Dynamic RLE-compressed edit distance tables under general weighted cost functions

Heikki Hyyrö
Faculty of Natural Sciences, University of Tampere, Finland
heikki.hyyro@uta.fi

Shunsuke Inenaga
Department of Informatics, Kyushu University, Japan
inenaga@inf.kyushu-u.ac.jp

Abstract

Kim and Park [A dynamic edit distance table, J. Disc. Algo., 2:302–312, 2004] proposed a method (KP) based on a "dynamic edit distance table" that allows one to efficiently maintain unit cost edit distance information between two strings A of length m and B of length n when the strings can be modified by single-character edits to their left or right ends. This type of computation is useful e.g. in cyclic string comparison. KP uses linear time, $O(m + n)$, to update the distance representation after each single edit. Recently Hyyrö et al. [Incremental string comparison, J. Disc. Algo., 34:2-17, 2015] presented an efficient method for maintaining the dynamic edit distance table under general weighted edit distance, running in $O(c(m+n))$ time per single edit, where c is the maximum weight of the cost function. The work noted that the $\Theta(mn)$ space requirement, and not the running time, may be the main bottleneck in using the dynamic edit distance table. In this paper we take the first steps towards reducing the space usage of the dynamic edit distance table by RLE compressing A and B. Let M and N be the lengths of RLE compressed versions of A and B, respectively. We propose how to store the dynamic edit distance table using $\Theta(mN + Mn)$ space while maintaining the same time complexity as the previous methods for uncompressed strings.

1 Introduction

Edit distance is a classic and widely used similarity measure between two strings A and B. In this paper we concentrate on Levenshtein-type edit distance that is defined as the minimum total cost of a sequence of single-character insertions, deletions, and/or substitutions that transform A into B.

Let m and n be the lengths of A and B, respectively, and $ed(A, B)$ the edit distance between A and B. The fundamental $\Theta(mn)$ time dynamic programming edit distance algorithm computes information in a “directional” manner. W.l.o.g. we assume the typical left-to-right direction that allows one to efficiently cope with changes to the *right* ends of A or B: the solution to $ed(A, B)$ can for example be updated into a solution to $ed(Ac, B)$ or $ed(A, Bc)$, where c is a character appended to the right end of A or B, in $\Theta(n)$ or $\Theta(m)$ additional time, respectively.[1] Changes to the *left* end are much more costly: e.g. updating a standard dynamic programming solution to $ed(A, B)$ into a solution to $ed(cA, B)$ or $ed(A, cB)$, where c is prepended to the left end of A or B, takes $\Theta(mn)$ worst-case time.

There are several solutions (mainly [14, 12, 11]) that can handle left-end modifications in $O(m + n)$ time under unit cost edit distance. Efficient support for left-end modifications is important in many applications, such as e.g. cyclic string comparison and computing approximate periods (see [14, 17, 12, 8, 4] for more details). The key to overcome the $\Theta(mn)$ limit of the basic dynamic programming algorithm is to use an indirect representation of the edit distance information. In this paper we concentrate on the difference-representation used by the ”dynamic edit distance table” that was first introduced by Kim and Park [12] for unit cost edit distance and recently extended by Hyyrö et al. [11] to general weighted edit distance.

It was noted in [11] that the main practical limitation of the dynamic edit distance table is its $\Theta(mn)$ space requirement. As the first step towards reducing space usage, the previous version [10] of this present article considered a method to compactly store the dynamic edit distance table under the unit cost function. Let M and N denote the sizes of *run-length encodings* (*RLEs*) of A and B, respectively. We presented an algorithm which updates a sparse representation of the dynamic edit distance table of size $\Theta(mN + Mn)$ in linear $O(m+n)$ time per left-end modification.

This present paper extends our method in [10] to general weighted edit distance. To deal with weighted costs, the proposed algorithm processes each block of the edit distance table defined by two runs from A and B in a bit different manner from the basic algorithm of [10] for unit cost edit distance, but it retains the efficiency. Namely, the proposed algorithm stores the dynamic edit distance table for weighted costs with $\Theta(mN + Mn)$ space and updates it in $O(c(m + n))$ time per left-end modification, where c is the maximum weight of the cost function. Hence, the proposed algorithm runs in $O(m + n)$ time per left-end modification for constant weights.

