0,t}^{k',\ell} + \sum_{s=k'}^{k} p_s\right) \qquad 1 \leq t \leq r_\ell,$$

$$d_{s,r_\ell}^{k,\ell} = min\left(d_{0,r_\ell}^{k,\ell} + s, d_{s,0}^{k,\ell'} + \sum_{t=\ell'}^{\ell} r_t\right) \qquad 1 \leq s \leq p_k.$$

□

Now it is obvious how to speed up the retrieval of values $d_{p_k,r_\ell}^{k,\ell}$ 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 $d_{p_k,r_\ell}^{k,\ell}$ in an equal letter box, we can use Lemma 8 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 8 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 9 *Given strings* A *and* B *of lengths* m *and* n *that are run-length encoded to lengths* m' *and* n'*, such that all the runs of different letters over an alphabet of size* $|\Sigma|$ *are equally likely and in random order, there is an algorithm to calculate* $D_{ID}(A, B)$ *in* $O(m'n'(1 + (m' + n')/|\Sigma|^2))$ *time in the average.*

Proof. (Sketch) 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. Between two runs of a letter $\sigma \in \Sigma$, there are on the average $|\Sigma| - 1$ runs of other letters. This holds both for strings A and B. In other words, the expected length of a run of different letter boxes is $|\Sigma| - 1$. Therefore the retrieval of the value $d_{p_k,r_\ell}^{k,\ell}$ in an equal letter box takes time at most $O((m' + n')/|\Sigma|)$ in the average. □

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 $d_{p_k,t}^{k,\ell}$, $1 \leq t \leq r_\ell$ (or symmetrically $d_{s,r_\ell}^{k,\ell}$, $1 \leq s \leq p_k$) in the border of a different letter box. If $d_{p_k,r_\ell}^{k,\ell} = d_{p_k,0}^{k,\ell} + r_\ell$ then it must hold $d_{p_k,t}^{k,\ell} = d_{p_k,0}^{k,\ell} + t$, otherwise we get a contradiction: $d_{p_k,r_\ell}^{k,\ell} < d_{p_k,0}^{k,\ell} + r_\ell$.

We call the above situation a horizontal (vertical) *bridge*. Note that from Lemma 6 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 10 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 Fig. 4.

Lemma 10 *Let* $((d_{s,t}^{k',\ell}), (d_{s,t}^{k'+1,\ell}), \ldots, (d_{s,t}^{k,\ell}))$ *be a vertical run of different letter boxes. If there is a horizontal bridge* $d_{p_{k'},r_\ell}^{k',\ell} = d_{p_{k'},0}^{k',\ell} + r_\ell$ *then there is a horizontal bridge* $d_{p_{k''},r_\ell}^{k'',\ell} = d_{p_{k''},0}^{k'',\ell} + r_\ell$ *for all* $k' < k'' \leq k$. *The symmetric result holds for horizontal runs of different letter boxes.*

Proof. We use the counter-argument that $d_{p_{k''},r_\ell}^{k'',\ell} = d_{p_{k''},0}^{k'',\ell} + r_\ell$ does not hold for some $k' < k'' \leq k$. Then by Lemma 8 and by the bridge assumption it holds

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

On the other hand, using the counter-argument and the fact that consecutive cells in the (d_{ij}) matrix differ at most by 1 [18], we get

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

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

Lemma 10 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.

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 (2), and this makes the reasoning with probabilities much more complex. We are still confident that the following conjecture holds, although we are not (yet) able to prove it.

```
LCS (A' = (a_{i_1}, p_1)(a_{i_2}, p_2) ... (a_{i_{m'}}, p_{m'}), B' = (b_{j_1}, r_1)(b_{j_2}, r_2) ... (b_{j_{n'}}, r_{n'}))
1.        /* We use structure d^{k,ℓ} to denote a box (d^{k,ℓ}_{s,t}) as follows: */
2.    d^{k,ℓ}.corner := d^{k,ℓ}_{p_k,r_ℓ}
3.    d^{k,ℓ}.jumptop := "location of the next equal letter box above"
4.    d^{k,ℓ}.jumpleft := "location of the next equal letter box in the left"
5.    d^{k,ℓ}.sumtop := If a_{i_k} ≠ b_{j_ell} Then ∑_{t=d^{k,ℓ}.jumptop+1}^{k} p_t
6.    d^{k,ℓ}.sumleft := If a_{i_k} ≠ b_{j_ℓ} Then ∑_{t=d^{k,ℓ}.jumpleft+1}^{ℓ} r_t
7.        /* Initialize first row and column (let a_{i_0} = b_{j_0} = ε, p_0 = r_0 = 1) */
8.    d^{00}.corner ← 0
9.    For k ∈ 1...n' Do d^{k,0}.corner ← d^{k-1,0}.corner + r_{k-1}
10.   For ℓ ∈ 1...m' Do d^{0,ℓ}.corner ← d^{0,ℓ-1}.corner + p_{ℓ-1}
11.   Calculate values d^{k,ℓ}.(jumptop, jumpleft, sumtop, sumleft)
12.       /* Now we fill the rest of the corner values */
13.   For k ∈ 1...m' Do
14.       For ℓ ∈ 1...n' Do
15.           (bridge, k', ℓ', p, r, sum, d^{k,ℓ}.corner) ← (false, k, ℓ, p_k, r_ℓ, 0, ∞)
16.           If a_{i_k} ≠ b_{j_ℓ} Then /* Different letter box */
17.               d^{k,ℓ}.corner ← min(d^{k-1,ℓ}.corner + a_{i_k}, d^{k,ℓ-1}.corner + b_{j_ℓ})
18.           Else While bridge = false Do
19.               /* Equal letter box, trace d^{k,ℓ}.corner */
20.               If p = r Then /* Straight from the diagonal */
21.                   d^{k,ℓ}.corner ← min(d^{k,ℓ}.corner, sum + d^{k'-1,ℓ'-1}.corner)
22.                   bridge ← true
23.               Else If p < r Then /* Diagonal up */
24.                   (r, k') ← (r - p, k' - 1)
25.                   d^{k,ℓ}.corner ← min(d^{k,ℓ}.corner, sum + d^{k',ℓ'-1}.corner + r)
26.                   If d^{k',ℓ'}.corner = d^{k',ℓ'-1}.corner + r_{ℓ'} Then bridge ← true
27.                   Else /* Jump to the next equal letter box */
28.                       (sum, k') ← (sum + d^{k',ℓ'}.sumtop, d^{k',ℓ'}.jumptop)
29.                       p ← p_{k'}
30.                       If k' = 0 Then /* First row */
31.                           d^{k,ℓ}.corner ← min(d^{k,ℓ}.corner,
                                                   sum + d^{k',ℓ'-1}.corner + r)
32.                           bridge ← true
33.               Else /* Diagonal left similarly*/
34.   Return (m + n - d^{m',n'}.corner)/2 /* return the length of the LCS */
```

