Algorithmica (2013) 65:354–370
DOI 10.1007/s00453-011-9592-4

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

Kuan-Yu Chen · Kun-Mao Chao

Received: 13 January 2011 / Accepted: 2 November 2011 / Published online: 12 November 2011

Abstract A recent trend in stringology explores the possibility of utilizing text compression to speed up similarity computation between strings. In this line of investigation, run-length encoding is one of the earliest studied compression schemes. Despite its simple coding nature, the only positive result before this work is the computation of the in-del distance (dual of longest common subsequence), which requires $O(mn \log mn)$ time, where m and n denote the number of runs of the input strings. The worst-case time complexity of computing the edit distance between two run-length encoded strings still depends on the uncompressed string lengths. In this paper, we break the foundational gap by providing its first "fully compressed" algorithm whose running time depends solely on the compressed string lengths. 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 strings. Our approach also yields the first fully compressed solution to approximate matching of a pattern of m runs in a text of n runs in $O(mn^2)$ time.

Keywords Compressed pattern matching · Edit distance · Run length

1 Introduction

Edit distance, also known as Levenshtein distance, serves as a common similarity measure between strings. It is defined as the minimum number of steps required to

A preliminary version of this work appeared in the 18th Annual European Symposium on Algorithms, United Kingdom, 2010.

K.-Y. Chen · K.-M. Chao (✉)
Department of Computer Science and Information Engineering, National Taiwan University, Taipei 106, Taiwan
e-mail: kmchao@csie.ntu.edu.tw

transform one string into the other via operations of insertion, deletion, or substitution. The computation of edit distance between two strings has been explored for decades. The classic dynamic programming solution, proposed by Wagner and Fischer [24], 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 [12, 17] for more details). The first breakthrough to $O(N^2/\log N)$-time was made by Masek and Paterson [19], who used the Four-Russians technique to speed up the computation of edit distance. The Four-Russians technique can be seen as a naïve compression which utilizes the fact that sufficiently short substrings must appear many times in a string over a constant alphabet. Another breakthrough was made by Crochemore et al. [9], who exploited the LZ-factorization of the input strings. The algorithm given in [9] spends $O(hN^2/\log N)$ time, where $h \leq 1$ is the entropy of the strings, and is general enough for the sequence alignment problem with an unrestricted scoring matrix.

In this paper, we accelerate the edit-distance computation by exploiting symbol repetitions in strings. The underlying compression scheme considered in this work is called *run-length encoding* (abbreviated as RLE), which compresses consecutive identical symbols into a run, usually denoted by σ^i, where σ is an alphabet symbol and i denotes its repetition times. For example, string $bbcccddaaaaa$ can be encoded as the RLE format $b^2c^3d^2a^5$. This encoding is largely applied in applications such as FAX transmission, optical character recognition, and several image compression standards. Previous results of computing the similarity between RLE strings include the computation of the in-del distance (dual of longest common subsequence), which requires $O(mn \log mn)$ time [5, 21], where m and n denote the number of runs of the input strings. Several papers showed how to compute the edit distance of RLE strings in $O(Mn + mN)$ time [6, 9, 20], where M and N denote the uncompressed string sizes. Recently, Huang et al. [14] and Liu et al. [18] improved the bound to $O(\min\{Mn, mN\})$ time. It should be noted that the works of [6, 18] focused on Levenshtein distance in which each edit operation has a unit cost, while the results of [9, 14, 20] apply to the more general sequence alignment problem with an unrestricted scoring matrix. To date, all the time bounds for the edit distance computation still depend on the uncompressed string lengths. Therefore, an intriguing question is (as asked in [14, 17, 20]) whether one can design an algorithm that is "fully compressed," i.e. its time complexity depends solely on the number of runs of both RLE strings. This paper answers the question affirmatively for the case of Levenshtein distance by presenting an $O(mn^2)$-time algorithm.

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*. This problem has been explored under various existing compression schemes, e.g. RLE scheme [2, 3], LZ-family compression schemes [4, 10], straight-line programs [15, 16], etc. For RLE scheme, the exact string matching problem can be solved optimally in $O(m + n)$ time, where m and n denote the number of runs of pattern and text. The k-mismatch with wildcards problem can be solved in $O(mn \log m)$ time [8]. As for the *approximate matching problem*, which seeks for occurrences of an RLE string within another RLE string,

allowing up to k edit operations, Mäkinen et al. proposed an $O(mnM)$-time algorithm [20], where M denotes the uncompressed pattern length. Huang et al. later improved the bound to $O(nM)$ time [14]. The approach presented in this paper gives its first fully compressed solution of $O(mn^2)$ time.

2 Preliminaries

2.1 A Propagation-Based Approach

Given two strings A and B of lengths M and N, the edit distance problem can be solved via dynamic programming 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 then the value stored in $ED(M, N)$. This dynamic programming solution can be represented in terms of a weighted, acyclic grid graph G, called an *alignment graph*, of $(M+1) \times (N+1)$ vertices. Each vertex (i, j) stores the value of $ED(i, j)$, and any shortest path from $(0, 0)$ to (M, N) in G specifies an optimal *edit trace* (a sequence of edit operations).

Let m and n denote the number of runs of A and B. Based on the RLE factorization of A and B, the alignment graph G is partitioned into $m \times n$ blocks. Note that two adjacent blocks in G share the vertices in their adjoining borders. As observed in [6, 9, 20], instead of computing all the vertices in G, it suffices to compute only the vertices in block borders. More specifically, given a block H of G, we refer to the left and top borders of H as its *input border* and the bottom and right borders of H as its *output border*. The algorithms proposed in [6, 9, 20] traverse the $m \times n$ blocks of G in a left-to-right, top-to-bottom order and propagate the accumulated values from the input borders to the output borders without filling in the vertices lying in-between. See Fig. 1 for an example.

2.2 Problem Reduction

In this subsection, we briefly review the observations made in previous work. Let $\text{dist}(i', j', i, j)$ denote the weight of a shortest path from vertex (i', j') to vertex (i, j) of G, where $i' \leq i$ and $j' \leq j$. Given a block H of G, we let I and O denote its input and output borders and (i_0, j_0) denote its upper-left corner. We say that vertex (i, j) of G is in *diagonal* $j - i$ of G. For each vertex (i, j) in O, we have the following recurrence relation:

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

This recurrence relation can be further simplified for match and mismatch blocks, as shown in the following two lemmas. Let (i_d, j_d) be the intersection of I with diagonal $j - i$ of G.