Related work. The problem of computing the edit distance and its related metrics between two RLE compressed strings has been extensively studied in the literature (e.g., see [5, 3, 15, 9, 1, 2, 16, 6, 13]). Among all the existing work, our methods are most related to the followings:

Arbell et al. [3] showed how to store the edit distance table with $\Theta(mN+Mn)$ space and update it in $O(m + n)$ time per right-modification. Their algorithm

[1]The case of right-to-left direction is symmetric and would within the context of this paper only result in interchanging the notions of “left” and “right” ends of a string.

only supports unit cost edit distance. The previous version [10] of this present article is built on Arbell et al.'s method. Bunke and Csirik [5] considered the problem of computing the *longest common subsequence* (LCS) of two RLE compressed strings, and showed an algorithm which uses $\Theta(mN + Mn)$ space and runs in $O(m + n)$ time per right-modification. This relates to the edit distance allowing only for insertions and deletions (namely the in-del distance). Mäkinen et al. [15] proposed an algorithm which stores the edit distance for general weighted costs with $\Theta(mN + Mn)$ space, and updates it in $O(m + n)$ time per right-modification. Our method proposed in this present article is built on their method.

2 Preliminaries

We use the following notation with strings. The set of all characters (alphabet) is Σ. Let A be a string consisting of m characters. For $1 \leq i \leq m$, $A[i]$ denotes the ith character of A, and for $1 \leq i \leq j \leq m$, $A[i : j]$ denotes the substring of A that starts at its ith character and ends at its jth character. If $i > j$, we define $A[i : j] = \varepsilon$, where ε denotes the empty string.

Run length encoding (RLE) is a string compression method that compresses a string A by replacing each maximally long substring $A[i : j]$, where $A[i] = A[k]$ for all $k \in [i..j]$, by the pair $(a, j - i + 1)$, where $a = A[i]$. That is, each maximal run of equal characters is replaced by a value-pair that describes the character and the length of the run. It is usual to express such pairs (a, x) in the form a^x. For example if $A =$ `aaaabbacccbbaabbb`, the RLE compressed representation of A may be written as $\mathtt{a}^4\mathtt{b}^2\mathtt{a}^1\mathtt{c}^3\mathtt{b}^2\mathtt{a}^2\mathtt{b}^3$. The length of an RLE compressed string (or the RLE length of a string) is the number of maximal runs in it. E.g. the length of the preceding example string A is 17 and its RLE length is 7. RLE compression is effective when the strings contain long runs of equal characters. This makes RLE compression useful e.g. in image compression: pixel rows tend to contain relatively long runs of similar pixels, which allows to save space by storing pixel rows as RLE compressed strings.

Let $ed(A, B)$ denote the edit distance between two strings A and B. The distance $ed(A, B)$ is generally defined as the minimum total cost of a sequence of edit operations that transforms A into B, where the individual operation costs are given by a predefined cost function δ. The value $\delta(x, y)$ is defined for all $x, y \in \Sigma \cup \{\varepsilon\}$ and specifies a non-negative cost for replacing x with y. We assume that δ obeys the triangle inequality: $\delta(x, y) \leq \delta(x, z) + \delta(z, y)$ for all $x, y, z \in \Sigma \cup \{\varepsilon\}$.

The cost function essentially defines a Levenshtein-type edit distance that permits the following three edit operations for a string A:

1. Insert a character x after position i of A. If $i = 0$, insert it to the left end. The operation cost is $\delta(\varepsilon, x)$.

