RLE Edit Distance in Near Optimal Time

Raphaël Clifford ⓘD
Department of Computer Science, University of Bristol, UK
Raphael.Clifford@bristol.ac.uk

Paweł Gawrychowski ⓘD
Institute of Computer Science, University of Wrocław, Poland
gawry@cs.uni.wroc.pl

Tomasz Kociumaka ⓘD
Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
Institute of Informatics, University of Warsaw, Poland
kociumaka@mimuw.edu.pl

Daniel P. Martin[1] ⓘD
The Alan Turing Institute, British Library, London, UK
dmartin@turing.ac.uk

Przemysław Uznański ⓘD
Institute of Computer Science, University of Wrocław, Poland
puznanski@cs.uni.wroc.pl

──────── **Abstract** ────────

We show that the edit distance between two run-length encoded strings of compressed lengths m and n respectively, can be computed in $\mathcal{O}(mn\log(mn))$ time. This improves the previous record by a factor of $\mathcal{O}(n/\log(mn))$. The running time of our algorithm is within subpolynomial factors of being optimal, subject to the standard SETH-hardness assumption. This effectively closes a line of algorithmic research first started in 1993.

1 Introduction

The edit distance is one of the most common distance measures between strings. For two strings of length M and N respectively, the edit distance counts the minimum number of single character insertions, deletions and substitutions needed to transform one string into the other. The first record of an $\mathcal{O}(MN)$ algorithm to compute the edit distance is from 1968 [14] although it was rediscovered independently a number of times subsequently. Masek and Paterson improved the running time to $\mathcal{O}(MN/\log N)$, for $N \geq M$, in 1980 and this is the fastest known algorithm to date [12]. Much more recently it has been shown that no $\mathcal{O}(MN^{1-\epsilon})$ time edit distance algorithm can exist, subject to the strong exponential time hypothesis (SETH) [4,5]. As a result, it is likely that little further progress can be made in terms of improving its worst case complexity.

---

[1] Research carried out while the author was a member of the University of Bristol and the Heilbronn Institute for Mathematical Research.

44th International Symposium on Mathematical Foundations of Computer Science (MFCS 2019).
Editors: Peter Rossmanith, Pinar Heggernes, and Joost-Pieter Katoen; Article No. 66; pp. 66:1–66:13
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper we focus on the problem of computing the edit distance between two compressed strings. The run-length encoding (RLE) of a string compresses consecutive identical symbols into a run, denoted σ^i if the symbol σ is repeated i times. For example `aaabbbbaaa` would be compressed to $\mathtt{a}^3\mathtt{b}^4\mathtt{a}^3$. This form of compression is commonly used for image compression but also has wider applications including, for example, in image processing [9, 15] and succinct data structures [11].

In 1993 Bunke and Csirik proposed the first algorithm for computing the edit distance between RLE strings. For two strings of RLE-compressed lengths m and n respectively, their algorithm runs in $\mathcal{O}(mn)$ time in the special case where all the runs are of the same length [6]. However the running time falls back to the naive complexity of $\mathcal{O}(MN)$ time in the worst case where M and N are the uncompressed lengths of the two strings. This worst case complexity was subsequently improved to $\mathcal{O}(Nm + Mn)$ [3, 7] and then $\mathcal{O}(\min\{Nm, Mn\})$ time in 2007 [10]. Finally in 2013 the fastest solution prior to this current work was given running in $\mathcal{O}(mn^2)$ time, where $n \geq m$ [8]. This was the also the first algorithm for the RLE edit distance problem whose running time did not depend on the uncompressed lengths of the input strings.

For uncompressed strings, the longest common subsequence (LCS) problem has long been considered a close relative of the edit distance problem. This is partly due to the similarity of their dynamic programming solutions and partly because LCS is a special case of edit distance when general costs are allowed for the different mismatch and substitution operations. Moreover, the two problems have the same quadratic time upper bounds and SETH-hardness lower bounds [5]. Somewhat surprisingly, however, the history of algorithms for LCS and edit distance have *not* mirrored each other when the problems are considered on RLE strings. In particular, an $\mathcal{O}(mn \log(mn))$ time algorithm for computing the LCS on RLE strings was given in 1999 [2] which is considerably faster than has been possible up to this point for the edit distance problem. Some work has also been carried out since that date to improve the log factor in the running time complexity for the LCS problem [1, 13].

In this paper we speed up the running time for the edit distance problem on RLE strings by a factor of $\mathcal{O}(n/\log(mn))$, matching the fastest LCS algorithm to within a logarithmic factor and making it within subpolynomial factors of being optimal, assuming SETH holds. As a result, our new algorithm shows that the LCS and edit distance problems are indeed of essentially the same complexity even when the input strings are run-length encoded. This effectively closes a line of algorithmic research first started in 1993.

▶ **Theorem 1.** *Given two RLE strings of compressed length n and m respectively, there exists an algorithm to compute their edit distance which runs in $\mathcal{O}(mn \log(mn))$ time.*

2   Previous Work and Preliminaries

The classic dynamic programming solution for computing the edit distance between uncompressed strings X and Y of uncompressed lengths M and N respectively, computes the distance between all prefixes $X[1, \ldots, i]$ and $Y[1, \ldots, j]$. The key recurrence which enables us to do this efficiently is given by:

$$\mathsf{ED}(i, j) = \min(\mathsf{ED}(i - 1, j - 1) + \delta(X_i \neq Y_j), \mathsf{ED}(i - 1, j) + 1, \mathsf{ED}(i, j - 1) + 1).$$

From this the classic $\mathcal{O}(MN)$ time solution follows directly.