Fig. 4. The improved greedy algorithm to compute the LCS between A and B, coded as a run-length sequence of pairs $(letter, run_length)$.

Conjecture 11 *Let A and B be strings that are run-length encoded to lengths m' and n', such that the runs are equally distributed with the same mean in both strings. Under these assumptions the expected running time of the algorithm in Fig. 4 for calculating $D_{ID}(A, B)$ is $O(m'n')$.*

5.4 Experimental Results

To test the Conjecture 11, we ran the algorithm in Fig. 4 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 (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.).

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 10-10000 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)

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 also L 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 also pairs that were close to the worst case (close to $A = a^n, B = (ab)^{n/2}$), and therefore the results were slightly worse than with randomly generated data: $L \in \{2.00, 2.34, 2.31\}$ with the three images.

6 Conclusions

We have presented new algorithms to compute approximate matches between run-length compressed strings. The previous algorithms [7, 3] permit computing their LCS. We have extended an LCS algorithm [7] to the Levenshtein distance without increasing the cost, and presented an algorithm with nontrivial complexity for approximate searching a run-length compressed pattern on a run-length compressed text under either model.

Future work involves adapting our algorithm to more complex versions of the Levenshtein distance, including at least different costs for the edit operations. This would be interesting for applications related to image compression, where the change from a pixel value to the next is smooth.

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 two-dimensional approximate pattern matching algorithm with two-dimensional run-length compression [5, 1] seems extremely interesting.

References

1. A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proc. DCC'92*, pages 279–288, 1992.
2. A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *J. of Comp. and Sys. Sciences*, 52(2):299–307, 1996.
3. A. Apostolico, G. Landau, and S. Skiena. Matching for run-length encoded strings. *J. of Complexity*, 15:4–16, 1999. (Also at Sequences '97, Positano Italy, June 11–13, 1997).
4. O. Arbell, G. Landau, and J. Mitchell. Edit distance of run-length encoded strings. Submitted for publication, August 2000.
5. R. Baeza-Yates and G. Navarro. Fast two-dimensional approximate pattern matching. In *Proc. LATIN'98*, LNCS 1380, pages 341–351, 1998.
6. H. Bunke and J. Csirik. An algorithm for matching run-length coded strings. *Computing*, 50:297–314, 1993.
7. 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.
8. M. Farach and M. Thorup. String matching in Lempel-Ziv compressed texts. *Algorithmica*, 20:388–404, 1998.

9. T. Kida, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. A unifying framework for compressed pattern matching. In *Proc. SPIRE'99*, pages 89–96. IEEE CS Press, 1999.
10. T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. In *Proc. DCC'98*, pages 103–112, 1998.
11. J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching over Ziv-Lempel compressed text. In *Proc. CPM'2000*, LNCS 1848, pages 195–209, 2000.
12. V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady* 6:707–710, 1966.
13. T. Matsumoto, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Bit-parallel approach to approximate string matching. In *Proc. SPIRE'2000*, IEEE CS Press, pages 221–228, 2000.
14. 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.
15. 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)*, 2001, To appear.
16. P. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *J. of Algorithms*, 1(4):359–373, 1980.
17. E. Ukkonen. Algorithms for approximate string matching. *Information and Control* 64(1–3):100–118, 1985.
18. E. Ukkonen. Finding approximate patterns in strings. *J. of Algorithms* 6(1–3):132–137, 1985.
19. R. Wagner and M. Fisher. The string-to-string correction problem. *J. of the ACM* 21(1):168–173, 1974.
20. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23:337–343, 1977.