2. Delete the character a_i from position i of A. The operation cost is $\delta(a_i, \varepsilon)$.

3. Substitute the character a_i at position i of A by a character x. The operation cost is $\delta(a_i, x)$.

We further let $ed_1(A, B)$ denote the so-called *unit cost edit distance* between A and B, which uses the specific operation costs $\delta(x, y) = 0$, if $x = y$, and $\delta(x, y) = 1$, if $x \neq y$. Note that $ed_1(A, B)$ essentially corresponds to the minimum *number* of edit operations required in transforming A into B. For example $ed_1(\texttt{apple}, \texttt{carpe}) = 3$ and an optimal three-operation way to transform $A = \texttt{apple}$ into $B = \texttt{carpe}$ is to delete $A[4] = \texttt{l}$, substitute $A[2] = \texttt{p}$ by $\texttt{r}$ and insert the character $\texttt{c}$ to the front.

Throughout the paper let m denote the length of A and n denote the length of B. The fundamental $\Theta(mn)$ time solution for computing $ed(A, B)$ fills an $(m+1) \times (n+1)$ dynamic programming table D with values $D[i, j] = ed(A[1 : i], B[1 : j])$ for $0 \leq i \leq m$ and $0 \leq j \leq n$. The cell $D[m, n]$ will hold the desired result $ed(A, B)$. Each value $D[i, j]$ is computed using the following well-known recurrence (1).

$$\begin{aligned}
&D[i,0] = \textstyle\sum_{h=1}^{i}\ \delta(a_h, \varepsilon) \text{ for } 0 \leq i \leq m,\\
&D[0,j] = \textstyle\sum_{h=1}^{j}\ \delta(\varepsilon, b_h) \text{ for } 0 \leq j \leq n, \text{ and}\\
&D[i,j] = \ \min\{D[i, j-1] + \delta(\varepsilon, b_j), D[i-1, j] + \delta(a_i, \varepsilon),\\
&\qquad\qquad D[i-1, j-1] + \delta(a_i, b_j)\}, \text{ for } 1 \leq i \leq m \text{ and } 1 \leq j \leq n.
\end{aligned} \tag{1}$$

Note that under unit costs, each cost of form $\delta(x, \varepsilon)$ or $\delta(\varepsilon, x)$ in the recurrence could be replaced by the value 1.

It is often useful to view edit distance computation as shortest path computation in a grid graph where each cell $D[i, j]$ is a node and each cell $D[i, j]$ with $i > 0$ and $j > 0$ has three incoming directed edges: one with weight $\delta(\varepsilon, b_j)$ from the node $D[i, j-1]$, one with weight $\delta(a_i, \varepsilon)$ from the node $D[i-1, j]$, and one with weight $\delta(a_i, b_j)$ from the node $D[i-1, j-1]$. The boundary cells $D[0, j]$ with $j > 0$ have an incoming edge with weight $\delta(\varepsilon, b_j)$ from the node $D[0, j-1]$, and the boundary cells $D[i, 0]$ with $i > 0$ have an incoming edge with weight $\delta(a_i, \varepsilon)$ from the node $D[i-1, 0]$. Now each value $D[i, j] = ed(A[1 : i], B[1 : j])$ corresponds to the length of a shortest path from the start node $D[0, 0]$ to the node $D[i, j]$. Paths in this type of grid graph are typically called edit paths.

In the rest of the paper we assume that we are given edit distance information, e.g. the table D, that corresponds to computing $ed(A, B)$, and that the string B will then be subjected to an edit operation at its left or right end. The case of editing A is symmetric. Let B' denote B after the operation. The goal is to update the edit distance information, for example D, so that it corresponds to $ed(A, B')$.

Let D' denote D after it has been updated to correspond to $ed(A, B')$. If the operation to B is done at its right end, in which case either $B' = Bc$ (insertion), $B' = B[1 : n-1]$ (deletion) or $B' = B[1 : n-1]c$ (substitution), D may be updated into D' in $O(m)$ time by computing a single column at index $j = n$ or $j = n + 1$ using recurrence (1). It is well-known (see e.g. [12]) that any of the

		c	a	r	p	e
	0	1	2	3	4	5
a	1	1	1	2	3	4
p	2	2	2	2	2	3
p	3	3	3	3	2	3
l	4	4	4	4	3	3
e	5	5	5	5	4	3

D

		c	a	r	p	e
a	1	0	-1	-1	-1	-1
p	1	1	1	0	-1	-1
p	1	1	1	1	0	0
l	1	1	1	1	1	0
e	1	1	1	1	1	0