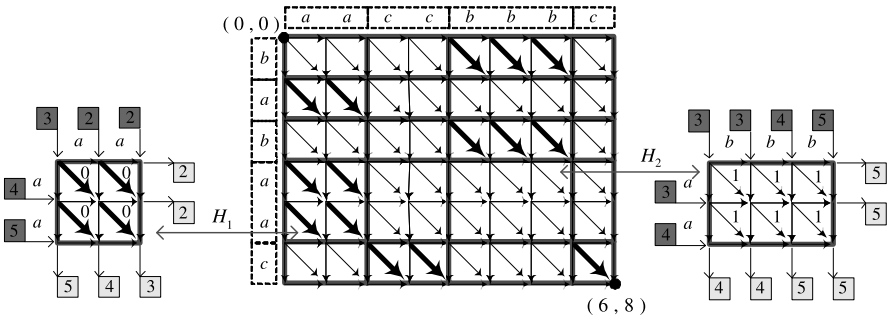

Fig. 1 The alignment graph G of strings $A = b^1a^1b^1a^2c^1$ and $B = a^2c^2b^3c^1$. The subgraph H_1 shows a *match block*, which corresponds to a run of A and a run of B that encode an identical symbol, whereas H_2 shows a *mismatch block*. All the edges in G are assigned weight 1 except for the diagonal edges in match blocks, which are assigned weight 0. The values in the input borders of H_1 and H_2 are shown as *dark-grey shaded rectangles*, and the values in the output borders of H_1 and H_2 are shown as *light-grey shaded rectangles*. The values in the input borders can be directly propagated to the corresponding output borders, bypassing the computation of vertices lying in between

Lemma 1 [7] *If H is a match block, then $ED(i, j) = ED(i_d, j_d)$.*

Lemma 2 [6, 20] *If H is a mismatch block, then* $ED(i, j) = \min\{\min_{i_d \le i' \le i}(ED(i', j_0) + j - j_0), \min_{j_d \le j' \le j}(ED(i_0, j') + i - i_0)\}$.

By Lemma 1, the propagation over a match block is easy. To obtain the 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 *sliding-window minima problem*, defined as follows.

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 to be $S^{(h)}[i] = \min\{S[j] \mid i - h + 1 \le j \le i \text{ and } 1 \le j \le \ell\}$ for $i \in [1, \ell + h - 1]$. The sliding-window minima problem (abbreviated as the SWM problem) is, given array S, to compute array $S^{(h)}$.

Let $LEFT[1 \ldots h]$ and $TOP[1 \ldots w]$ denote the values of the left border and the top border of I such that the entries of $LEFT$ and TOP are numbered in a bottom-to-top and a left-to-right direction, respectively. Let $OUT[1 \ldots w + h - 1]$ denote the values of O such that the entries of OUT are numbered in a counterclockwise direction, starting from the lower-left corner. For simplicity, we only consider the case where block H is flat, i.e. $h \le w$. The other case where $h > w$ can be argued symmetrically.

Definition 1 For $i \in [1, w + h - 1]$, we let $LEFT_2[i]$ (resp., $TOP_2[i]$) denote the weight of a shortest path passing through the left border (resp., top border) of I and ending at the vertex of $OUT[i]$.

By Lemma 2, $LEFT_2$ and TOP_2 can be expressed in terms of the sliding-window minima arrays of $LEFT$ and TOP as follows.

$$LEFT_2[i] = \begin{cases} LEFT^{(h)}[i] + i - 1, & \text{for } i \in [1, h]; \\ LEFT^{(h)}[h] + i - 1, & \text{for } i \in [h, w]; \\ LEFT^{(h)}[i - w + h] + w - 1, & \text{for } i \in [w, w + h - 1]; \end{cases} \tag{2}$$

$$TOP_2[i] = \begin{cases} TOP^{(h)}[i] + h - 1, & \text{for } i \in [1, w]; \\ TOP^{(h)}[i] + w + h - 1 - i, & \text{for } i \in [w, w + h - 1]. \end{cases} \tag{3}$$

Moreover, by (1) we have that:

$$OUT[i] = \min\{LEFT_2[i], TOP_2[i]\} \quad \text{for } i \in [1, w + h - 1]. \tag{4}$$

Hence, once we obtain arrays $LEFT^{(h)}$ and $TOP^{(h)}$, it is easy to compute array OUT through Equations (2)–(4). The propagation over a mismatch block thus relies on solving two instances of the SWM problem. By utilizing *deques with heap orders* proposed in [11], one can solve the SWM problem in time linear in the input array. Thus, the propagation over a mismatch block can be done in time linear in the size of its input border, which implies the result given in [6, 9, 20].

Theorem 1 [6, 9, 20] *Computing the edit distance between two run-length encoded strings can be done in time linear in the total size of block borders of G, i.e. $O(Mn + mN)$ time.*

3 A Geometric View

In this section, we show that the propagation-based approach described in the previous section can be further accelerated by representing the border values in a geometric form.

3.1 A Succinct Representation of the Border Values

Instead of storing the border values explicitly in arrays, we represent them with a series of two-dimensional points, termed the *turning points*. We call such succinct representation the *geometric encoding* of arrays, defined formally as follows.

Definition 2 Given a numerical array $S[1 \ldots \ell]$, we define $\Delta S(i) = S[i] - S[i-1]$ for $i \in [2, \ell]$. For simplicity's sake, we let $\Delta S(1) = \Delta S(\ell + 1) = \infty$. We call $(i, S[i])$ a *turning point* of S if $\Delta S(i) \neq \Delta S(i + 1)$ 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. Pictorially, if we plot all $(i, S[i])$ in the plane and connect two adjacent points with a straight line segment, we obtain a *trajectory* in the plane, which can be represented by the list of points $GE(S)$. For ease of notation, we use $GE(S)$ to denote both the

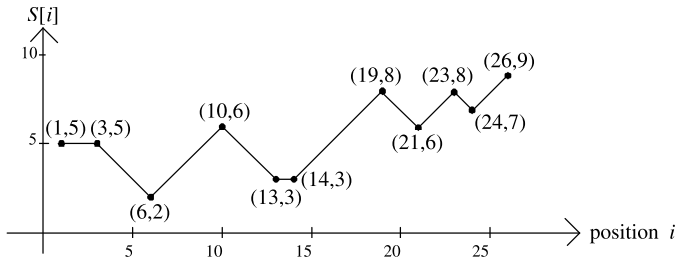

Fig. 2 The geometric encoding of $S[1 \ldots 26] = 55543234565433456787678789$. Array S comprises 26 numbers, whereas its geometric encoding $GE(S)$ is composed of 11 turning points. Among them there are three valley points, $(6, 2)$, $(21, 6)$, and $(24, 7)$

Procedure PROPAGATE

Step 1: Use lists $GE(LEFT)$ and $GE(TOP)$ to compute lists $GE(LEFT^{(h)})$ and $GE(TOP^{(h)})$;
Step 2: Use lists $GE(LEFT^{(h)})$ and $GE(TOP^{(h)})$ to compute lists $GE(LEFT_2)$ and $GE(TOP_2)$;
Step 3: Use lists $GE(LEFT_2)$ and $GE(TOP_2)$ to compute list $GE(OUT)$;

Fig. 3 Procedure of propagating turning points over a mismatch block

list of turning points of S and the trajectory represented by those points. We shall add the words "list" or "trajectory" in front of $GE(S)$ for the sake of clarity. Notice that $|GE(S)| \leq |S|$, where $|GE(S)|$ denotes the size of list $GE(S)$ and $|S|$ denotes the size of array S.

