A Fully Compressed Algorithm for Computing the Edit Distance of Run-Length Encoded Strings[⋆]

Kuan-Yu Chen[1] and Kun-Mao Chao[1,2,3]

[1] Department of Computer Science and Information Engineering
[2] Graduate Institute of Biomedical Electronics and Bioinformatics
[3] Graduate Institute of Networking and Multimedia
National Taiwan University, Taipei, Taiwan 106

Abstract. In this paper, a commonly used data compression scheme, called run-length encoding, is employed to speed up the computation of edit distance between two strings. Our algorithm is the first to achieve "fully compressed," meaning that it runs in time polynomial in the number of runs of both strings. Specifically, given two strings, compressed into m and n runs, $m \leq n$, we present an $O(mn^2)$-time algorithm for computing the edit distance of the two strings. Our approach also gives the first fully compressed algorithm for approximate matching of a pattern of m runs in a text of n runs in $O(mn^2)$ time.

1 Introduction

The edit distance (a.k.a Levenshtein distance) is a common similarity measure between two strings. It is defined as the minimum steps required to transform one string into the other via operations of insertions, deletions, or substitutions. The problem of computing the edit distance between two strings has been explored for decades. The classic dynamic programming solution takes $O(N^2)$ time, where N denotes the length of both strings. All known techniques for breaking the $O(N^2)$ time bound essentially follow the paradigm of *acceleration via compression* (see [8] for more details.). The first breakthrough to $O(N^2/\log N)$-time was made by Masek and Paterson [12], who used the Four-Russians technique to speed up the edit-distance computation. The Four-Russians technique can be seen as a naïve compression utilizing the fact that sufficiently short substrings over a constant alphabet must appear many times. After that, Crochemore *et al.* [6] exploited LZ-factorization of strings and gave an $O(hN^2/\log N)$-time algorithm, where $h \leq 1$ is the entropy of the text. Their solution is general enough for the sequence alignment problem with unrestricted scoring matrices.

In this paper, we accelerate the edit-distance computation by exploiting symbol repetitions in strings. The underlying compression scheme is called *run-length*

[⋆] Partially supported by NSC grants 97-2221-E-002-097-MY3 and 98-2221-E-002-081-MY3 from the National Science Council, Taiwan.

encoding (RLE), which is widely applied in many areas, e.g. FAX transmission, image compression, and optical character recognition. Previous results under this paradigm include the computation of indel-distance (dual of longest common subsequence) of two run-length encoded strings which requires $O(mn \log mn)$ time [2,14], where m and n denote the number of runs of input strings of lengths M and N. Several papers showed how to compute the edit distance of two run-length encoded strings in $O(Mn+mN)$ time [3,6,13]. Recently, Huang *et al.* [10] and Liu *et al.* [11] improved the bound to $O(\min\{Mn, mN\})$ time. (It should be noted that the works of [3,11] focused on the Levenshtein distance in which each edit operation has a unit cost, while the results of [6,10,13] apply to the general edit distance problem with weighted costs.) To date, all the time bounds for the edit distance problem still depend on the uncompressed string lengths. Therefore, an intriguing question is (as asked in [10,13]) whether one can design an algorithm that is "fully compressed," i.e. its time complexity depends solely on the number of runs of the RLE strings. For Levenshtein distance, this paper answers the question affirmatively by giving an $O(mn^2)$-time solution.

On the other hand, a closely related problem, called *compressed pattern matching* is, given a compressed text T and an uncompressed pattern P, to find all occurrences of P in T without decompressing T. If pattern P is also compressed, the problem is referred to as *fully compressed pattern matching.* Many studies have been made around this subject under different compression schemes, e.g. RLE compression, LZ-family compression, and straight-line programs. For RLE inputs, the exact string matching problem can be easily solved in $O(m+n)$ time, where m and n denote the number of runs of the pattern and the text. The k-mismatch with wildcards problem can be solved in $O(mn \log m)$ time [5]. The *approximate matching problem* seeks for occurrences of an RLE string within another RLE string, allowing up to k edit operations. For this problem, Mäkinen *et al.* [13] proposed an $O(mnM)$-time algorithm, where M denotes the uncompressed pattern length. Huang *et al.* later improved the bound to $O(nM)$ time [10]. The approach presented in this paper gives the first fully compressed algorithm running in $O(mn^2)$ time.

2 Preliminaries

2.1 Edit Graph

Given two strings $A[1 \ldots M]$ and $B[1 \ldots N]$, the edit distance problem can be solved as follows. Initially, $ED(i, 0) = i$ and $ED(0, j) = j$ for $0 \leq i \leq M$ and $1 \leq j \leq N$. For $1 \leq i \leq M$ and $1 \leq j \leq N$, $ED(i, j) = \min\{ED(i-1, j) + 1, ED(i, j-1) + 1, ED(i-1, j-1) + \delta(A[i], B[j])\}$, where $\delta(A[i], B[j]) = 0$ if $A[i]$ matches $B[j]$, and $\delta(A[i], B[j]) = 1$ otherwise. The edit distance between A and B is the value of $ED(M, N)$. This dynamic programming solution can be represented in terms of a weighted, acyclic grid graph G of $(M+1) \times (N+1)$ vertices, called *edit graph.* Each vertex (i, j) stores the value of $ED(i, j)$, and any shortest path in G from vertex $(0, 0)$ to vertex (M, N) specifies an optimal *edit trace* (a sequence of edit operations).