DR.U

	c	a	r	p	e
	1	1	1	1	1
a	0	0	1	1	1
p	0	0	0	0	1
p	0	0	0	-1	1
l	0	0	0	-1	0
e	0	0	0	-1	-1

DR.L

Figure 1: The tables D and DR for $A =$ `apple` and $B =$ `carpe` under unit cost edit distance.

analogous left end modifications, corresponding to either $B' = cB$ (insertion), $B' = B[2 : n]$ (deletion) or $B' = cB[2 : n]$ (substitution), may lead to up to $\Theta(mn)$ differences between D and D'. This gives a worst-case bound of $\Theta(mn)$ for updating D into D'.

3 The dynamic edit distance table

The "dynamic edit distance table", originally proposed by Kim and Park [12] for unit cost edit distance, avoids the $\Theta(mn)$ bound of updating D into D' by maintaining a difference representation DR of D (instead of the original D). Each cell $DR[i, j]$ of the difference table has two fields: a vertical (upper) difference $DR[i, j].U$ and a horizontal (left) difference $DR[i, j].L$. These difference values are defined as

$$\begin{aligned} DR[i,j].U &= D[i,j] - D[i-1,j] \text{ and} \\ DR[i,j].L &= D[i,j] - D[i,j-1], \text{ for } i = 1,\dots,m \text{ and } j = 1,\dots,n. \end{aligned}$$

That is, $DR[i, j].U$ tells the difference between $D[i, j]$ and its upper neighbor $D[i-1, j]$ and $DR[i, j].L$ tells the difference between $D[i, j]$ and its left neighbor $D[i, j-1]$. Fig. 1 shows an example of D and the corresponding U- and L-values in DR under unit cost edit distance. Note that if we have only DR available, computing an arbitrary $D[i, j]$ value requires $O(\min\{m, n\})$ time since we need to backtrack $\min\{i, j\}$ cells from $DR[i, j]$. However, the computation of DR can be set to keep track of a constant number of specific values of interest, such as $D[m, n] = ed(A, B)$, without causing asymptotic (or practically significant) overhead. This makes DR sufficient for many applications.
Let DR' denote DR after it has been updated to correspond to $ed(A, B')$. Modifying the left end of B may shift column indices within B and DR. E.g. if a character is deleted from the left end of B, then for $j = 2, \dots, n$ the equality $B[j-1] = B'[j]$ holds and column $j-1$ in DR corresponds to column j in DR'. We define ℓ as a correcting offset: $\ell = -1$ if a character was deleted from the left

end, $\ell = 1$ if a character was inserted to the left end of B, and $\ell = 0$ otherwise. Now $B[j-\ell] = B'[j]$ and column $j - \ell$ in DR corresponds to column j in DR'.

The crucial benefit from using DR instead of D is that under a variety of cost functions DR and DR' differ in much less than $\Theta(mn)$ positions. In fact, as first shown by Kim and Park [12], under unit cost edit distance DR differs from DR in at most $O(m+n)$ positions. A more general characterization is given by the following Theorem 1, which is derived from the proof of Theorem 9 in [11].

Theorem 1 ([11]) *Let c be the maximum weight of the cost function δ. Any single row $i \in [1\,..\,m]$ of DR' contains at most $O(c)$ columns j where $DR'[i,j].L \neq DR[i,j-\ell].L$. Any single column $j \in [1\,..\,n]$ of DR' contains at most $O(c)$ rows i where $DR'[i,j].U \neq DR[i,j-\ell].U$. Overall the table DR' contains at most $O(c(m+n))$ positions where $DR'[i,j] \neq DR[i,j-\ell]$.*

Both the unit cost algorithm of Kim and Park [12] and the general cost algorithm of Hyyrö et al. [11] update DR into DR' in $O(m + \#_{ch})$ time, where $\#_{ch}$ denotes the overall number of differing positions between DR and DR'. Thus both algorithms are optimal in the sense that their running times are directly proportional to the number of entries that change when DR is transformed into DR'. The following result concerning the algorithm of Hyyrö et al. follows from Theorem 1.