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

See Fig. 2 for an example of turning points and valley points. As will be shown in Sect. 5, the number of turning points and valley points in the block borders is crucial to the time analysis of our algorithm.

3.2 Propagation of Turning Points

We now use geometric encoding to represent the border values of G. Since $ED(i, 0) = i$ for $1 \leq i \leq M$ and $ED(0, j) = j$ for $1 \leq j \leq N$, each of the leftmost and topmost border values of G can be represented by a line segment, containing exactly two turning points. Our approach then propagates the turning points in a left-to-right, top-to-bottom order, over the mn blocks.

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 once again lies in the propagation over a mismatch block, which, by (2)–(4), can be handled by procedure PROPAGATE of Fig. 3.

In Step 1 of PROPAGATE, we need to solve the SWM problem with its input and output arrays encoded as trajectories. We call the problem the *continuous sliding-window minima problem* (abbreviated as CSWM), defined formally as follows.

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 a list of 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 the list of turning points representing the trajectory drawn as above.

Step 1 of PROPAGATE requires transforming the two input border trajectories, $GE(LEFT)$ and $GE(TOP)$, into their sliding-window minima trajectories, $GE(LEFT^{(h)})$ and $GE(TOP^{(h)})$. We will show in Sect. 4 that this step can be done in time linear in $|GE(LEFT)|$ and $|GE(TOP)|$.

Step 2 of PROPAGATE can be handled as follows. Given a point p, we let $x(p)$ and $y(p)$ denote its x- and y-coordinates. We first split trajectory $GE(LEFT^{(h)})$ into two sub-trajectories $\mathcal{L}_1 = GE(LEFT^{(h)}[1 \ldots h])$ and $\mathcal{L}_2 = GE(LEFT^{(h)}[h \ldots 2h-1])$. We next replace each point $p \in \mathcal{L}_1$ with point $(x(p), y(p) + x(p) - 1)$ and each point $q \in \mathcal{L}_2$ with point $(x(q) + w - h, y(q) + w - 1)$. List $GE(LEFT_2)$ is then the concatenation of the adjusted $\mathcal{L}_1$ and $\mathcal{L}_2$. Similarly, list $GE(TOP_2)$ can be easily computed from list $GE(TOP^{(h)})$.

Step 3 of PROPAGATE can be done by traversing trajectories $GE(LEFT_2)$ and $GE(TOP_2)$ simultaneously and outputting their lower part.

To conclude, if the claimed time bound for the CSWM problem is correct, the three steps of PROPAGATE can be done in time linear in $|GE(LEFT)|$ and $|GE(TOP)|$, which implies the following crucial theorem of the paper.

Theorem 2 *Computing the edit distance between two run-length encoded strings can be done in $O(\mathcal{R})$ time, where $\mathcal{R}$ denotes the total number of turning points in all block borders of G.*

We will show in Sect. 5 that $\mathcal{R}$ is bounded by $O(mn^2)$, leading to the fully compressed feature of our algorithm. Note that the time of Theorem 2 is never worse than that of Theorem 1, as the number of turning points in each block border is bounded by the border size.

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

In this section, we show how to handle Step 1 of PROPAGATE by providing a linear-time algorithm for the CSWM problem. Given list $GE(S)$ and a window size h, we aim to compute list $GE(S^{(h)})$ in $O(|GE(S)|)$ time. Our algorithm acts like a sweep-line approach which simulates a vertical scan-line sliding across the plane and stopping at each turning point of $GE(S)$. From a high-level viewpoint, the algorithm will trace the trajectory of $GE(S^{(h)})$ from left to right and output all its turning points on the fly.

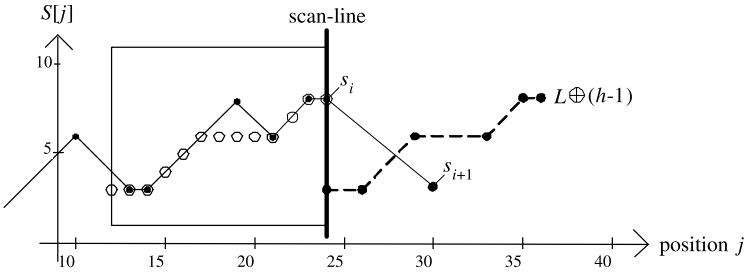

Fig. 4 The auxiliary trajectory $\mathcal{L}$ maintained by the algorithm. The rectangle adjacent to the scan-line, currently stopping at position 24, shows a sliding window of size 13. The *hollow circles* depict the trajectory of $\mathcal{L}$. The *dashed lines* depict the trajectory of $\mathcal{L} \oplus (h-1)$, which should be compared with $\overline{s_i s_{i+1}}$, in order to determine the sub-trajectory of $GE(S^{(h)})$ in region $(x(s_i), \min\{x(s_i)+h-1, x(s_{i+1})\}]$

4.1 The Auxiliary Trajectory

Let $GE(S) = \langle s_1, s_2, \ldots, s_k \rangle$, where s_i denotes the i-th turning point of S. At iteration i, as the scan-line slides from position $x(s_i)$ to position $x(s_{i+1})$, the algorithm will trace the sub-trajectory of $GE(S^{(h)})$ in region $(x(s_i), x(s_{i+1})]$ and output all the turning points lying in that region. To this end, the algorithm needs to maintain the auxiliary trajectory $\mathcal{L}$ described as follows.

Definition 4 For each position j of S, we define the *suffix-minimum array* $SM_j[i] = \min\{S[i], S[i+1], \ldots, S[j]\}$ for $i \le j$.

At iteration i, our algorithm needs to consult the values of $SM_{x(s_i)}[x(s_i)-h+1 \ldots x(s_i)]$, which summarizes the minimal values of S before position $x(s_i)$. In implementation, the algorithm maintains $\mathcal{L} = GE(SM_{x(s_i)}[x(s_i)-h+1 \ldots x(s_i)])$. Once again, we will use $\mathcal{L}$ to denote both a list of discrete points and the trajectory represented by those points. We shall add words like "list" or "trajectory" in our description to avoid confusion. Observe that by definition, the trajectory of $\mathcal{L}$ never descends and its last point is point s_i.