The previous approaches for the edit distance problem on RLE strings take this recurrence and the implied dynamic programming table as their starting point. The basic idea was introduced by Bunke and Csirk [6] whose algorithm works by dividing the dynamic programming table into "blocks", where each block is defined by a run in the original strings.

For each block the central task is to compute its bottom row and rightmost column given the bottom row of the block above and the rightmost column of the block to the left. For simplicity of terminology, we will refer to the rightmost column of the block to the left and the bottom row of the block above collectively as the *input border* of a block and the bottom row and rightmost column of a block as its *output border*. Figure 1 illustrates an example.

In [3, 7] it was shown that the work needed to derive the values of all the output borders of blocks is at most linear in their length. When computing the edit distance between strings X and Y, the length of each row in the dynamic programming table is the uncompressed length of string Y and the length of each column is the uncompressed length of X. If there are m runs in string X and n runs in Y then the total time complexity for computing the edit distance using their approach is therefore $\mathcal{O}(Nm + Mn)$.

The work closest to ours is the $\mathcal{O}(mn^2)$ algorithm of Chen and Chao [8]. They observe that the borders of the blocks in the dynamic programming table are piecewise linear with gradient ± 1 or 0. The borders can be therefore concisely represented by specifying their starting values as well as the positions and types of the points of changing gradient called the *turning points*. They prove that for a given block the number of turning points in an output border is at most a constant greater than in its input border. Consequently, a simple calculation shows that the total number of turning points is $\mathcal{O}(mn^2)$ (for $m \leq n$). Chen and Chao arrive at their final complexity by designing a procedure that computes the representation of the output border of a block, given the representation of its input border, in time proportional to the number of turning points. We now summarise their approach using our own notation.

There are two distinct types of blocks in the dynamic programming table. A match block corresponds to a rectangle where the corresponding symbols in the two strings match. A mismatch block corresponds to a rectangle where the corresponding symbols mismatch. Figure 1 shows both match and mismatch blocks. Borrowing notation from [8] we say that element (i, j) of the dynamic programming table ED is in *diagonal* $j - i$. Let (i_d, j_d) be the intersection of the input border with diagonal $j - i$.

▶ **Lemma 2** ([8, Lemma 1]). *For a match block,* $ED(i, j) = ED(i_d, j_d)$.

Lemma 2 indicates that for a match block we can simply copy the values from the corresponding position in the input border to derive the values of the output border. The main challenge is therefore how to handle mismatch blocks.
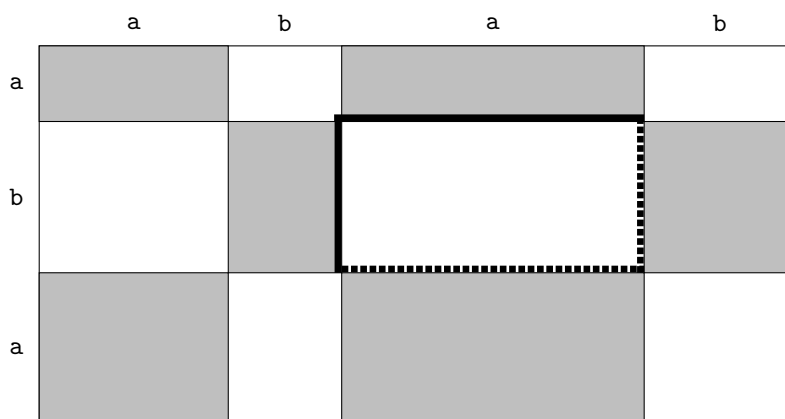
▪ **Figure 1** Match blocks are in grey. The thick line is for an input border and the dotted line an output border. The input border is contained entirely inside the neighbouring match blocks.

For mismatch blocks Chen and Chao's algorithm, in a similar manner to previous RLE edit distance algorithms, splits the problem into two parts corresponding to shortest paths that pass through the leftmost column or the top row [3, 6–8]. Consider a mismatch block of height h and width w corresponding to runs $\mathtt{a}^{h-1}$ and $\mathtt{b}^{w-1}$ such that $h \leq w$ (the other case can be processed similarly by swapping the left and the top part of the input border). $\text{LEFT}[1, h]$ and $\text{TOP}[1, w]$ denotes the values of the left and the top part of the input border, numbered in a bottom-to-top and a left-to-right direction, respectively. For an array $S[1, n]$ and a parameter $h \in \mathbb{Z}^+$, let $S^{(h)}[i] = \min\{S[j] \mid i - h + 1 \leq j \leq i, 1 \leq j \leq n\}$. Chen and Chao separately compute all the output border values that are derived from a value in the left part of the input border, denoted $\text{OUT}_{\text{LEFT}}[1, w + h - 1]$, and similarly compute all the output border values that are derived from a value in the top part of the input border, denoted $\text{OUT}_{\text{TOP}}[1, w + h - 1]$, as follows.

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

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

We start with reformulating the algorithmic framework of Chen and Chao using the following notation.

▶ **Definition 3.** *Let $S[1, n]$ be a 1-indexed array of length n.*
- *For a parameter $h \in \mathbb{Z}^+$, **SWM**(S, h) (Sliding Window Minima) returns the array $S^{(h)}$ of length $n + h - 1$.*
- ***split**(S, m) returns the two subarrays $S[1, m]$, and $S[m + 1, n]$.*
- *$S \pm \overrightarrow{1}$ returns S with the gradient decreased/increased by one, or formally $S'[i] = S[i] \pm i$.*
- *For an integer constant c, $S + c$ returns S with every value increased by c.*
- ***initialise**(ℓ) returns an array of length ℓ initially filled with zeroes.*
- ***join**(S_1, S_2) simply concatenates two arrays.*

Now, equations (1) and (2) can be rephrased as Algorithms 1 and 2, respectively. The final step of the algorithm is to compute the output border as the minimum of $\text{OUT}_{\text{TOP}}[i]$ and $\text{OUT}_{\text{LEFT}}[i]$ for each index i. This is performed in linear time per block by Chen and Chao [8]. In Section 3 we will design a new implementation of both algorithms and a subtle amortised argument for this final step. The latter is based on the fundamental property of the values in an output border summarised by Lemma 4.