Theorem 2 ([11]) *Let c be the maximum weight of the cost function δ. DR can be updated to DR' in $O(c(m+n))$ time.*

Note that the running time is linear under unit cost edit distance, as then $c = 1$.

Let us first briefly review how the algorithm of Hyyrö et al. [11] (we will from now on call it HNI) works. It is based on using Lemma 1 which states those cells in DR' that need to be recomputed.

Lemma 1 ([11]) *Assume that the values $DR'[i^*, j^*]$ are correct for all cells where $i^* < i$ or $j^* < j$. The entry $DR'[i,j]$ needs to be recomputed if and only if $DR'[i-1,j].L \neq DR[i-1,j-\ell].L$ or $DR'[i,j-1].U \neq DR[i,j-1-\ell].U$.*

Recall that the value ℓ referred to in Lemma 1 is a correcting offset that keeps the indices aligned correctly when comparing values in DR and DR'. If $\ell = 1$ and Lemma 1 is applied in the first non-trivial column $j = 1$, the lemma references values $DR[i,-1].U$ in a non-existing column $j - \ell - 1 = -1$. We accommodate such negative columns by using the convention that $D[i,j] = D[i,0]$ if $j < 0$. This defines the values $DR[i,-1]$ as $DR[i,-1].U = D[i,-1] - D[i-1,-1] = D[i,0] - D[i-1,0]$ and $DR[i,-1].L = D[i,-1] - D[i,-2] = D[i,0] - D[i,0] = 0$.

We also remark that the same arguments as Lemma 1 apply to DS and DS', which are respectively sparse representations of DR and DR' based on the RLEs of the strings. The formal definitions of DS and DS' will be given later in Section 6.

Lemma 1 has two valuable consequences. The first is that when B is modified, we only need to update values in DR' from that column j' onwards (towards right) into which the modification was done. The second is that if this computation has determined at some column $j' + k$ that $DR'[i, j' + k].U = DR[i, j' + k - \ell].U$ in all rows $i = 1 \dots m$, then the computation can be stopped and DR' is ready.

The algorithm uses the following recurrence (2) when (re)computing the value of any entry $DR[i, j]$.

$$
\begin{aligned}
&DR[i,0].U = \delta(a_i, \varepsilon) \text{ for every } 1 \le i \le m,\\
&DR[0,j].L = \delta(\varepsilon, b_j) \text{ for every } 1 \le j \le n, \text{ and}\\
&DR[i,j].U = z - DR[i-1,j].L \text{ and } DR[i,j].L = z - DR[i,j-1].U, \text{where}\\
&z = \min\{DR[i-1,j].L + \delta(\varepsilon, b_j), DR[i,j-1].U + \delta(a_i,\varepsilon), \delta(a_i, b_j)\}, \text{ for}\\
&\quad \text{every } 1 \le i \le m \text{ and every } 1 \le j \le n.
\end{aligned}
\tag{2}
$$

Assume that HNI is currently processing column j. The algorithm maintains a sorted list *prevΔ* of rows i that may need to be recomputed in column j, that is, those indices i for which the inequality $DR'[i, j-1].U \neq DR[i, j-1-\ell].U$ was true in the previous column $j - 1$.[2] This enforces the second condition in Lemma 1. HNI processes the column j rows listed in *prevΔ* in increasing row order. Each such cell $DR'[i, j]$ is recomputed, and the U- and L-fields of the new value are compared with the old ones (which corresponded to $DR[i, j - \ell]$). If the the U-fields do not match, the row i of the next column $j + 1$ is added to a second list, *currΔ*, that will later take the role of *prevΔ* for column $j + 1$. If the L-fields do not match, the first rule of Lemma 1 is enforced: also the row $i + 1$ in column j will be computed (regardless of whether row $i + 1$ is present in *prevΔ* or not). The computation can be stopped if *currΔ* remains empty or j was the last column of DR.