2.2 Propagation over Run-Sized Blocks

Run-length encoding (abbreviated as RLE) is a well-known coding scheme that performs lossless data compression. RLE compression simply groups consecutive, identical symbols of a string into a run, usually denoted by σ^i, where σ is an alphabet symbol and i is its repetition times. For example, string *bbcccddaaaaa* can be compressed into RLE format as $b^2c^3d^2a^5$. Based on the RLE factorization of input strings A and B, the edit graph G can be partitioned into mn blocks, where m and n denote the number of runs of A and B, respectively. We refer to a subgraph of G as a *match block* if it corresponds to a run of A and a run of B that encode the same symbol, and as a *mismatch block* otherwise. Note that two adjacent blocks in G share the vertices in their adjoining borders. As observed in [3,6,13], instead of computing all the vertex values of G, it suffices to compute only the vertex values in the block borders. That is, given a block H of G, we refer to the left and top borders of H as its *input border*, denoted by I, and refer to the bottom and right borders of H as its *output border*, denoted by O. Our algorithm follows the framework of [3,6,13], traversing the blocks of G in a left-to-right, top-to-bottom order, and propagates the accumulated values from I to O without filling in the vertices lying in-between.

2.3 Problem Reduction

In this subsection, we briefly review the observations made in previous work [4,3,13]. Given two vertices (i',j') and (i,j) of G, where $i' \le i$ and $j' \le j$, we let $dist(i',j',i,j)$ denote the total weight of the shortest path from (i',j') to (i,j). Moreover, we say that vertex (i,j) of G is in *diagonal* $j-i$ of G. Let vertex (i_0,j_0) denote the upper-left corner of block H. For each vertex (i,j) in O, we have the following recurrence relation:

$$ED(i,j) = \min \left\{ \begin{array}{l} \min_{i_0 \le i' \le i}(ED(i',j_0) + dist(i',j_0,i,j)) \\ \min_{j_0 \le j' \le j}(ED(i_0,j') + dist(i_0,j',i,j)) \end{array} \right\} \quad (1)$$

Lemma 1 ([4]). *If H is a match block, for each vertex (i,j) in O, we have that $ED(i,j) = ED(i_d,j_d)$, where (i_d,j_d) is the intersection of diagonal $j-i$ with I.*

By Lemma 1, the propagation over a match block is easy. To obtain the vertex values in the output border, we simply copy their diagonal values in the input border. Therefore, the difficulty lies in the propagation over a mismatch block, which can be reduced to the so-called *sliding-window minima problem* as follows.

Lemma 2 ([3,13]). *If H is a mismatch block, for vertex (i,j) in O, we have that $ED(i,j) = \min(\min_{i_d \le i' \le i}(ED(i',j_0) + j - j_0), \min_{j_d \le j' \le j}(ED(i_0,j') + i - i_0))$, where (i_d,j_d) is the intersection of diagonal $j-i$ with I.*

Let $LEFT[1\ldots h]$ and $TOP[1\ldots w]$ denote the vertex values of the left border and the top border of I. The entries of $LEFT$ and TOP are numbered in a bottom-to-top and left-to-right direction, respectively. Let $OUT[1\ldots w+h-1]$

denote the vertex values of O. The entries of OUT are numbered in a counter-clockwise direction, starting from the lower-left corner. For simplicity, we only discuss the case that block H is flat, i.e. $h \leq w$. The other case where $h > w$ can be argued symmetrically.

Definition 1. *For $i \in [1, w+h-1]$, we let $OUT_{left}[i]$ (resp., $OUT_{top}[i]$) denote the weight of the shortest path passing through the left border (resp., top border) of I and ending at the vertex corresponding to $OUT[i]$.*

By Equation (1), we can write:

$$OUT[i] = \min\{OUT_{left}[i], OUT_{top}[i]\} \text{ for } i \in [1, w+h-1]. \tag{2}$$

Problem 1. Given a numerical array $S[1 \ldots \ell]$ and a positive integer h, denoting the window size, we define the sliding-window minima array of S as $S^{(h)}[i] = \min\{S[j] \mid i-h+1 \leq j \leq i \text{ and } 1 \leq j \leq \ell\}$ for $i \in [1, \ell+h-1]$. The sliding-window minima problem (abbreviated as the SWM problem) is, given array S, to compute array $S^{(h)}$.