▶ **Lemma 4** ([8, Lemma 7]). *If there exists an i such that $\text{OUT}_{\text{TOP}}[i] \leq \text{OUT}_{\text{LEFT}}[i]$, then $\text{OUT}_{\text{TOP}}[j] \leq \text{OUT}_{\text{LEFT}}[j]$ for all $j \geq i$.*

■ **Algorithm 1** Compute the shortest path passing through the left border.

---

1 $S \leftarrow \mathsf{SWM}_h(\text{LEFT}[1, h], h)$;
2 $S^\ell, S^r \leftarrow \mathsf{split}(S, h)$;
3 $S_1 \leftarrow S^\ell + \overrightarrow{1} - 1$;
4 $S_2 \leftarrow \mathsf{initialise}(w - h) + \overrightarrow{1} + S[h] + h - 1$;
5 $S_3 \leftarrow S^r + w - 1$;
6 $\text{OUT}_{\text{LEFT}} \leftarrow \mathsf{join}(\mathsf{join}(S_1, S_2), S_3)$;
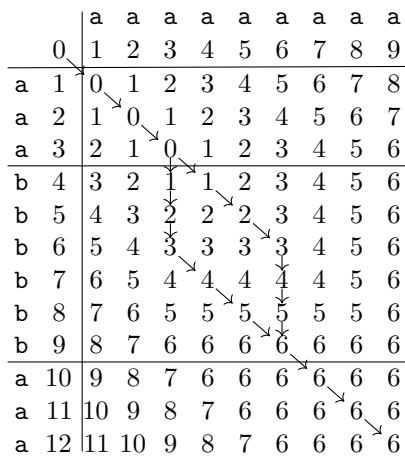
> **Algorithm 2** Compute the shortest path passing through the top border.

---
1 $S \leftarrow \mathsf{SWM}_h(\mathrm{TOP}[1, w], h);$
2 $S^\ell, S^r \leftarrow \mathsf{split}(S, w);$
3 $S_1 \leftarrow S^\ell + h - 1;$
4 $S_2 \leftarrow S^r - \overrightarrow{1} + w + h - 1;$
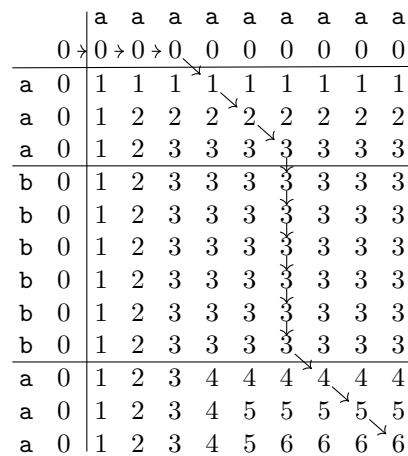**5** $\mathrm{OUT}_{\mathrm{TOP}} \leftarrow \mathsf{join}(S_1, S_2);$

Before we go on to explain how we speed up the task of deriving the borders of blocks, it is worth exploring for a moment why we cannot simply apply, perhaps with some small modifications, the known $\mathcal{O}(mn \log(mn))$ time solution for LCS on RLE strings [2]. The key obstacle comes from the different nature of optimal paths in the dynamic programming table of the LCS and edit distance problems.

For the LCS problem on RLE strings Apostolico et al. [2] introduced two important concepts. The first is *forced paths* and the second *corner paths*. They say that a path beginning at the upper-left corner of a match block is *forced* if it traverses the block by strictly diagonal moves and, whenever the right (respectively, lower) side of an intermediate match block is reached, proceeds to the next match block by a straight horizontal (respectively, vertical) line through the mismatch blocks in between. A *corner* path is one that enters match blocks in the top left corner and exits only through the bottom right corner. They show that there is always a longest common subsequence between two strings corresponding to the concatenation of subpaths of corner and forced paths. This fact greatly reduces the number of different paths that have to be considered and hence the complexity of solving the overall LCS problem. However for the edit distance problem this property of forced paths no longer holds. Figures 2 and 3 show an example of this key difference between optimal paths under edit distance and LCS. In Figure 2 we can see that there is no optimal vertical (or horizontal) path through the mismatch block. By contrast, there is indeed an optimal vertical path for the LCS problem as illustrated by Figure 3.

In order to speed up edit distance computation on RLE strings we introduce a new data structure for input borders and output borders. This will allow us to derive the values of output borders from their respective input borders in amortised logarithmic time per border, rather than the previous linear time. The rest of the paper is devoted to this task.

		a	a	a	a	a	a	a	a	a
	0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7	8
a	2	1	0	1	2	3	4	5	6	7
a	3	2	1	0	1	2	3	4	5	6
b	4	3	2	1	1	2	3	4	5	6
b	5	4	3	2	2	2	3	4	5	6
b	6	5	4	3	3	3	3	4	5	6
b	7	6	5	4	4	4	4	4	5	6
b	8	7	6	5	5	5	5	5	5	6
b	9	8	7	6	6	6	6	6	6	6
a	10	9	8	7	6	6	6	6	6	6
a	11	10	9	8	7	6	6	6	6	6
a	12	11	10	9	8	7	6	6	6	6

> **Figure 2** Edit distance with forced turn in mismatch block.

		a	a	a	a	a	a	a	a	a
	0	0	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1	1	1
a	0	1	2	2	2	2	2	2	2	2
a	0	1	2	3	3	3	3	3	3	3
b	0	1	2	3	3	3	3	3	3	3
b	0	1	2	3	3	3	3	3	3	3
b	0	1	2	3	3	3	3	3	3	3
b	0	1	2	3	3	3	3	3	3	3
b	0	1	2	3	3	3	3	3	3	3
b	0	1	2	3	3	3	3	3	3	3
a	0	1	2	3	4	4	4	4	4	4
a	0	1	2	3	4	5	5	5	5	5
a	0	1	2	3	4	5	6	6	6	6

> **Figure 3** LCS with optimal vertical path through mismatch block.