4 Edit distance of RLE compressed strings

For the rest of the paper we assume that A and B have been RLE compressed and denote their RLE lengths by M and N, respectively. In this section we mostly follow the ideas of the algorithm of Mäkinen et al. [15] that computes $ed(A, B)$ in $\Theta(mN + Mn)$ time under general weighted edit distance. Their algorithm is superficially identical with the algorithm of Arbell et al. [3] for unit cost distance; the algorithms differ only in details related to handling general costs. Note that the time $\Theta(mN + Mn)$ complexity holds even if A and B are given in uncompressed form: in that case A and B can first be RLE compressed in $O(m + n)$ time.

[2] In case of using a linked representation of DR, the list should also contain pointers to the corresponding entries in column j.

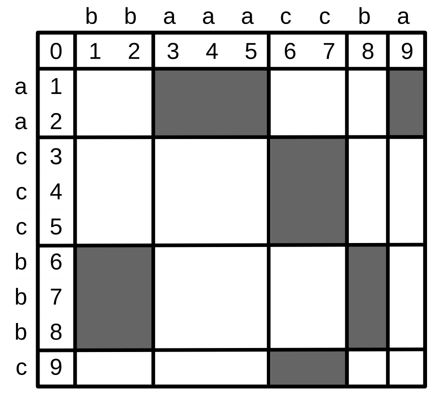

Figure 2: Boxes defined by intersecting RLE runs. Boxes that consist of matching characters are highlighted with dark grey.

The key idea is to divide the dynamic programming table D into "boxes" that are defined by intersections of maximal runs of A and B (see Fig. 2). D contains $(M+1)\times(N+1)$ such boxes. Let M^I denote the length of the Ith run in A and N^J denote the length of the Jth run in B. We also define the end position of the Ith run in A as $i_{\mathrm{B}}^{I} = \Sigma_{k=1}^{I} M^k$ and the end position of the Jth run in B as $j_{\mathrm{R}}^{J} = \Sigma_{k=1}^{J} N^k$. It is convenient to define special cases $i_{\mathrm{B}}^{0} = j_{\mathrm{R}}^{0} = 0$ and $i_{\mathrm{B}}^{-1} = j_{\mathrm{R}}^{-1} = -1$. Under these conventions, the box $\mathcal{B}^{I,J}$ is defined for $0 \le I \le M$ and $0 \le J \le N$ as the two dimensional index interval that spans the rows $i = i_{\mathrm{B}}^{I-1}+1,\ldots,i_{\mathrm{B}}^{I}$ and the columns $j = j_{\mathrm{R}}^{J-1}+1,\ldots,j_{\mathrm{R}}^{J}$. Thus i_{B}^{I} tells the bottom row and j_{R}^{J} the rightmost column in the box $\mathcal{B}^{I,J}$.

We may say that the box $\mathcal{B}^{I,J}$ resides on box row I and in box column J. Since the box $\mathcal{B}^{I,J}$ is an index interval instead of a concrete sub-table of D, we may refer to a box $\mathcal{B}^{I,J}$ also in alternative representations of D, such as DR.

The table D is processed one box at a time, and in each box $\mathcal{B}^{I,J}$ only the cells on its right/bottom boundary (in rightmost column j_{R}^{J} and/or bottom row i_{B}^{I}) are filled. It is convenient to define the left/top boundary to consist of those cells that are immediate left/top neighbours of $\mathcal{B}^{I,J}$: the left boundary resides on column j_{R}^{J-1} and the top boundary on row i_{B}^{I-1}. The boxes are processed in such a manner that the left/top neighboring boxes $\mathcal{B}^{I-1,J-1}$, $\mathcal{B}^{I,J-1}$ and $\mathcal{B}^{I-1,J}$ are processed before the box $\mathcal{B}^{I,J}$. This guarantees that the cells in the left/top boundary have been computed before $\mathcal{B}^{I,J}$. The values in the right/bottom boundary can be computed from the values in the left/top boundary.