Given input array S, the SWM problem can be solved in $O(|S|)$ time using *deques with heap orders* described in [7]. By Lemma 2, we can derive the following equations, which are expressed in terms of the sliding-window minima arrays of $LEFT$ and TOP.

$$OUT_{left}[i] = \begin{cases} LEFT^{(h)}[i] + i - 1, & \text{for } i \in [1, h]; \\ LEFT^{(h)}[h] + i - 1, & \text{for } i \in [h, w]; \\ LEFT^{(h)}[i-w+h] + w - 1, & \text{for } i \in [w, w+h-1]; \end{cases} \tag{3}$$

$$OUT_{top}[i] = \begin{cases} TOP^{(h)}[i] + h - 1, & \text{for } i \in [1, w]; \\ TOP^{(h)}[i] + w + h - 1 - i, & \text{for } i \in [w, w+h-1]; \end{cases} \tag{4}$$

According to Equations (2)–(4), once arrays $LEFT^{(h)}$ and $TOP^{(h)}$ are obtained, the values of array OUT are easily computed. Therefore, the propagation over a mismatch block is reduced to two instances of the SWM problem.

3 A Geometric View

In this section, we show that the propagation described in the previous section can be further accelerated by computing only a subset of the border values.

3.1 A Succinct Representation of Border Values

Instead of storing the I/O values explicitly in arrays, we represent them with a series of two-dimensional points, which we call the *turning points* of arrays.

Definition 2. *Given a numerical array $S[1 \ldots \ell]$, we define $\Delta S(i) = S[i+1] - S[i]$ for $i \in [1, \ell-1]$. For simplicity's sake, we let $\Delta S(0) = \Delta S(\ell) = \infty$. We call $(i, S[i])$ a turning point of S if $\Delta S(i-1) \neq \Delta S(i)$ for $i \in [1, \ell]$. The geometric encoding of S, denoted by $GE(S)$, is the list of turning points of S from left to right.*

By definition $(1, S[1])$ and $(\ell, S[\ell])$ are the first and the last turning points of $S[1 \ldots \ell]$. If we plot points $(i, S[i])$ for all $i \in [1, \ell]$ in the plane and connect two neighboring points with a straight line, we obtain the *trajectory* (piecewise line segments) of S, and $GE(S)$ is the list of turning points of the trajectory from left to right. It is easily seen that $|GE(S)| \leq |S|$, where $|GE(S)|$ denotes the size of list $GE(S)$ and $|S|$ denotes the length of numerical array S.

Definition 3. *We call $(i, S[i])$ a valley point of S if $\Delta S(i-1) < 0$ and $\Delta S(i) > 0$ for $i \in [2, \ell - 1]$. Similarly, we let $VP(S)$ denote the list of valley points of S and $|VP(S)|$ denote its size.*

Figure 1 gives an example of turning points and valley points. As will be discussed in Section 5, the number of turning points and valley points in the block borders is crucial to the time analysis of our algorithm.

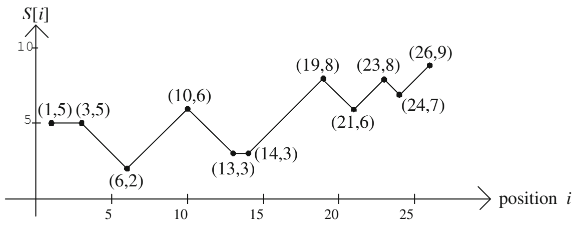

Fig. 1. The geometric encoding of array $S[1 \ldots 26] = 55543234565433456787678789$. Array S comprises 26 integers, whereas its geometric encoding $GE(S)$ is composed of 11 turning points. Among them there are three valley points, $(6, 2), (21, 6)$, and $(24, 7)$.

3.2 Propagation of Turning Points

We use geometric encoding to encode the border values. Since $ED(i, 0) = i$ for $1 \leq i \leq M$ and $ED(0, j) = j$ for $1 \leq j \leq N$, we have that each of the leftmost and topmost block borders of G contains exactly two turning points. Again, by Lemma 1 the propagation over a match block is easily handled by copying the turning points from the input border to the output border. Thus, the difficulty lies in the propagation over a mismatch block. Based on the discussion of Section 2.3, it can be handled by procedure PROPAGATE of Figure 2.

```
Procedure PROPAGATE
 Step 1: Use $GE(LEFT)$ and $GE(TOP)$ to compute $GE(LEFT^{(h)})$ and $GE(TOP^{(h)})$;
 Step 2: Use $GE(LEFT^{(h)})$ and $GE(TOP^{(h)})$ to compute $GE(OUT_{left})$ and $GE(OUT_{top})$;
 Step 3: Use $GE(OUT_{left})$ and $GE(OUT_{top})$ to compute $GE(OUT)$;
```

Fig. 2. Procedure of propagating turning points over a mismatch block

In Step 1 of PROPAGATE, we need to solve the SWM problem with its input and output arrays represented as lists of turning points. We call the problem the *continuous sliding-window minima problem* (abbreviated as CSWM).

Problem 2. Let S be a numerical array and h be a positive integer, denoting the window size. The CSWM problem is, given $GE(S)$, to compute $GE(S^{(h)})$.