3   Efficient Manipulation of Piecewise-Linear Functions

In this section, we describe the data structure we will use to represent input borders and output borders in the dynamic programming table. We will then show how the operations from Definition 3 can be implemented efficiently using this data structure.

Recall that *a piecewise linear function* is a real-valued function F whose domain $\mathrm{dom}(F)$ is a closed interval that can be represented as a union of closed intervals $\mathrm{dom}(F) = \bigcup_{j=1}^{k} I_j$ such that F restricted to I_j is an affine function ($g_j x + h_j$ for some coefficients g_j and h_j). The input and output borders as defined in Section 2 are by definition piecewise linear.

In this section, we impose a few further restrictions:
- For each integer $x \in \mathrm{dom}(F)$, the value $F(x)$ is also an integer.
- The gradient g_j of each $F|_{I_j}$ is -1, 0, or 1.
- The endpoints of $\mathrm{dom}(F)$ are integers or half-integers.

The graph of a piecewise linear function F is a simple polygonal curve, and thus it can be interpreted as a sequence of *turning points* connected by straight-line *segments*. Due to the restrictions imposed on F, each turning point has integer or half-integer coordinates. We represent such a function F as a sequence of segments stored in an annotated balanced binary search tree, where each segment explicitly keeps the coordinates of its endpoints.[2]

We first provide a simple implementation of curves supporting a few basic operations, and then we gradually augment it to handle more complicated operations. We conclude with an amortised running time analysis.

3.1   Basic Operations

Our first implementation just stores the corresponding segment for each node v:

(x_ℓ, y_ℓ)**:** The coordinates of the left endpoint of the segment corresponding to v.
(x_r, y_r)**:** The coordinates of the right endpoint of the segment corresponding to v.

Nevertheless, we are already able to implement some operations useful in Algorithms 1 and 2.

Create.   The create operation produces a function F whose graph consists of just one segment S with given endpoints (x_ℓ, y_ℓ) and (x_r, y_r). This enables us to implement the whole of line 4 of Algorithm 1 in worst-case constant time.

Join.   The join operation takes two functions, F_L and F_R with domains $\mathrm{dom}(F_L) = [x_L, x_M]$ and $\mathrm{dom}(F_R) = [x_M, x_R]$, respectively, and with a common endpoint $F_L(x_M) = F_R(x_M)$. It returns a function F with $\mathrm{dom}(F) = [x_L, x_R]$ such that $F_L = F|_{[x_L, x_M]}$ and $F_R = F|_{[x_M, x_R]}$. To implement this operation, we first join the two balanced binary search trees. If the rightmost segment of F_L has the same gradient as the leftmost segment of F_R, we also join these segments. The resulting tree represents F. The worst-case running time is logarithmic.

Split.   The split operation takes a function F with $\mathrm{dom}(F) = [x_L, x_R]$ and a value $x_M \in \mathrm{dom}(F)$. It returns two functions $F_L = F|_{[x_L, x_M]}$ and $F_R = F|_{[x_M, x_R]}$. To implement it, we first descend the binary search tree to find a segment S with $x_M \in \mathrm{dom}(S)$. If x_M lies in the interior of $\mathrm{dom}(S)$, we split this segment into two. Next, we split the binary search tree to separate the segments to the left of x_M from the segments to the right of x_M. The resulting two trees represent F_L and F_R, respectively. The worst-case running time is logarithmic.

---

[2] Note that the coordinates of each internal turning point are stored with both incident segments.

Combine.   The combine operation takes two functions F_1 and F_2 over the same domain $\mathrm{dom}(F_1) = \mathrm{dom}(F_2) = [x_L, x_R]$ and returns their pointwise minimum: a function F with $\mathrm{dom}(F) = [x_L, x_R]$ such that $F(x) = \min(F_1(x), F_2(x))$ for $x \in \mathrm{dom}(F)$. We assume that there exists $x_M \in [x_L, x_R]$ such that $F_1(x) > F_2(x)$ if $x < x_M$ and $F_1(x) \le F_2(x)$ if $x \ge x_M$.

If $F_1(x_L) \le F_2(x_L)$, then $x_M = x_L$. Hence, we return $F = F_1$ and discard F_2. Similarly, if $F_1(x_R) > F_2(x_R)$, then $x_M = x_R$. Hence, we return $F = F_2$ and discard F_1.

Otherwise, we are guaranteed that $F_1(x_M) = F_2(x_M)$. Our first task is to find x_M. For this, we locate segments S_1 of F_1 and S_2 of F_2 such that $x_M \in \mathrm{dom}(S_1) \cap \mathrm{dom}(S_2)$.

We observe that S_1 corresponds to the leftmost node v in the BST of F_1 such that $v.y_r = F_1(v.x_r) \le F_2(v.x_r)$. Hence, we perform a left-to-right in-order traversal of the BST to find S_1. For each visited node v, we evaluate $F_2(v.x_r)$ by descending the BST of F_2 to find a segment whose domain contains $v.x_r$. Symmetrically, S_2 corresponds to the rightmost node v in the BST of F_2 such that $F_1(v.x_\ell) > F_2(v.x_\ell) = v.y_\ell$, so we find S_2 by performing a right-to-left in-order traversal of the BST.