Given a point p and an integer d, we define $p \oplus d$ to be the point $(x(p)+d, y(p))$. Similarly, given a list of points $\mathcal{P}$, we define $\mathcal{P} \oplus d$ to be the list of points obtained by moving all the points of $\mathcal{P}$ distance d to the right. The usage of list $\mathcal{L}$ can be seen by the following observation. By definition, $S^{(h)}[j] = \min\{S[j-h+1], \ldots, S[j]\} = \min\{SM_{x(s_i)}[j-h+1], S[x(s_i)], \ldots, S[j]\}$ for $x(s_i) < j \le \min\{x(s_i)+h-1, x(s_{i+1})\}$. Hence, the sub-trajectory of $GE(S^{(h)})$ in region $(x(s_i), \min\{x(s_i)+h-1, x(s_{i+1})\}]$ coincides with the lower part of line segment $\overline{s_i s_{i+1}}$ and trajectory $\mathcal{L} \oplus (h-1)$. Figure 4 shows an example.

4.2 The Algorithm

Our algorithm, named SLIDE, for the CSWM problem is presented in Fig. 5. It should be noted that when we refer to $\mathcal{L}$ as "list $\mathcal{L}$", we treat $\mathcal{L}$ as a data structure which stores a list of discrete points; however, when we refer to $\mathcal{L}$ as "trajectory $\mathcal{L}$", we treat $\mathcal{L}$

Algorithm SLIDE
Input: List $GE(S) = \langle s_1, s_2, \ldots, s_k \rangle$ and a window size $h > 0$.
Output: List $GE(S^{(h)})$.
1 **output** point s_1;
2 Set list $\mathcal{L} = \langle s_1 \oplus (-h+1), s_1 \rangle$;
3 **for** $i = 1$ to $k-1$ **do**
4 Retrieve the next point s_{i+1} from list $GE(S)$;
5 **Case 1**: If $\overline{s_i s_{i+1}}$ is strictly above trajectory $\mathcal{L} \oplus (h-1)$ after position $x(s_i)$;
6 **output** those points of list $\mathcal{L} \oplus (h-1)$ whose x-coordinates lie in $(x(s_i), x(s_{i+1})]$;
7 **truncate** the back part of trajectory $\mathcal{L}$ that is higher than point s_{i+1};
8 Insert point s_{i+1} into the end of list $\mathcal{L}$;
9 **truncate** the front part of trajectory $\mathcal{L}$ that is before position $x(s_{i+1}) - h + 1$;
10 **Case 2**: Otherwise, let r be the leftmost intersection of $\overline{s_i s_{i+1}}$ and trajectory $\mathcal{L} \oplus (h-1)$;
11 **output** those points of list $\mathcal{L} \oplus (h-1)$ whose x-coordinates lie in $(x(s_i), x(r))$;
12 **output** points r and s_{i+1} in order;
13 Set list $\mathcal{L} = \langle s_{i+1} \oplus (-h+1), s_{i+1} \rangle$;
14 **end for**
15 **output** those points of list $\mathcal{L} \oplus (h-1)$ whose x-coordinates lie in $(x(s_k), x(s_k) + h - 1]$;

Fig. 5 Algorithm for the continuous sliding-window minima problem

as a continuous path in the plane obtained by connecting those discrete points. For example, in lines 7 and 9 the **truncate** operation on trajectory $\mathcal{L}$ means to remove a portion from the continuous path, which in effect would delete some points from and insert a new point into the data structure of $\mathcal{L}$. In contrast, in line 8 the insertion of s_{i+1} into list $\mathcal{L}$ actually puts s_{i+1} into the data structure, which in effect introduces a new line segment to the continuous path of $\mathcal{L}$.

Notice that the first point of list $\mathcal{L}$ will serve as a dummy point, which never contributes a point to the output throughout the procedure. This can be seen from lines 6, 11, and 15, in which the left boundary of the output region is always open. Moreover, the **output** operation in lines 1, 6, 11, 12, and 15 performs an extra check before it appends a point to the output sequence. More specifically, let u_1 and u_2 be the last two points output by the algorithm and let v be the point to be output. The operation checks if u_2 is contained in $\overline{u_1 v}$. If so, u_2 is removed from the output sequence. With this extra check, no repeated point or redundant point remains in the output.

We now examine algorithm SLIDE line by line. Initially, point s_1 is output and the scan-line begins at position $x(s_1)$ (line 1). Since s_1 is the only point in the window at this moment, the auxiliary trajectory $\mathcal{L}$ is initialized as the horizontal line segment with the two ends $s_1 \oplus (-h+1)$ and s_1 (line 2). At each iteration of the for-loop, the algorithm proceeds in two cases, depending on whether $\overline{s_i s_{i+1}}$ is strictly above trajectory $\mathcal{L} \oplus (h-1)$ or not (lines 5–13).

In Case 1 (lines 5–9), we assume that $\overline{s_i s_{i+1}}$ is strictly above trajectory $\mathcal{L} \oplus (h-1)$ after position $x(s_i)$. We consider two possible subcases. If $x(s_{i+1}) \leq x(s_i) + h - 1$, we have that the complete trajectory of $GE(S^{(h)})$ in region $(x(s_i), x(s_{i+1})]$ coincides with trajectory $\mathcal{L} \oplus (h-1)$. One can see that the algorithm correctly handles this subcase (line 6). However, if $x(s_{i+1}) > x(s_i) + h - 1$, we have that only the part of trajectory $GE(S^{(h)})$ in region $(x(s_i), x(s_i) + h - 1]$ coincides with trajectory $\mathcal{L} \oplus (h-1)$. Note that in this subcase, the slope of $\overline{s_i s_{i+1}}$ must be positive, since otherwise $\overline{s_i s_{i+1}}$ will intersect with trajectory $\mathcal{L} \oplus (h-1)$. Thus, we have

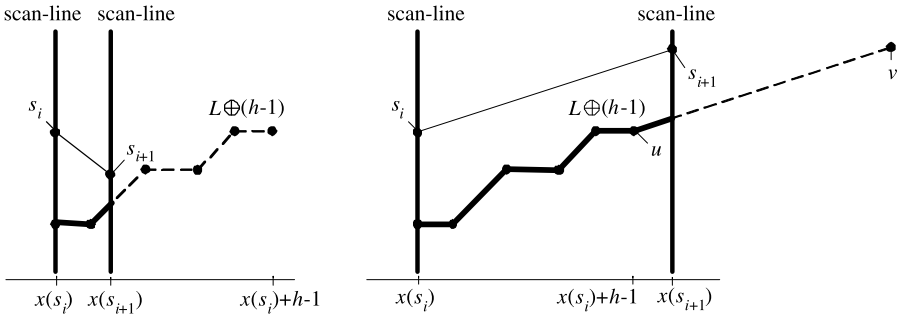

Fig. 6 Two subcases of Case 1. The *left side* of the figure shows the case where $x(s_{i+1}) \leq x(s_i) + h - 1$, and the *right side* of the figure shows the other case. As the scan-line moves from position $x(s_i)$ to position $x(s_{i+1})$, the algorithm traces the trajectory of the non-vertical *bold lines*