The CSWM problem can be described graphically as follows. We are given a trajectory $\mathcal{T}$ in the plane, represented by its turning points, and a window $\mathcal{W}$, having a fixed width and an unlimited height. Window $\mathcal{W}$ is slid, from left to right, across $\mathcal{T}$ in a *continuous* manner, and at any time, the lowest point of the portion of $\mathcal{T}$ covered by $\mathcal{W}$ is plotted in the plane. The CSWM problem seeks for a list of turning points describing the resulting trajectory drawn as above. We will show in Section 4 that the CSWM problem can be solved in $O(|GE(S)|)$ time, a gain of efficiency over $O(|S|)$ time.

Given a point p, we let $x(p)$ and $y(p)$ denote its x-coordinate and its y-coordinate, i.e. $p = (x(p), y(p))$. According to Equations (3) and (4), Step 2 of Propagate can be handled as follows. We split list $GE(LEFT^{(h)})$ into two lists $\mathcal{L}_1 = GE(LEFT^{(h)}[1 \dots h])$ and $\mathcal{L}_2 = GE(LEFT^{(h)}[h \dots 2h-1])$. We retrieve each point $p \in \mathcal{L}_1$ in order and output point $(x(p), y(p) + x(p) - 1)$, and then retrieve point $q \in \mathcal{L}_2$ in order and output $(x(q) + w - h, y(q) + w - 1)$. This produces all the turning points of list $GE(OUT_{left})$. Similarly, we can easily compute list $GE(OUT_{top})$ from list $GE(TOP^{(h)})$.

According to Equation (2), Step 3 of Propagate can be handled by traversing lists $GE(OUT_{left})$ and $GE(OUT_{top})$ simultaneously and output the lower part of the two trajectories.

To conclude, both Step 2 and Step 3 of Propagate can be done in linear time. Hence, if the CSWM problem is also solvable in linear time, the propagation over a mismatch block can be done in time linear in the number of turning points of $LEFT$ and TOP, which implies the following crucial theorem.

Theorem 1. *The edit distance problem can be solved in $O(\mathcal{R})$ time, where $\mathcal{R}$ denotes the total number of turning points in all block borders.*

Note that the time of Theorem 1 is never worse than that of [3,6,13]. We will show in Section 5 that $\mathcal{R}$ is bounded by $O(mn^2)$, leading to the *fully compressed* feature of our algorithm.

4 A Linear-Time Algorithm for the Continuous Sliding-Window Minima Problem

In this section, we present a linear-time algorithm for the CSWM problem. We are given $GE(S)$, the geometric encoding of array S, and a positive integer h, the window size. Our algorithm simulates the window sliding from left to right across the trajectory of S, and performs actions whenever the right boundary of the window meets a turning point. We begin by describing the information maintained by our algorithm. Given point p and a positive number d, we define $p \oplus d = (x(p) + d, y(p))$. Similarly, given a list of points $\mathcal{L}$, we define $\mathcal{L} \oplus d$ to be the list of points obtained by moving the points in $\mathcal{L}$ distance d to the right.

Definition 4. *For each prefix array $S[1 \dots j]$ of $S[1 \dots \ell]$, $j \le \ell$, we define its suffix-minimum array as $SM_j[i] = \min\{S[i], S[i+1], \dots, S[j]\}$ for $i \le j$.*

Note that the values of SM_j are increasing, i.e. $SM_j[i] \le SM_j[i+1]$ for $i \in [1, j-1]$. Let $GE(S) = \langle s_1, s_2, \ldots, s_k \rangle$, where s_i is the i-th turning point of S. At iteration i, our algorithm needs to consult the values of $SM_{x(s_i)}[x(s_i) - h + 1 \ldots x(s_i)]$. In implementation, the algorithm maintains list $\mathcal{L} = GE(SM_{x(s_i)}[x(s_i) - h + 1 \ldots x(s_i)]) \oplus (h-1)$. See Figure 3 for an example. The window is of size 13 and its right boundary is currently at position 24. The bold lines depict the trajectory of $SM_{24}[12 \ldots 24]$, and the dashed lines depict the trajectory of list $\mathcal{L}$. As the window slides from position 24 to the right, the trajectory of $\mathcal{L}$ specifies the minimal values contributed by the values of S before position 24.

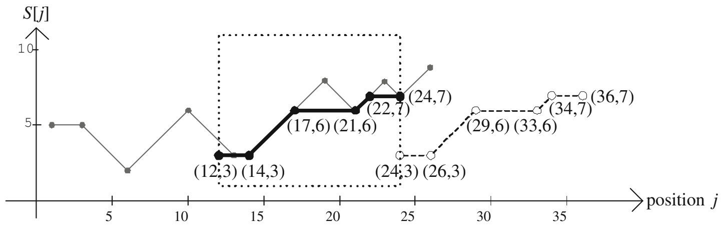

Fig. 3. The data structure, list $\mathcal{L}$, maintained by the algorithm

Our algorithm, named Slide (see Figure 4), is devised to follow the trajectory of $S^{(h)}$ from left to right and output the turning points of $S^{(h)}$ on the fly. Specifically, at iteration i the algorithm follows the sub-trajectory of $S^{(h)}[x(s_i) \ldots x(s_{i+1})]$ and outputs the turning points of $S^{(h)}[x(s_i) \ldots x(s_{i+1})]$.