Next, we note that $(x_M, F_1(x_M)) = (x_M, F_2(x_M))$ is the leftmost common point of S_1 and S_2. Hence, we can now compute x_M easily (the restrictions on F_1 and F_2 guarantee that it is an integer or a half-integer). Finally, we split both F_1 and F_2 at x_M, discard $F_1|_{[x_L, x_M]}$ and $F_2|_{[x_M, x_R]}$, join $F_2|_{[x_L, x_M]}$ with $F_1|_{[x_M, x_R]}$, and return the resulting function as F.

As far as the running time is concerned, the cost is logarithmic for each discarded segment. We can now also implement the final combine step that produces our representation of the output border from the outputs of Algorithms 1 and 2 by finding the minimum at each index.

3.2   Shifts

Next, we extend our data structure to implement the shift operation which moves the whole function by a given vector. It is useful in Algorithms 1 and 2 for altering S^ℓ and S^r.

Formally, given a function F with $\mathrm{dom}(F) = [x_L, x_R]$ and a vector (Δ_x, Δ_y), we transform F into F' such that $F'(x) = F(x - \Delta_x) + \Delta_y$ for each $x \in \mathrm{dom}(F') = [x_L + \Delta_x, x_R + \Delta_x]$.

This update is performed using a technique known as *lazy propagation*. We augment each node v with the following extra field:

(δ_x, δ_y): A deferred shift to be propagated within the subtree of v.

This change is then lazily propagated as further operations are executed. Here, we rely on a key structural property of BST operations:

▶ **Observation 5.** *The execution of every BST operation can be extended (at the cost of an extra multiplicative constant in the running time) with a sequence of node* activations *and* deactivations *such that:*
- *a node v is accessed only when it is active and has no active descendant,*
- *when v is active, then all its ancestors are active,*
- *no node is active at the beginning and the end of the execution.*

The idea behind lazy propagation is that the deferred updates stored at a node v are propagated when v is activated. This way, every active node has no delayed updates pending. Hence, from the perspective of any other operation, the effect is the same as if we have meticulously modified every node for each update.

The shift propagation is very simple: when a node v receives a request for a shift by (Δ_x, Δ_y), then we just add (Δ_x, Δ_y) to the delayed shift $(v.\delta_x, v.\delta_y)$ stored at v. Upon activation of v, we propagate $(v.\delta_x, v.\delta_y)$ to the children of v, add $(v.\delta_x, v.\delta_y)$ to both (x_ℓ, y_ℓ)

and (x_r, y_r), and reset $(v.\delta_x, v.\delta_y) := (0, 0)$. To implement the *shift* operation, we just send a request for a shift by (Δ_x, Δ_y) to the root node r.

The worst-case running time of a shift is constant, and the extra cost of propagation does not increase the asymptotic running time of the remaining operations.

3.3   Gradient Changes

The gradient change operation takes a function F and a coefficient Δ_g, and it transforms F into F' such that $F'(x) = F(x) + \Delta_g \cdot x$ for each $x \in \text{dom}(F') = \text{dom}(F)$. This operation is needed in both Algorithms 1 and 2 to transform S^r and S^ℓ, respectively.

We first note that the constraints imposed on the gradients of functions F and F' yield that $\Delta_g = -1$, F is non-decreasing, and F' is non-increasing, or $\Delta_g = 1$, F is non-increasing, and F' is non-decreasing. However, these limitations only become relevant in Section 3.4.

To implement gradient change, we just add another field to each node v:

δ_g: A deferred gradient change to be propagated within the subtree of v.

We now have two types of lazily propagated updates: shift and gradient change. These two operations do *not* commute, so we need to decide how to interpret the two kinds of deferred updates stored at a node v. We shall assume that the gradient change by δ_g is to be performed *before* the shift by (δ_x, δ_y).

Thus, while shift propagation is implemented as in Section 3.2, adding Δ_g to $v.\delta_g$ is insufficient when a node v receives a request to change gradient by Δ_g: we also need to add $\Delta_g \cdot \delta_x$ to δ_y. This approach is correct since a shift by (Δ_x, Δ_y) followed by a gradient change by Δ_g is equivalent to a gradient change by Δ_g followed by a shift by $(\Delta_x, \Delta_y + \Delta_g \cdot \Delta_x)$.

Upon activation of v, we first apply the deferred gradient change: we propagate it to the children of v, increase $v.y_\ell$ by $v.\delta_g \cdot v.x_\ell$ and $v.y_r$ by $v.\delta_g \cdot v.x_r$, and reset $v.\delta_g = 0$. Then, we handle the deferred shift as in Section 3.2.

Finally, we note that to implement the *gradient change* operation, we just send a request for a gradient change by Δ_g to the root node r. The worst-case running time is constant.

3.4   Sliding Window Minima

We can finally show how to implement the SWM operation efficiently on our data structure. This is the most involved of the operations we will need. The SWM operation given a function F with $\text{dom}(F) = [x_L, x_R]$ and a *window width* t, returns a function F' with $\text{dom}(F') = [x_L, x_R + t]$ such that $F'(x) = \min\{F(x') : x' \in [x - t, x] \cap \text{dom}(F)\}$.

Combinatorial Properties

We begin by observing that the SWM operation is composable.

▶ **Observation 6.** *Every function F and positive window widths t, t' satisfy*
$SWM(SWM(F, t), t') = SWM(F, t + t')$.