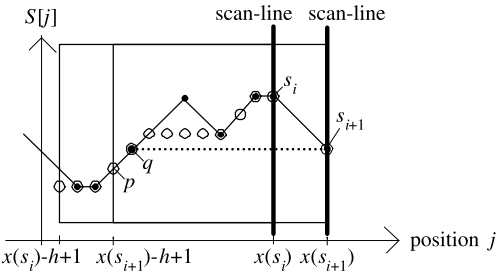

Fig. 7 The update of trajectory $\mathcal{L}$ in Case 1. The update involves the truncations of both ends of trajectory $\mathcal{L}$, at positions $x(p)$ and $x(q)$ respectively, followed by the insertion of the horizontal line segment $\overline{qs_{i+1}}$ into the end of trajectory $\mathcal{L}$

that $S^{(h)}[j] = \min\{S[j-h+1], \ldots, S[j]\} = S[j-h+1]$ for $x(s_i) + h - 1 < j \leq x(s_{i+1})$, which implies that the algorithm should trace line segment $\overline{uv}$ in region $(x(s_i) + h - 1, x(s_{i+1})]$, where $u = s_i \oplus (h-1)$ and $v = s_{i+1} \oplus (h-1)$. Moreover, as point s_{i+1} is strictly above $\overline{uv}$, the algorithm will continue (at least shortly) to trace $\overline{uv}$ after position $x(s_{i+1})$. This can be seen from the fact that in the next iteration, the intersection of $\overline{s_{i+1}s_{i+2}}$ and $\overline{uv}$, if exists, can only take place after position $x(s_{i+1})$. Hence, we conclude that $GE(S^{(h)})$ contains no extra turning point in region $(x(s_i)+h-1, x(s_{i+1})]$. The algorithm thus properly handles the latter subcase as well (line 6). Figure 6 provides an illustration for both subcases.

We now examine the update of trajectory $\mathcal{L}$ in Case 1 (lines 7–9), which is handled by the following two steps. See Fig. 7 for an illustration.

1. We first update trajectory $\mathcal{L}$ to $GE(SM_{x(s_{i+1})}[x(s_i) - h + 1 \ldots x(s_{i+1})])$. This requires a truncation of the back part of trajectory $\mathcal{L}$ that is higher than s_{i+1} (line 7), followed by the insertion of the horizontal line segment $\overline{qs_{i+1}}$ into the end of trajectory $\mathcal{L}$ (line 8). Specifically, the algorithm traces trajectory $\mathcal{L}$ from right to left for the leftmost point q such that $y(q) = y(s_{i+1})$. If no such q exists, the algorithm directly inserts s_{i+1} into the end of list $\mathcal{L}$. Otherwise, it deletes all the points of list $\mathcal{L}$ after position $x(q)$ and then inserts q and s_{i+1} into the end of list $\mathcal{L}$.
2. We next update the trajectory obtained in the previous step to the desired $GE(SM_{x(s_{i+1})}[x(s_{i+1}) - h + 1 \ldots x(s_{i+1})])$. This requires a simple truncation of the front part of trajectory $\mathcal{L}$ before position $x(s_{i+1}) - h + 1$ (line 9). Specif-

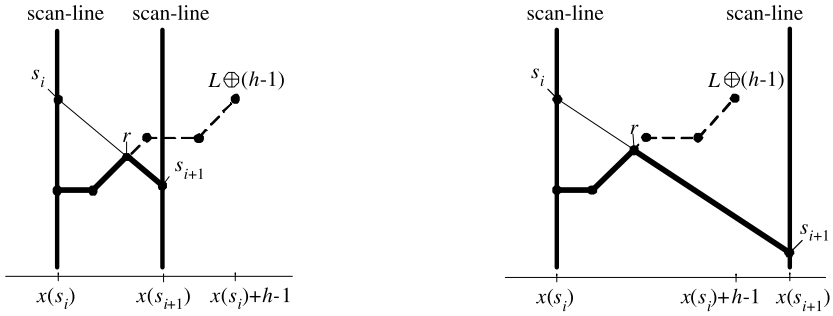

Fig. 8 Two subcases of Case 2. The *left side* of the figure shows the case where $x(s_{i+1}) \leq x(s_i) + h - 1$, and the *right side* of the figure shows the other case. As the scan-line moves from position $x(s_i)$ to position $x(s_{i+1})$, the algorithm traces the trajectory of the non-vertical bold lines

ically, the algorithm traces trajectory $\mathcal{L}$ from left to right for point p such that $x(p) = x(s_{i+1}) - h + 1$. It deletes all the points of list $\mathcal{L}$ before position $x(p)$ and inserts p into the front of list $\mathcal{L}$.

We proceed to examine Case 2 (lines 10–13) as follows. Observe that the slope of $\overline{s_i s_{i+1}}$ is non-positive in this case, for $\overline{s_i s_{i+1}}$ cannot be strictly above trajectory $\mathcal{L} \oplus (h-1)$. Let r be the leftmost intersection point of $\overline{s_i s_{i+1}}$ and trajectory $\mathcal{L} \oplus (h-1)$. We have that before position $x(r)$, trajectory $GE(S^{(h)})$ coincides with trajectory $\mathcal{L} \oplus (h-1)$, and after position $x(r)$, trajectory $GE(S^{(h)})$ coincides with line segment $\overline{r s_{i+1}}$. The algorithm thus outputs the lower part of $\overline{s_i s_{i+1}}$ and trajectory $\mathcal{L} \oplus (h-1)$ (lines 11–12). Note that it is possible that $r = s_i$ or $r = s_{i+1}$, in which cases the **output** operation removes the repeated point from the output sequence. To update list $\mathcal{L}$, the algorithm sets $\mathcal{L} = \langle s_{i+1} \oplus (-h+1), s_{i+1} \rangle$ (line 13), due to the fact that s_{i+1} becomes the lowest point in the window. Figure 8 provides an illustration of Case 2.

Finally, after the last point s_k is processed, the algorithm outputs the remaining points in list $\mathcal{L} \oplus (h-1)$ (line 16). This is because array S contains no element after position $x(s_k)$, and thus the sub-trajectory of $GE(S^{(h)})$ after position $x(s_k)$ is determined solely by trajectory $\mathcal{L} \oplus (h-1)$.