Lemma 3. *Given a numerical array S and a positive number h, algorithm* SLIDE *correctly outputs a list of points containing $GE(S^{(h)})$ in $O(|GE(S)|)$ time.*

Proof. We show that at iteration i, the algorithm correctly computes the turning points of $S^{(h)}[x(s_i) \ldots x(s_{i+1})]$ and updates list $\mathcal{L}$ to $GE(SM_{i+1}[x(s_{i+1}) - h + 1 \ldots x(s_{i+1})]) \oplus (h-1)$ for the use of iteration $i+1$. Suppose by induction that list $\mathcal{L} = GE(SM_{x(s_i)}[x(s_i) - h + 1 \ldots x(s_i)]) \oplus (h-1)$ is computed at iteration $i-1$. Observe that for $j \in [x(s_i), x(s_i) + h - 1]$, $S^{(h)}[j] = \min\{S[j-h+1], \ldots, S[j]\} = \min(SM_{x(s_i)}[j-h+1], \min(S[x(s_i)], \ldots, S[j]))$. Hence, the trajectory of $S^{(h)}[x(s_i) \ldots x(s_{i+1})]$ is the lower part of the trajectories of $\mathcal{L}$ and $\overline{s_i s_{i+1}}$. The algorithm proceeds in the following cases (see Figure 5 for an illustration):

Case 1: The slope of $\overline{s_i s_{i+1}}$ is nonnegative, which implies that $\overline{s_i s_{i+1}}$ never goes below the trajectory of $\mathcal{L}$. The algorithm thus outputs the turning points belonging to $\mathcal{L}$ (see line 6). Observe that $SM_{x(s_{i+1})}[j] = SM_{x(s_i)}[j]$ for $j \in [x(s_i) - h + 1, x(s_i)]$, and $SM_{x(s_{i+1})}[j] = S[j]$ for $j \in (x(s_i), x(s_{i+1})]$. Hence, to update $\mathcal{L}$ the algorithm appends point $s_{i+1} \oplus (h-1)$ to its end and then cuts off its front piece before position $x(s_{i+1})$ (see lines 7–10).

```
Algorithm Slide
Input: $GE(S) = \langle s_1, s_2, \ldots, s_k \rangle$, where $x(s_1) < \ldots < x(s_k)$, and a positive number $h$.
Output: A list of points containing $GE(S^{(h)})$.
Output point $s_1$;
Initialize list $\mathcal{L} \leftarrow \langle s_1, s_1 \oplus (h-1) \rangle$;
for $i \leftarrow 1$ to $k-1$ do
    Retrieve the next point $s_{i+1}$ from $GE(S)$;
    Case 1: $y(s_i) \le y(s_{i+1})$
        Output all the points $p$ in $\mathcal{L}$, where $x(p) \in (x(s_i), x(s_{i+1})]$;
        Insert point $s_{i+1} \oplus (h-1)$ into the end of $\mathcal{L}$;
        Traverse $\mathcal{L}$ from left to right to point $q$ such that $x(q) = x(s_{i+1})$;
        Delete all the points $p$ in $\mathcal{L}$, where $x(p) \in [x(s_i), x(q)]$;
        Insert $q$ into the front of $\mathcal{L}$;
    Case 2: $y(s_i) > y(s_{i+1})$
        Traverse $\mathcal{L}$ from left to right and check if it intersects with $\overline{s_i s_{i+1}}$;
        Case 2a: there is no intersection.
            Output all the points $p$ in $\mathcal{L}$, where $x(p) \in (x(s_i), x(s_{i+1})]$;
            Traverse $\mathcal{L}$ from left to right to point $q$ such that $x(q) = x(s_{i+1})$;
            Delete all the points $p$ in $\mathcal{L}$, where $x(p) \in [x(s_i), x(q)]$
            Insert $q$ into the front of $\mathcal{L}$;
            Traverse $\mathcal{L}$ from right to left to the leftmost $q'$ such that $y(q') = y(s_{i+1})$;
            Delete all the points $p$ in $\mathcal{L}$, where $x(p) \in [x(q'), x(s_i) + h - 1]$;
            Insert $q'$ and $s_{i+1} \oplus (h-1)$ into the end of $\mathcal{L}$;
        Case 2b: there is an intersection $r$.
            Output all the points $p$ in $\mathcal{L}$, where $x(p) \in (x(s_i), x(r))$;
            Output points $r$ and $s_{i+1}$;
            Reset list $\mathcal{L} \leftarrow \langle s_{i+1}, s_{i+1} \oplus (h-1) \rangle$;
end for
Output all the points $p$ in $\mathcal{L}$, where $x(p) \in (x(s_k), x(s_k) + h - 1]$;
```

Fig. 4. Algorithm for the continuous sliding-window minima problem

Case 2: The slope of $\overline{s_i s_{i+1}}$ is negative.