Hence, instead of applying $SWM(\cdot, t)$ for an integer width t, we may equivalently apply the $SWM(\cdot, 1)$ operation t times. The key property of width 1 is that the changes to the transformed function are very local. The structure of these modifications can be described in terms of *types* of turning points. We classify internal turning points by the gradients (**I**ncreasing, **F**lat, or **D**ecreasing) of the incident segments; see Table 1, where we also analyse how a function changes in the vicinity of each turning point subject to $SWM(\cdot, 1)$.

Table 1 Types of internal turning points and their behaviour subject to the SWM operation.

Type DI	Type ID	Type IF	Type FD	Type FI	Type DF

- A point of type FD or DF remains intact.
- A point of type FI or IF is shifted by $(1,0)$.
- A point of type ID is shifted by $(0.5, -0.5)$.
- A point of type DI transformed into a point of type DF and a point of type FI, and the latter is shifted by $(1,0)$.

Note that the behaviour of DI points is unlike that of other types. However, this discrepancy disappears if we replace every DI point with two coinciding points of types DF and FI, respectively, with an artificial length-0 segment in between. Hence, whenever a new internal turning point is created (which happens only within the join operation), if the turning point would be of type DI, we pre-emptively replace it by two coinciding points of type DF and FI, respectively. Note that the resulting length-0 segment never changes its gradient since gradient change is allowed only on a monotone function. However, when an incident segment is modified, we may need to remove the length-0 segment. This process cannot cascade, though, causing another length zero segment to be removed.

Next, we analyse in Table 2 how the SWM$(F, 1)$ operation affects the endpoints of the graph of F. In most cases, the left endpoint stays intact and the right endpoint is shifted by $(0, 1)$. The only exceptions are endpoints of type -I and D-, which exhibit similar behaviour to the internal turning points of type DI. Moreover, this discrepancy also disappears if we introduce artificial flat segments of length 0. Hence, we replace a point of type -I with two points of type -F and FI, respectively, and a point of type D- with two points of type DF and F-, respectively. However, this time the replacement is *not* pre-emptive: we perform it as the first step in the implementation of the SWM operation. This is possible because there are just two endpoints, while the number of internal turning points of type DI could be large. Our gain, on the other hand, is that we avoid length-0 segments changing their gradients.

With the artificial length-0 segments in place, it is now true that the effect of the SWM operation on each turning point can be described as a shift depending only on the type of the point. As a result of these shifts, some segments may disappear as their length reaches 0; in this case, we say that a segment *collapses*. Only segments of three kinds may collapse:

- a segment between a point of type ID and point of type DF;
- a segment between a point of type IF and point of type FD;
- a segment between a point of type FI and point of type ID;

Table 2 Types of endpoints and their behaviour subject to the SWM operation.

Type -I	Type -F	Type -D	Type I-	Type F-	Type D-

When a segment collapses, it is removed and the two incident turning points are merged.[3]
Each segment of the three affected kinds has the *collapse time*, defined as the smallest t for
which $\mathsf{SWM}(\cdot, t)$ makes it collapse (assuming no interaction from incident segments) equal
to the Manhattan distance between its endpoints. Note that due to the restrictions on the
piecewise linear functions considered in this section, the collapse time is always an integer.

Implementation

To implement the SWM operation, we augment each node v with the following fields:

$\mathbf{type}_\ell, \mathbf{type}_r$: The types of the turning points joined by the segment corresponding to v.
$\boldsymbol{\delta_t}$: The amount of a deferred SWM to be propagated within the subtree of v.
$\boldsymbol{t_{\min}}$: The minimum collapse time among the segments in the subtree of v.

Note that the type of each internal turning point is stored twice. Hence, whenever a node
type changes, this fact needs to be reflected at both incident segments (and we need to reach
the corresponding nodes by descending the BST; shortcuts would violate Observation 5).
The field $v.t_{\min}$ is of a kind we have not encountered yet: its value depends on the
corresponding values for the children of v and on other fields at v. It is brought up to date
whenever v is deactivated (so that it can be accessed only when v is inactive). We shall
assume that its value already reflects the deferred updates stored at v. The procedure of
recomputing $v.t_{\min}$ is simple: we determine the collapse time of the segment represented by
v (which is infinite or equal to $|v.x_r - v.x_\ell| + |v.y_r - v.y_\ell|$ depending on the types of the
incident turning points), and take the minimum of this value and $u.t_{\min}$ for every child u of
v. Since v has no deferred changes when it is deactivated, the resulting minimum is $v.t_{\min}$.

Propagation.   The main structural modification to the lazy propagation procedures is that
we maintain an additional invariant that no deferred changes are stored on the leftmost and
on the rightmost path of the BSTs representing every function F. To maintain this invariant,
immediately after lazily updating of the whole F (sending a request to the root node r), we
descend to the leftmost and to the rightmost segment F; this increases the cost of shift and
gradient change to logarithmic. Note that the split operation must anyway visit the nodes
representing the new boundary segments (to update the types of new endpoints). Moreover,
if a path from the root to a given node v contains no deferred updates, then this is still true
after any rebalancing of the BST (as only active nodes get rotated).
Concerning the lazy SWM propagation, we explicitly forbid requesting for SWM with
window width exceeding $r.t_{\min}$, because collapsed segments need to be removed before we
proceed further. Also, the window widths (and hence the values δ_t) are always non-negative.
We have three kinds of deferred updates now: SWM, gradient change, and shift. We fix
the semantics of the fields δ_t, δ_g, and (δ_x, δ_y) so that an SWM of width δ_t is performed first,
a gradient change by δ_g second, and a shift by (δ_x, δ_y) last. The requests for a shift and for
a gradient change are still implemented as in Section 3.3; note that these updates do not
affect the collapse times (the three segment kinds with finite collapse times cannot appear
in monotone functions). On the other hand, the request for an SWM with a window width
Δ_t requires more care. We clearly need to increase δ_t by Δ_t and decrease $t_{\min}$ by the same
amount. The aforementioned steps suffice if $\delta_g = 0$. Otherwise, we note that the turning