Theorem 3 *Algorithm* SLIDE *computes list* $GE(S^{(h)})$ *in* $O(|GE(S)|)$ *time.*

Proof Observe that if a point in list $\mathcal{L} \oplus (h-1)$ is output by lines 6 or 11, its corresponding point in list $\mathcal{L}$ is removed, at the same iteration, by lines 9 or 13 respectively. Moreover, the **truncate** operations in lines 7 and 9 spend time linear in the number of elements removed from list $\mathcal{L}$. Hence, we have that the time of the for-loop procedure is bounded by the total number of elements removed from list $\mathcal{L}$. Since there are $O(|GE(S)|)$ points inserted into list $\mathcal{L}$ throughout the procedure, the algorithm clearly runs in $O(|GE(S)|)$ time. □

4.3 Properties of the Output Trajectory

Recall that our algorithm for the edit distance problem is based on the propagation of turning points over the mn blocks. Each propagation over a block may cause a

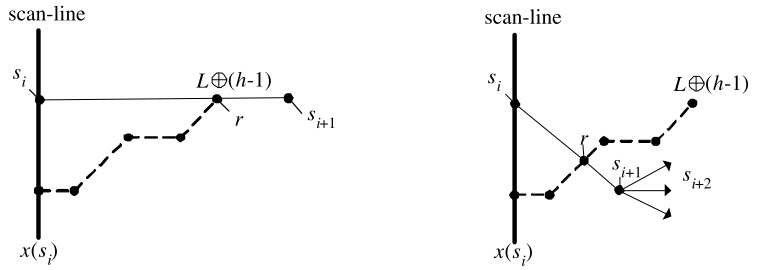

Fig. 9 The two subcases of Case 2 discussed in the proof of Lemma 3. The *left side* of the figure shows the case where $\overline{s_i s_{i+1}}$ is a *horizontal line* segment, and the *right side* of the figure shows the case where $\overline{s_i s_{i+1}}$ has a negative slope. We note that it is possible that $r = s_i$ or $r = s_{i+1}$, in which cases the repeated point will be removed by operation **output**

size expansion of turning points which we cannot afford. For example, consider the scenario in which each propagation doubles the number of turning points to the output border. The number of turning points will grow in an exponential manner as they cascade down. The following lemma is dedicated to an upper bound of the growth of turning points caused by Step 1 of PROPAGATE.

Lemma 3 $|GE(S^{(h)})| \leq |GE(S)| + |VP(S)| + 2$.

Proof We examine the total number of points output by algorithm SLIDE. Once point s_{i+1} is inserted into the end of list $\mathcal{L}$ by lines 8 or 13, it may contribute a point to the output in a later iteration by lines 6, 11, 12, or 15. Therefore, we say that point s_{i+1} is "deposited" in list $\mathcal{L}$. Note that the first point s_1 is not only output in line 1 but also deposited in $\mathcal{L}$ in line 2. Thus, s_1 contributes at most two points to the output. The remaining points s_{i+1} of list $GE(S)$ are divided into two cases. If s_{i+1} is a point of Case 1, s_{i+1} contributes at most one point to the output by being deposited in $\mathcal{L}$ in line 8. If s_{i+1} is a point of Case 2, we first observe that the contribution of the output r in line 12 can be ignored as follows. Let $\overline{uv}$ be the leftmost line segment of trajectory $\mathcal{L} \oplus (h-1)$ intersected by $\overline{s_i s_{i+1}}$. The output of r can be credited to point v, since v will be removed from list $\mathcal{L}$ at the end of this iteration and will never be output. Therefore, in Case 2, s_{i+1} contributes at most two points to the output by being output in line 12 and deposited in $\mathcal{L}$ in line 13. This gives us the first bound of $|GE(S^{(h)})| \leq \sigma_1 + 2\sigma_2 + 2 = |GE(S)| + \sigma_2 + 1$, where σ_1 and σ_2 denote the number of points of Case 1 and Case 2 respectively. To derive the claimed bound, we consider those σ_2 points of Case 2 in more details as follows. (See Fig. 9 for an illustration.)

1. Suppose that the slope of $\overline{s_i s_{i+1}}$ is zero. We have that r, s_{i+1}, and $s_{i+1} \oplus (h-1)$ lie in the same horizontal line. Hence, if the deposited s_{i+1} contributes a point to the output in a later iteration, the output point will cause the removal of s_{i+1} from the output.
2. Suppose that the slope of $\overline{s_i s_{i+1}}$ is negative. We further take the next point s_{i+2}, if any, into account. If the slope of $\overline{s_{i+1} s_{i+2}}$ is non-positive, we have that $\overline{s_{i+1} s_{i+2}}$ will immediately intersect with trajectory $\mathcal{L} \oplus (h-1)$ at point s_{i+1} in the next

iteration, since $\mathcal{L} \oplus (h-1)$ is set to $\langle s_{i+1}, s_{i+1} \oplus (h-1) \rangle$ at the end of the current iteration. Hence, in this case the deposited s_{i+1} in line 13 will be removed from list $\mathcal{L}$ before it can contribute any point to the output. Thus, we have that only when the slope of $\overline{s_{i+1}s_{i+2}}$ is positive (in which case, s_{i+1} is a valley point) or s_{i+2} does not exist (in which case, $s_{i+1} = s_k$), the deposited s_{i+1} may actually contribute one more point to the output.

Based on the above discussion, we conclude that a point of Case 2 contributes two points to the output only if it is a valley point or it is the last point s_k. Thus, we can refine the bound into $|GE(S^h)| \leq |GE(S)| + |VP(S)| + 2$. □

Below, we introduce two more properties of trajectory $GE(S^{(h)})$, which will be used in Sect. 5.

Lemma 4 *The trajectory of $GE(S^{(h)})$ contains no valley point, i.e. $|VP(S^{(h)})| = 0$.*

Proof Because the auxiliary trajectory $\mathcal{L}$ maintained by algorithm SLIDE never descends, the algorithm traces a negative-slope line only if a point of Case 2 is encountered. Observe that the algorithm cannot trace a positive-slope line right after a point of Case 2, because trajectory $\mathcal{L}$ is set to a horizontal line segment at the end of Case 2. Hence, we have that the complete trajectory of $GE(S^{(h)})$ contains no valley point. □

Lemma 5 *Except for horizontal line segments, the trajectory of $GE(S^{(h)})$ does not contain a line segment with a slope different from any of the line segments in the trajectory of $GE(S)$.*

Proof Note that at any time, algorithm SLIDE either traces trajectory $\mathcal{L} \oplus (h-1)$ or line segment $\overline{s_i s_{i+1}}$. In Case 1, the **truncate** operation in lines 7 and 9 does not introduce any new-slope line segment into trajectory $\mathcal{L}$, and the insertion of s_{i+1} in line 8 either appends $\overline{s_i s_{i+1}}$ or a horizontal line segment to trajectory $\mathcal{L}$. In Case 2, trajectory $\mathcal{L}$ is completely replaced by a horizontal line segment. Hence, we conclude that except for horizontal line segments, trajectory $\mathcal{L}$ never contains a line segment with a slope different from any of the line segments in trajectory $GE(S)$. The lemma thus follows. □

5 Time Complexity of the Edit Distance Problem

We now return to the edit distance problem of run-length encoded strings. We will show that our algorithm, based on the propagation of turning points, solves the edit distance problem in $O(mn^2)$ time. Recall that the algorithm runs in time linear in the total number of turning points in all the block borders. If the number of turning points grows to a multiple $c > 1$ of its original size after each propagation, the total time for the mn propagations will be exponential in m and n. Therefore, in what follows we aim to show that the number of turning points can only grow by a constant after each propagation. This obviously holds for the propagation over a match block, for it is a