Throughout the rest of this section we concentrate on computing a $D[i,j]$ inside the current box $\mathcal{B}^{I,J}$. Let a and b be respectively the characters of A and B whose runs the current box $\mathcal{B}^{I,J}$ corresponds to. Now, the dynamic programming recurrence (1) has the form

$$D[i,j] = \min\{D[i,j-1]+\delta(\varepsilon,b), D[i-1,j]+\delta(a,\varepsilon), D[i-1,j-1]+\delta(a,b)\} \quad (3)$$

for any cell (i,j) inside the current box $\mathcal{B}^{I,J}$. According to recurrence (3), the box $\mathcal{B}^{I,J}$ corresponds to a grid subgraph with uniform costs for each direction: each horizontal step from $(i,j-1)$ to (i,j) costs $\delta(\varepsilon,b)$, each vertical step from $(i-1,j)$ to (i,j) costs $\delta(a,\varepsilon)$, and each diagonal step from $(i-1,j-1)$ to (i,j) costs $\delta(a,b)$.

Consider a reversed optimal edit path from the cell (i,j) to the cell $(0,0)$ and define (i^*,j^*) as the first cell on this path that resides on the left/top boundary of $\mathcal{B}^{I,J}$. We will analyze an optimal subpath $\mathcal{P}$ from the cell (i,j) to the cell

(i^*, j^*). Note that (i^*, j^*) must be the only left/top boundary cell in $\mathcal{P}$. We say that $\mathcal{P}$ is an L-path, when the cell $(i^*, j^*) = (i^*, j_{\mathrm{R}}^{J-1})$ resides on the left boundary, and a T-path, when the cell $(i^*, j^*) = (i_{\mathrm{B}}^{I-1}, j^*)$ resides on the top boundary.

Let $h(i, j, i^*, j^*)$, $v(i, j, i^*, j^*)$ and $d(i, j, i^*, j^*)$ denote the number of horizontal, vertical and diagonal steps along $\mathcal{P}$. This gives rise to the equality

$$D[i, j] = D[i^*, j^*] + h(i, j, i^*, j^*)\delta(\varepsilon, b) + v(i, j, i^*, j^*)\delta(a, \varepsilon) + d(i, j, i^*, j^*)\delta(a, b). \tag{4}$$

In order to use equation (4), we need to determine the values $h(i, j, i^*, j^*)$, $v(i, j, i^*, j^*)$ and $d(i, j, i^*, j^*)$. Since δ obeys the triangle inequality, $\mathcal{P}$ will always contain as many diagonal steps as possible. This gives the equality $d(i, j, i^*, j^*) = \min\{i - i^*, j - j^*\}$. Depending on which of the previous two minimum cases holds, the remaining steps (if any) will be either $h(i, j, i^*, j^*) = j_{\mathrm{R}}^{J} - d(i, j, i^*, j^*) - j^*$ horizontal steps from $(i^*, j - d(i, j, i^*, j^*))$ to (i^*, j^*), or $v(i, j, i^*, j^*) = i - d(i, j, i^*, j^*) - i^*$ vertical steps from $(i - d(i, j, i^*, j^*), j^*)$ to (i^*, j^*). This implies that an L-path contains only diagonal and horizontal steps and a T-path only diagonal and vertical steps. Let $s = \min\{i - i_{\mathrm{B}}^{J-1}, j - j_{\mathrm{R}}^{J-1}\}$ denote the minimum distance from (i, j) to the left/top boundary. An L-path must end in one of the $s + 1$ left boundary cells $(i^*, j_{\mathrm{R}}^{J-1})$ for $i^* = i - s, \ldots, i$, and a T-path must end in one of the $s + 1$ top boundary cells $(i_{\mathrm{B}}^{I-1}, j^*)$ for $j^* = j - s, \ldots, j$, as the maximal number of s diagonal steps will reach the boundary cell $(i - s, j_{\mathrm{R}}^{J-1})$ or $(i_{\mathrm{B}}^{I-1}, j - s)$ before any cell $(i - s - k, j_{\mathrm{R}}^{J-1})$ or $(i_{\mathrm{B}}^{I-1}, j - s - k)$ with $k \geq 1$. We call the set of possible L-path end cells an "L-zone" and the set of possible t-path end cells"T-zone", respectively. Fig. 3 illustrates.