- **Case 2a:** If $\overline{s_i s_{i+1}}$ does not intersect with the trajectory of $\mathcal{L}$, we have that $\overline{s_i s_{i+1}}$ never goes below the trajectory of $\mathcal{L}$. The algorithm again outputs the turning points belonging to $\mathcal{L}$. Observe that $SM_{x(s_{i+1})}[j] = SM_{x(s_i)}[j]$ for $j \in [x(s_i) - h + 1, x(q'))$, and $SM_{x(s_{i+1})}[j] = S[x(s_{i+1})]$ for $j \in [x(q'), x(s_{i+1})]$, where q' is the leftmost point in $\mathcal{L}$ such that $y(q') = y(s_{i+1})$. Therefore, to update list $\mathcal{L}$ the algorithm cuts off its front piece before position $x(s_{i+1})$ and its back piece after position $x(q')$, and then appends point $s_{i+1} \oplus (h-1)$ to its end (see lines 15–20).
- **Case 2b:** If $\overline{s_i s_{i+1}}$ intersects with the trajectory of $\mathcal{L}$ at point r, we have that the trajectory of $S^{(h)}[x(s_i) \ldots x(s_{i+1})]$ coincides with the trajectory of $\mathcal{L}$ before position $x(r)$ and coincides with $\overline{s_i s_{i+1}}$ after position $x(r)$. The algorithm thus outputs the turning points of $\mathcal{L}$ lying in $(x(s_i), x(r))$, and then points r and s_{i+1} (see lines 22–23). List $\mathcal{L}$ is reset to $\langle s_{i+1}, s_{i+1} \oplus (h-1) \rangle$ due to the fact that $SM_{x(s_{i+1})}[j] = S[x(s_{i+1})]$ for $j \in [x(s_{i+1}) - h + 1, x(s_{i+1})]$ (see line 24).

Observe that the total time spent in the above cases is proportional to the number of elements deleted from list $\mathcal{L}$. Since there are $O(|GE(S)|)$ points inserted into list $\mathcal{L}$, the algorithm runs in $O(|GE(S)|)$ time. $\square$

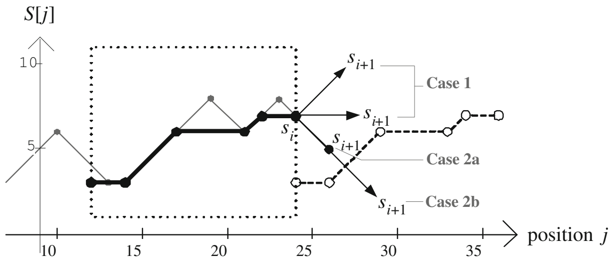

Fig. 5. The three possible positions of the next turning point s_{i+1}. The dashed lines depict the trajectory of $\mathcal{L}$, which should be compared with line segment $\overline{s_i s_{i+1}}$ in order to determine the trajectory of $S^{(h)}[x(s_i) \dots x(s_{i+1})]$.

According to algorithm Slide, below we prove two properties of the trajectory of $S^{(h)}$, which will be used in the next section.

Lemma 4. *Given a numerical array S and a positive number h, we have that $|GE(S^{(h)})| \leq |GE(S)| + |VP(S)|$.*

Proof. We examine the points output by algorithm Slide and show that among them there are at most $|GE(S)|+|VP(S)|$ turning points of $S^{(h)}$. Let σ_1, σ_2, and σ_3 denote the numbers of points of Case 1, Case 2a, and Case 2b, encountered by algorithm Slide, respectively. Note that $|GE(S)| = \sigma_1+\sigma_2+\sigma_3+1$. Observe that if s_i is a point of Case 1 or Case 2a, then s_i is not output but deposited in list $\mathcal{L}$ as point $s_i \oplus (h-1)$. If s_i is a point of Case 2b, it is output as well as deposited in $\mathcal{L}$ as point $s_i \oplus (h-1)$. That is, a point of Case 1 or Case 2a contributes at most one turning point to $S^{(h)}$, whereas a point of Case 2b may contribute two turning points. Hence, we have that $|GE(S^{(h)})| \leq \sigma_1 + \sigma_2 + 2\sigma_3 + 1 = |GE(S)| + \sigma_3$. We next examine those points of Case 2b in more details. Let s_i be a point of Case 2b. If the slope of $\overline{s_i s_{i+1}}$ is non-positive, the deposited point $s_i \oplus (h-1)$ will not be a turning point of $S^{(h)}$. This is because after iteration $i-1$, $\mathcal{L}$ is set to $\langle s_i, s_i \oplus (h-1)\rangle$, a horizontal line segment. Thus, if the slope of $\overline{s_i s_{i+1}}$ is non-positive, then either $\overline{s_i s_{i+1}}$ intersects with the trajectory of $\mathcal{L}$ immediately or both of their slopes are 0. Therefore, s_i contributes two turning points to $S^{(h)}$ only when the slope of $\overline{s_i s_{i+1}}$ is positive (in this case, s_i is a valley point). Hence, we can refine the bound into $|GE(S^h)| \leq |GE(S)| + |VP(S)|$. $\square$

Lemma 5. *Given a numerical array S and a positive number h, we have that $|VP(S^{(h)})| = 0$.*