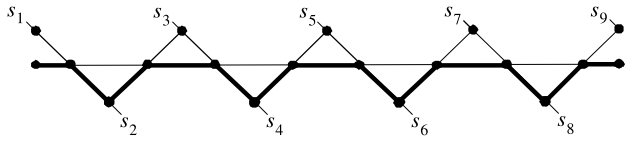

Fig. 10 An example where $GE(LEFT_2)$ and $GE(TOP_2)$ cross multiple times, which results in $|GE(OUT)| = \frac{3}{2}(|GE(LEFT_2)| - 1) + |GE(TOP_2)|$. In this example, $GE(LEFT_2)$ is a zigzag path, containing nine turning points, and $GE(TOP_2)$ is a line segment, containing two turning points. The lower part of the two trajectories, $GE(OUT)$, contains $\frac{3}{2} \times 8 + 2 = 14$ turning points, shown as the non-vertical bold lines

simple copy of turning points from the input border. Below, we show that this also holds for the propagation over a mismatch block.

We begin by examining the size expansion caused by Step 3 of procedure PROPAGATE. Recall that Step 3 outputs the lower part of the two trajectories $GE(LEFT_2)$ and $GE(TOP_2)$, which may cause a large size expansion, as shown in the example of Fig. 10. We rule out such extreme cases by proving that trajectories $GE(LEFT_2)$ and $GE(TOP_2)$ cross at most once. To this end, we introduce the *monotonicity property* of the border values of G, which was first observed in [1] and applied in [1, 9, 14, 22].

Lemma 6 [1, 22] *Given vertices* (x_4, y_4) *and* (x_3, y_3) *in* I, *and vertices* (x_1, y_1) *and* (x_2, y_2) *in* O, *where* $x_4 \le x_3 \le x_1 \le x_2$ *and* $y_3 \le y_4 \le y_2 \le y_1$, *if* $ED(x_3, y_3) + \text{dist}(x_3, y_3, x_1, y_1) \le ED(x_4, y_4) + \text{dist}(x_4, y_4, x_1, y_1)$, *then* $ED(x_3, y_3) + \text{dist}(x_3, y_3, x_2, y_2) \le ED(x_4, y_4) + \text{dist}(x_4, y_4, x_2, y_2)$.

Lemma 7 *The two trajectories obtained in Step* 2 *of* PROPAGATE, $GE(LEFT_2)$ *and* $GE(TOP_2)$, *cross at most once. That is, if* $TOP_2[i] \le LEFT_2[i]$ *for some* i, *then* $TOP_2[j] \le LEFT_2[j]$ *for all* $j \ge i$.

Proof Suppose that $OUT[i]$ and $OUT[j]$ correspond to vertex (x_1, y_1) and vertex (x_2, y_2) in G. From $TOP_2[i] \le LEFT_2[i]$, we know that there exists a vertex (x_3, y_3) in the top border such that $ED(x_3, y_3) + \text{dist}(x_3, y_3, x_1, y_1) \le ED(x_4, y_4) + \text{dist}(x_4, y_4, x_1, y_1)$ for all vertices (x_4, y_4), $y_4 \le y_2$, in the left border. By Lemma 6, we have that $ED(x_3, y_3) + \text{dist}(x_3, y_3, x_2, y_2) \le ED(x_4, y_4) + \text{dist}(x_4, y_4, x_2, y_2)$ for all vertices (x_4, y_4), $y_4 \le y_2$, in the left border. Since $LEFT_2[j] \le ED(x_3, y_3) + \text{dist}(x_3, y_3, x_2, y_2)$, the lemma thus follows. □

On the other hand, Step 1 of procedure PROPAGATE transforms two border trajectories into their sliding-window minima trajectories, which, by Lemma 4, causes a size expansion depending on the number of valley points in the input borders. Figure 11 shows an example in which the size expansion caused by such transformation can be up to one and a half of its original size. We rule out such extreme cases by showing that the number of valley points in any block border of G is in fact equal to zero. To this end, we introduce the property observed in [23], concerning the value difference between two adjacent vertices of G.

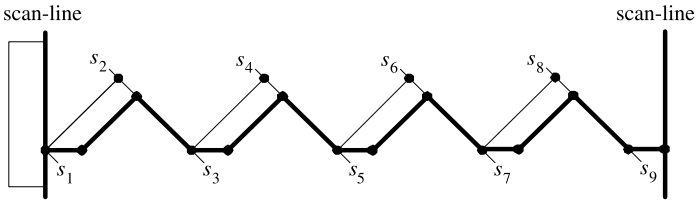

Fig. 11 An example where $|GE(S^{(h)})| = |GE(S)| + |VP(S)| + 2 = \frac{3}{2}|GE(S)| + \frac{1}{2}$. In this example, $GE(S)$ is a zigzag path, containing nine turning points, and the rectangle on the *left side* of the figure shows the sliding window, having a small width. There are three valley points s_3, s_5, and s_7 in $GE(S)$. The trajectory of $GE(S^{(h)})$ is shown as the non-vertical *bold lines*, which contains $9 + 3 + 2 = 14$ turning points

Lemma 8 [23] *The values of adjacent vertices in G never differ by more than one. That is, for each vertex* (i, j) *of* G, $ED(i, j) - ED(i-1, j), ED(i, j) - ED(i, j-1) \in \{-1, 0, +1\}$.

Lemma 9 *The line segments in trajectories* $GE(LEFT^{(h)})$ *and* $GE(TOP^{(h)})$ *are of slope* -1, 0, *or* $+1$.

Proof By Lemma 8, the line segments in trajectories $GE(LEFT)$ and $GE(TOP)$ are of slope -1, 0, or $+1$. The lemma follows immediately from Lemma 5. □

Lemma 10 *There is no valley point in any block border of G.*

Proof Clearly, the leftmost and topmost borders of G contain no valley point. By Lemma 1, the propagation over a match block produces no extra valley point in the output border. Below, we examine the propagation over a mismatch block, which is handled by Steps 1–3 of procedure PROPAGATE. By Lemma 4, Step 1 results in no valley point in $GE(LEFT^{(h)})$ and $GE(TOP^{(h)})$. Step 2 splits $GE(LEFT^{(h)})$ into two sub-trajectories $\mathcal{L}_1$ and $\mathcal{L}_2$, whose turning points are linearly adjusted based on (2). After the adjustment, the slope of each line segment in $\mathcal{L}_1$ is increased by one, while the slope in $\mathcal{L}_2$ remains the same. Hence, if there exists a valley point in the adjusted $\mathcal{L}_1$, we have that $GE(LEFT^{(h)})$ contains a line segment of slope -2, contradicting to Lemma 9. On the other hand, if there exists a valley point in the adjusted $\mathcal{L}_2$, we have that $GE(LEFT^{(h)})$ contains a valley point, contradicting to Lemma 4. We thus deduce that $GE(LEFT_2)$, the concatenation of the adjusted $\mathcal{L}_1$ and $\mathcal{L}_2$, has no valley point. We can similarly deduce that $GE(TOP_2)$ also has no valley point. Finally, $GE(OUT)$ obtained in Step 3 clearly contains no valley point, for it is the lower part of $GE(LEFT_2)$ and $GE(TOP_2)$. Thus, the propagation over a mismatch block also produces no extra valley point in the output border. The lemma thus follows by induction. □