---

[3] Two adjacent segments may collapse simultaneously. In that special case, three subsequent points, of
type FI, ID, and DF, respectively, need to be deleted.

points in the subtree of v are all of types DF and FD or all of types IF and FI. (Observe that there are no deferred changes in the proper ancestors of v and that v is not on the leftmost or rightmost path in this case.) We can easily distinguish the two cases by analysing the endpoints of the segment corresponding to v. Moreover, the SWM operation is void in the first of these cases, and in the second one it reduces to a shift by $(\Delta_t, 0)$. Hence, we shall implement it this way rather than by modifying $v.\delta_t$.

The propagation itself is relatively easy: upon activation of a node v, we first propagate the SWM operation to the children of v, update the endpoints of the segment corresponding to v (according to Tables 1 and 2, with the shift multiplied by δ_t), and finally reset $\delta_t = 0$. Then, we propagate the gradient change and the shift. This is implemented as in Section 3.3 except that the gradient change now affects not only the coordinates but also the types of the segment's endpoints.

SWM Procedure.   To implement the SWM procedure itself, we first check the endpoint types and perform appropriate substitutions for endpoints of type -I and D-. Next, we would like to lazily apply the SWM operation with window width t to the root r. However, this could result in negative collapse time $r.t_{\min}$, so instead we perform SWM gradually based on Observation 6. If $r.t_{\min} < t$, we make a request for SWM with window width just $r.t_{\min}$, leaving the remaining quantity $t - r.t_{\min}$ for later on. This already results in $r.t_{\min} = 0$, which indicates that there is a collapsed segment. We descend the tree to find such a collapsed segment (activating nodes on the way there and deactivating them on the way back), and take care of this segment appropriately (this may affect neighbouring segments as well). We repeat the process as long as $r.t_{\min} = 0$. Once this value is positive again, we are ready to proceed with further application of SWM.

As far as the running time is concerned, the cost of SWM consists of a logarithmic term for visiting the endpoints and further logarithmic terms for each collapsed segment.

3.5   Complexity Analysis

We complete this section by showing that the aforementioned operations run in *amortised* logarithmic time.

▶ **Lemma 7.** *A sequence of k operations on piecewise linear functions takes $\mathcal{O}(k \log k)$ time.*

Proof. Our potential is $\log k$ multiplied by the total number of turning points in all the stored functions. First, we observe that this potential grows by $\mathcal{O}(\log k)$: each operation creates a constant number of new turning points. In particular, the total number of turning points is $\mathcal{O}(k)$, so manipulating BSTs takes $\mathcal{O}(\log k)$ time. Next, we note that the worst-case running time of most operations is $\mathcal{O}(\log k)$, with extra $\mathcal{O}(\log k)$ time needed for each discarded or collapsed segment. However, every such segment decreases the potential by $\log k$.   ◀

4   An $\mathcal{O}(mn \log(mn))$ time RLE Edit Distance Algorithm

As in the previous algorithm by Chen and Chao [8], we go through the dynamic programming table block by block. For every block, we transform the representation of its input border to the representation of its output border. As mentioned earlier, borders are piecewise linear with gradient ± 1 or 0 so they can be maintained in the structure described in Section 3. We will assume that the left and the top part of the input border of every block are stored in separate structures. We start by generating the structures corresponding to the left and the top border of the whole dynamic programming table. These left and top borders are each a

single decreasing and increasing sequence, respectively. As a result, we can generate the data structure for the parts corresponding to all blocks trivially in $\mathcal{O}(m+n)$ time using $m+n$ create operations. Now, we have to describe how to obtain the structure corresponding to the right and the bottom part of the output border of the current block given the structures corresponding to the left and the top part of its input border. We stress that any structure will be created and then used once as an input to a further operation, which is crucial for the amortisation argument within Lemma 7.

Recall that the semantics of split and join operating on arrays in Section 2 and of split and join operating on piecewise linear functions in Section 3 is slightly different: split now creates two functions that both contain the value of the original function at x_M; symmetrically, join takes two functions defined on $[x_L, x_M]$ and $[x_M, x_R]$ that share the same value at x_M. This is, however, not an issue because the cases in both (1) and (2) overlap at the boundaries.

For a match block, the value stored in an element (i, j) of the output border is a copy of the value stored in the corresponding element (i_d, j_d) of the input border. Recalling that (i_d, j_d) is the intersection of the input border with diagonal $j - i$, this can be readily implemented with a constant number of split and join operations.

For a mismatch block, we need to apply Algorithms 1 and 2, merge the returned solutions by taking the minimum at every position, and finally create separate structures corresponding to the right and the bottom part of the output border with a single split operation. Note that while we have already observed that both input border and output border are piecewise linear with gradient ± 1 or 0, we need to make sure that the same is true for every function obtained inside Algorithms 1 and 2, and for $\mathrm{OUT}_{\mathrm{TOP}}$ and $\mathrm{OUT}_{\mathrm{LEFT}}$ in particular.

▶ **Lemma 8.** *Every function obtained in Algorithms 1 and 2 is piecewise linear with gradient* ± 1 *or* 0.