Proof. Because the trajectory of $\mathcal{L}$ is always rising, algorithm Slide follows a negative-slope line only when a point of Case 2b is encountered. Observe that it cannot next follow a positive-slope line immediately, for $\mathcal{L}$ is set to a horizontal line segment after Case 2b. Hence, we have that the complete trajectory of $S^{(h)}$ contains no valley point, i.e. $|VP(S^{(h)})| = 0$. □

5 Time Complexity

Since there are mn propagations in total, if each propagation doubles the number of turning points, the number of turning points will grow in an exponential manner as they cascade down. Below, we show that the number of turning points can only grow by a constant in one propagation. This is obviously correct for propagations over match blocks, since they are simple copies of the turning points from the input border. Thus, we focus on propagations over mismatch blocks. We begin by introducing the *monotonicity property* of the border values. This property was observed by [1] and applied in [1,6,10,15].

Lemma 6 ([1,15]). *Given two vertices (x_4, y_4) and (x_3, y_3) in I, and two vertices (x_1, y_1) and (x_2, y_2) in O, such that $x_4 \leq x_3 \leq x_1 \leq x_2$ and $y_3 \leq y_4 \leq y_2 \leq y_1$, we have that if $ED(x_3, y_3) + dist(x_3, y_3, x_1, y_1) \leq ED(x_4, y_4) + dist(x_4, y_4, x_1, y_1)$, then $ED(x_3, y_3) + dist(x_3, y_3, x_2, y_2) \leq ED(x_4, y_4) + dist(x_4, y_4, x_2, y_2)$.*

Lemma 7. *The trajectories of OUT_{left} and OUT_{top}, obtained in Step 2 of* Propagate*, cross at most once. That is, if $OUT_{top}[i] \leq OUT_{left}[i]$ for some i, then $OUT_{top}[j] \leq OUT_{left}[j]$ for all $j \geq i$.*

Proof. Suppose that $OUT[i]$ and $OUT[j]$ correspond to vertex (x_1, y_1) and vertex (x_2, y_2) in G. From $OUT_{top}[i] \leq OUT_{left}[i]$, we know that there exists vertex (x_3, y_3) in the top border such that $ED(x_3, y_3) + dist(x_3, y_3, x_1, y_1) \leq ED(x_4, y_4) + dist(x_4, y_4, x_1, y_1)$ for all vertices (x_4, y_4), $y_4 \leq y_2$, in the left border. By Lemma 6 we have that $ED(x_3, y_3) + dist(x_3, y_3, x_2, y_2) \leq ED(x_4, y_4) + dist(x_4, y_4, x_2, y_2)$ for all vertices (x_4, y_4), $y_4 \leq y_2$, in the left border. Since $OUT_{left}[j] \leq ED(x_3, y_3) + dist(x_3, y_3, x_2, y_2)$, the lemma thus follows. □

Lemma 8. *After Step 1 of* Propagate*, we have that $|GE(LEFT^{(h)})| \leq |GE(LEFT)| + 1$ and $|GE(TOP^{(h)})| \leq |GE(TOP)| + 1$.*

Proof. We show by induction that the number of valley points in each block border is at most one, and the lemma thus follows from Lemma 4. Initially, each of the leftmost and topmost block borders contains no valley point. By Lemma 1, the propagation over a match block produces no extra valley point in the output border. As for the propagation over a mismatch block, by Lemma 5 we know that Step 1 leads to no valley point in $LEFT^{(h)}$ and $TOP^{(h)}$. Step 2 clearly produces no extra valley point, and we know by Lemma 7 that Step 3 produces at most one valley point. Hence, we conclude that the propagation over a mismatch block results in at most one valley point in the output border. □

It is also not hard to see that after Steps 2–3 of Propagate, the number of turning points can only grow by a constant. Now, we are ready to prove the time complexity of our algorithm.

Theorem 2. *Given two strings A and B, compressed into m and n runs, $m \leq n$, computing the edit distance between A and B can be done in $O(mn^2)$ time.*

Proof. Let $H_{i,j}$ denote the block corresponding to the i-th run of A and the j-th run of B for $i \in [1, m]$ and $j \in [1, n]$. Let $u_{i,j}$ (resp., $v_{i,j}$) denote the number of turning points in the top border (resp., left border) of $H_{i,j}$. Let $U_i = \sum_{j=1}^{n} u_{i,j}$ and $V_j = \sum_{i=1}^{m} v_{i,j}$. By Theorem 1, the time of our algorithm is proportional to $\sum_{i=1}^{m} U_i + \sum_{j=1}^{n} V_j$. Let $v_{i',n+1}$ (resp., $u_{m+1,j'}$) denote the number of turning points in the bottom border of $H_{n,j'}$ (resp., the right border of $H_{i',n}$) for $i' \in [1, m]$ (resp., for $j' \in [1, n]$). Since the number of turning points can only grow by a constant c in each propagation, we have that $u_{i+1,j} + v_{i,j+1} \leq u_{i,j} + v_{i,j} + c$ for all $i \in [1, m]$, $j \in [1, n]$. Hence, we have that $\sum_{j=1}^{n}(u_{i+1,j} + v_{i,j+1}) \leq \sum_{j=1}^{n}(u_{i,j} + v_{i,j}) + cn$, implying $\sum_{j=1}^{n} u_{i+1,j} \leq \sum_{j=1}^{n} u_{i,j} + cn + v_{i,1} - v_{i,n+1} \leq \sum_{j=1}^{n} u_{i,j} + (cn + 1)$. That is, $U_{i+1} \leq U_i + (cn + 1)$. Since $U_1 = \sum_{j=1}^{n} u_{1,j} = 2n$, it is not hard to derive that $\sum_{i=1}^{m} U_i = O(m^2 n)$. Similarly, we can also derive $\sum_{j=1}^{n} V_j = O(mn^2)$. The theorem thus follows. □