Lemma 11 *After Steps* 1–3 *of* PROPAGATE, *the number of turning points can only grow by a constant, i.e.* $|GE(OUT)| \leq |GE(LEFT)| + |GE(TOP)| + c$, *where c is a constant.*

Proof By Lemma 10, we have that $|VP(LEFT)| = |VP(TOP)| = 0$. Combining with Lemma 3, we obtain that after Step 1, $|GE(LEFT^{(h)})| \leq |GE(LEFT)| + 2$ and $|GE(TOP^{(h)})| \leq |GE(TOP)| + 2$. Observe that in Step 2 only the splitting step may cause the number of turning points increased by two. Thus, we have that $|GE(LEFT_2)| \leq |GE(LEFT^{(h)})| + 2$ and $|GE(TOP_2)| \leq |GE(TOP^{(h)})| + 2$. By Lemma 7, merging the two trajectories in Step 3 leads to $|GE(OUT)| \leq |GE(LEFT_2)| + |GE(TOP_2)|$. Therefore, we conclude that $|GE(OUT)| \leq |GE(LEFT)| + |GE(TOP)| + 8$. □

Now, we are ready to prove the claimed time complexity of our algorithm.

Theorem 4 *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. Let $u_{i,j}$ and $v_{i,j}$ denote the number of turning points of the top border and the 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 2, the time of our algorithm is proportional to $\sum_{i=1}^{m} U_i + \sum_{j=1}^{n} V_j$. By Lemma 11, 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, $\sum_{j=1}^{n}(u_{i+1,j} + v_{i,j+1}) \leq \sum_{j=1}^{n}(u_{i,j} + v_{i,j}) + cn$, implying $\sum_{j=1}^{n} u_{i+1,j} \leq \sum_{j=1}^{n} u_{i,j} + (cn + 1)$. 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 $\sum_{i=1}^{m} U_i = O(m^2 n)$. We can similarly derive that $\sum_{j=1}^{n} V_j = O(mn^2)$, and the theorem follows. □

6 Concluding Remarks

In this paper, we present the first fully compressed algorithm for computing the edit distance between two RLE strings. Our algorithm runs in $O(mn^2)$ time, where m and n denote the number of runs of the strings. 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 easily trace, for each vertex in the output border, back to its optimal source vertex in the input border. The space used is $O(mn^2)$. Hirschberg's space reduction method [13] can be used to reduce the space to $O(mn)$ without impairing the time bound.

On the other hand, 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 alignment 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.

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 and space complexity of our algorithm. Moreover, we are interested in whether our result can be extended to a more general distance setting.

Acknowledgements Kuan-Yu Chen and Kun-Mao Chao were supported in part by NSC grants 97-2221-E-002-097-MY3 and 98-2221-E-002-081-MY3 from the National Science Council, Taiwan.

References

1. Aggarwal, A., Park, J.K.: Notes on searching in multidimensional monotone arrays. FOCS, pp. 497–512 (1988)
2. Amir, A., Benson, G.: Efficient two-dimensional compressed matching. DCC, pp. 279–288 (1992)
3. Amir, A., Landau, G.M., Sokol, D.: Inplace run-length 2d compressed search. Theor. Comput. Sci. **290**(3), 1361–1383 (2003)
4. Amir, A., Benson, G., Farach, M.: Let sleeping files lie: pattern-matching in Z-compressed files. J. Comput. Syst. Sci. **52**(2), 299–307 (1996)
5. Apostolico, A., Landau, G.M., Skiena, S.: Matching for run-length encoded strings. J. Complex. **15**(1), 4–16 (1999)
6. Arbell, O., Landau, G.M., Mitchell, J.S.B.: Edit distance of run-length encoded strings. Inf. Process. Lett. **83**(6), 307–314 (2002)
7. Bunke, H., Csirik, J.: An improved algorithm for computing the edit distance of run-length coded strings. Inf. Process. Lett. **54**(2), 93–96 (1995)
8. Chen, K.-Y., Hsu, P.-H., Chao, K.-M.: Hardness of comparing two run-length encoded strings. J. Complex. **26**(4), 364–374 (2010)
9. Crochemore, M., Landau, G.M., Ziv-Ukelson, M.: A subquadratic sequence alignment algorithm for unrestricted scoring matrices. SIAM J. Comput. **32**(6), 1654–1673 (2003)
10. Gasieniec, L., Rytter, W.: Almost optimal fully LZW-compressed pattern matching. DCC, pp. 316–325 (1999)
11. Gajewska, H., Tarjan, R.E.: Deques with heap order. Inf. Process. Lett. **22**(4), 197–200 (1986)
12. Hermelin, D., Landau, G.M., Landau, S., Weimann, O.: A unified algorithm for accelerating edit-distance computation via text-compression. STACS, pp. 529–540 (2009)
13. Hirschberg, D.S.: A linear space algorithm for computing maximal common subsequences. Commun. ACM **18**(6), 341–343 (1975)
14. Huang, G.-S., Liu, J.J., Wang, Y.-L.: Sequence alignment algorithms for run-length-encoded strings. COCOON, pp. 319–330 (2008)
15. Hirao, M., Shinohara, A., Takeda, M., Arikawa, S.: Fully compressed pattern matching algorithm for balanced straight-line programs. SPIRE, pp. 132–138 (2000)
16. Karpinski, M., Rytter, W., Shinohara, A.: An efficient pattern-matching algorithm for strings with short descriptions. Nord. J. Comput. **4**(2), 172–186 (1997)
17. Kim, J.W., Amir, A., Landau, G.M., Park, K.: Similarity between compressed strings. Encyclopedia of Algorithms, pp. 843–845 (2008)
18. 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. Inf. Process. Lett. **105**(1), 12–16 (2007)
19. Masek, W.J., Paterson, M.: A faster algorithm computing string edit distances. J. Comput. Syst. Sci. **20**(1), 18–31 (1980)
20. Mäkinen, V., Ukkonen, E., Navarro, G.: Approximate matching of run-length compressed strings. Algorithmica **35**(4), 347–369 (2003)
21. 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)
22. Schmidt, J.P.: All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. SIAM J. Comput. **27**(4), 972–992 (1998)
23. Ukkonen, E.: Finding approximate patterns in strings. J. Algorithms **6**(1), 132–137 (1985)
24. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. J. ACM **21**(1), 168–173 (1974)