Proof. Consider Algorithm 1. It is easy to verify that S and hence also S^ℓ and S^r are indeed piecewise linear with gradient ± 1 or 0. Additionally, $S^\ell[i]$ is equal to the minimum in $\mathrm{LEFT}[1, i]$ and so S^ℓ is non-increasing. Consequently, S_1, S_2, and S_3 are all piecewise linear with gradient ± 1 or 0. We only need to verify that the same holds for their concatenation. This is true because each of these three parts corresponds to a case considered in (1), and these cases overlap at the boundaries.

Next, consider Algorithm 2. Similarly as above, it is easy to verify that S and so also S^ℓ and S^r are piecewise linear with gradient ± 1 or 0. Furthermore, $S^r[i]$ is equal to the minimum in $\mathrm{TOP}[w - h + i + 1, w]$ and so S^r is non-decreasing. Thus, S_1 and S_2 are piecewise linear with gradient ± 1 or 0 and the same holds for their concatenation because the cases in (2) overlap at the boundaries.   ◀

We now explain in detail how to implement Algorithm 1. We start with computing S^ℓ and S^r by first calling $\mathsf{SWM}(\mathrm{LEFT}, h - 1)$ and then using split. Next, S_1 is obtained by applying gradient change and shift to S^ℓ, S_2 is obtained by calling create, and S_3 is obtained by applying shift to S^r. Finally, $\mathrm{OUT}_{\mathrm{LEFT}}$ is created with two calls to join.

Algorithm 2 is implemented by calling $\mathsf{SWM}(\mathrm{TOP}, w - 1)$ and then using split. Next, S_1 is obtained by applying shift to S^ℓ, while S_2 is obtained by applying gradient change and shift to S^r. Finally, $\mathrm{OUT}_{\mathrm{TOP}}$ is created by a single call to join.

Having obtained a representation of $\mathrm{OUT}_{\mathrm{LEFT}}$ and $\mathrm{OUT}_{\mathrm{TOP}}$, we call combine to obtain a representation of the output border. Such a call is valid due to Lemma 4. The overall number of operations on structures is $\mathcal{O}(mn)$, making the final time complexity $\mathcal{O}(mn \log(mn))$ by Lemma 7.

─────── **References** ───────

1   Hsing-Yen Ann, Chang-Biau Yang, Chiou-Ting Tseng, and Chiou-Yi Hor. A fast and simple algorithm for computing the longest common subsequence of run-length encoded strings. *Information Processing Letters*, 108(6):360–364, 2008. `doi:10.1016/j.ipl.2008.07.005`.

2   Alberto Apostolico, Gad M. Landau, and Steven Skiena. Matching for Run-Length Encoded Strings. *Journal of Complexity*, 15(1):4–16, 1999. `doi:10.1006/jcom.1998.0493`.

3   Ora Arbell, Gad M. Landau, and Joseph S. B. Mitchell. Edit distance of run-length encoded strings. *Information Processing Letters*, 83(6):307–314, 2002. `doi:10.1016/S0020-0190(02)00215-6`.

4   Arturs Backurs and Piotr Indyk. Edit Distance Cannot Be Computed in Strongly Subquadratic Time (Unless SETH is False). *SIAM Journal on Computing*, 47(3):1087–1097, 2018. `doi:10.1137/15M1053128`.

5   Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In Venkatesan Guruswami, editor, *56th Annual Symposium on Foundations of Computer Science, FOCS 2015*, pages 79–97. IEEE Computer Society, 2015. `doi:10.1109/FOCS.2015.15`.

6   Horst Bunke and János Csirik. An algorithm for matching run-length coded strings. *Computing*, 50(4):297–314, 1993. `doi:10.1007/BF02243873`.

7   Horst Bunke and János Csirik. An improved algorithm for computing the edit distance of run-length coded strings. *Information Processing Letters*, 54(2):93–96, 1995. `doi:10.1016/0020-0190(95)00005-W`.

8   Kuan-Yu Chen and Kun-Mao Chao. A fully compressed algorithm for computing the edit distance of run-length encoded strings. *Algorithmica*, 65(2):354–370, 2013. `doi:10.1007/s00453-011-9592-4`.

9   Stuart C. Hinds, James L. Fisher, and Donald P. D'Amato. A document skew detection method using run-length encoding and the Hough transform. In *10th International Conference on Pattern Recognition, ICDR 1990*, volume 1, pages 464–468. IEEE Computer Society, 1990. `doi:10.1109/ICPR.1990.118147`.

10   Jia Jie Liu, Guan-Shieng Huang, Yue-Li Wang, and Richard C. T. Lee. Edit distance for a run-length-encoded string and an uncompressed string. *Information Processing Letters*, 105(1):12–16, 2007. `doi:10.1016/j.ipl.2007.07.006`.

11   Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005. URL: `https://users.dcc.uchile.cl/~gnavarro/ps/njc05.pdf`.

12   William J. Masek and Mike Paterson. A faster algorithm for computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980. `doi:10.1016/0022-0000(80)90002-1`.

13   Yoshifumi Sakai. Computing the longest common subsequence of two run-length encoded strings. In Kun-Mao Chao, Tsan-sheng Hsu, and Der-Tsai Lee, editors, *23rd International Symposium on Algorithms and Computation, ISAAC 2012*, volume 7676 of *LNCS*, pages 197–206. Springer, 2012. `doi:10.1007/978-3-642-35261-4_23`.

14   Taras K. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics*, 4(1):52–57, 1968. `doi:10.1007/bf01074755`.

15   Dong-Hui Xu, Arati S. Kurani, Jacob D. Furst, and Daniela S. Raicu. Run-length encoding for volumetric texture. In Juan J. Villanieva, editor, *4th IASTED International Conference on Visualization, Imaging, and Image Processing, VIIP 2004*. Acta Press, 2004. URL: `http://facweb.cs.depaul.edu/research/vc/Publications/final_submission_paper_452_131_last.pdf`.