To recover an optimal edit trace, we start from vertex (M, N) and trace a series of block-crossing paths back to vertex $(0, 0)$. This requires additional computation and storage of information during the propagation stage. As long as we associate each point output by algorithm Slide with its source entry, we can trace, for each vertex in the output border, back to its optimal source vertex in the input border. This requires $O(mn^2)$ space in total. Hirschberg's space reduction method [9] can be used to reduce the space to $O(mn)$ without impairing the $O(mn^2)$ time bound.

6 Concluding Remarks

We present the first fully compressed algorithm for computing the edit distance between two RLE strings, of m and n runs, in $O(mn^2)$ time and $O(mn)$ space. The approximate matching problem is, given pattern P and text T, to identify substrings T' of T such that the edit distance between P and T' is at most k. The dynamic programming solution for this problem requires the following setting. The vertex values in the first row of the edit graph are initialized as zeros, and the goal becomes to identify the vertex values in the last row that are less than or equal to k. Our approach can be easily adapted to this setting within the same time and space bound. Furthermore, our algorithm in fact runs in $O(\mathcal{R})$ time, where $\mathcal{R}$ denotes the total number of turning points in the block borders. In the paper, we prove that $\mathcal{R}$ is bounded by $O(mn^2)$. Providing a tighter bound for $\mathcal{R}$ implies better time complexity of our algorithm.

References

1. Aggarwal, A., Park, J.K.: Notes on Searching in Multidimensional Monotone Arrays. In: FOCS 1998, pp. 497–512 (1998)
2. Apostolico, A., Landau, G.M., Skiena, S.: Matching for Run-Length Encoded Strings. Journal of Complexity 15(1), 4–16 (1999)

3. Arbell, O., Landau, G.M., Mitchell, J.S.B.: Edit Distance of Run-Length Encoded Strings. Information Processing Letters 83(6), 307–314 (2002)
4. Bunke, H., Csirik, J.: An Improved Algorithm for Computing the Edit Distance of Run-Length Coded Strings. Information Processing Letters 54(2), 93–96 (1995)
5. Chen, K.-Y., Hsu, P.-H., Chao, K.-M.: Approximate Matching for Run-Length Encoded Strings Is 3SUM-Hard. Journal of Complexity (accepted); A preliminary version appeared in CPM 2009 (2009)
6. Crochemore, M., Landau, G.M., Ziv-Ukelson, M.: A Subquadratic Sequence Alignment Algorithm for Unrestricted Scoring Matrices. SIAM Journal on Computing 32(6), 1654–1673 (2003)
7. Gajewska, H., Tarjan, R.E.: Deques with Heap Order. Information Processing Letters 22(4), 197–200 (1986)
8. Hermelin, D., Landau, G.M., Landau, S., Weimann, O.: A Unified Algorithm for Accelerating Edit-Distance Computation via Text-Compression. In: STACS, pp. 529–540 (2009)
9. Hirschberg, D.S.: A Linear Space Algorithm for Computing Maximal Common Subsequences. Communications of the ACM 18(6), 341–343 (1975)
10. Huang, G.-S., Liu, J.J., Wang, Y.-L.: Sequence Alignment Algorithms for Run-Length-Encoded Strings. In: Hu, X., Wang, J. (eds.) COCOON 2008. LNCS, vol. 5092, pp. 319–330. Springer, Heidelberg (2008)
11. Liu, J.J., Huang, G.-S., Wang, Y.-L., Lee, R.C.-T.: Edit Distance for a Run-Length-Encoded String and an Uncompressed String. Information Processing Letters 105(1), 12–16 (2007)
12. Masek, W.J., Paterson, M.: A Faster Algorithm Computing String Edit Distances. Journal of Computer and System Sciences 20(1), 18–31 (1980)
13. Mäkinen, V., Ukkonen, E., Navarro, G.: Approximate Matching of Run-Length Compressed Strings. Algorithmica 35(4), 347–369 (2003)
14. Mitchell, J.S.B.: A Geometric Shortest Path Problem, with Application to Computing a Longest Common Subsequence in Run-Length Encoded Strings. Technical Report, SUNY Stony Brook (1997)
15. Schmidt, J.P.: All Highest Scoring Paths in Weighted Grid Graphs and Their Application to Finding All Approximate Repeats in Strings. SIAM Journal on Computing 27(4), 972–992 (1998)