Let us define $L_\delta(i, j, i^*, j_{\mathrm{R}}^{J-1})$ as the minimum cost for an L-path between (i, j) and an L-zone cell $(i^*, j_{\mathrm{R}}^{J-1})$, and $T_\delta(i, j, i_{\mathrm{B}}^{I-1}, j^*)$ as the minimum cost for a T-path between (i, j) and a T-zone cell $(i_{\mathrm{B}}^{I-1}, j^*)$. These now have the following equalities:

$$\begin{aligned} L_\delta(i, j, i^*, j_{\mathrm{R}}^{J-1}) &= (j - j_{\mathrm{R}}^{J-1} - (i - i^*))\delta(\varepsilon, b) + (i - i^*)\delta(a, b), \\ T_\delta(i, j, i_{\mathrm{B}}^{I-1}, j^*) &= (i - i_{\mathrm{B}}^{I-1} - (j - j^*))\delta(a, \varepsilon) + (j - j^*)\delta(a, b). \end{aligned}$$

The preceding discussion gives rise to the following Lemma 2, which is essentially a translation of Lemma 5 in Mäkinen et al. [15] to use our conventions.

Lemma 2 ([15]) *Let (i, j) reside in the box $\mathcal{B}^{I,J}$ and a and b be the corresponding characters of A and B. Then $D[i, j] = \min\{Z_L, Z_T\}$, where*
$Z_L = \min_{i-s \leq i^* \leq i}(D[i^*, j_R^{J-1}] + L_\delta(i, j, i^*, j_R^{J-1}))$, and
$Z_T = \min_{j-s \leq j^* \leq j}(D[i_B^{I-1}, j^*] + T_\delta(i, j, i_B^{I-1}, j^*))$.

Lemma 2 simply computes $D[i, j]$ as the minimum value given by equality (4) when the set of candidate cells $D[i^*, j^*]$ is confined within the L- and T-zones. The values Z_L and Z_T in Lemma 2 represent the minimum values of the respective zones.

A deque with heap order [7] ("min-deque") maintains a set of values in a double ended queue, requires $O(1)$ amortized time per queue insertion or removal, and provides the minimum value in the queue in $O(1)$ time. We will describe the details of this technique later in the next section, as it will also be used as a building block of our dynamic edit distance computation for RLE compressed strings.

If the L-zone values defined by Lemma 2 are stored in a min-deque `Ldeque` and the T-zone values in a min-deque `Tdeque`, the computation $D[i,j] = \min\{Z_L, Z_T\} = \min\{\texttt{Ldeque.min}(), \texttt{Tdeque.min}()\}$ takes $O(1)$ time. The min-deques can be maintained efficiently when the right boundary cells $D[i^*, j_{\mathrm{R}}^{J}]$ are computed in the order $i^* = i_{\mathrm{B}}^{I-1}+1, \ldots, i_{\mathrm{B}}^{I}$ and the bottom boundary cells $D[i_{\mathrm{B}}^{I}, j^*]$ in the order $j^* = j_{\mathrm{R}}^{J-1}+1, \ldots, j_{\mathrm{R}}^{J}$. Each boundary is handled separately.

Consider first the first right boundary value $D[i_{\mathrm{B}}^{I-1}+1, j_{\mathrm{R}}^{J}]$. `Ldeque` is initialized with the two L-zone based values $D[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1}] + L_\delta(i_{\mathrm{B}}^{I-1}+1, j_{\mathrm{R}}^{J}, i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1})$ and $D[i_{\mathrm{B}}^{I-1}+1, j_{\mathrm{R}}^{J-1}] + L_\delta(i_{\mathrm{B}}^{I-1}+1, j_{\mathrm{R}}^{J}, i_{\mathrm{B}}^{I-1}+1, j_{\mathrm{R}}^{J-1})$. `Tdeque` is initialized with the two T-zone based values $D[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J}-1] + T_\delta(i_{\mathrm{B}}^{I-1}+1, j_{\mathrm{R}}^{J}, i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J}-1)$ and $D[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J}] + T_\delta(i_{\mathrm{B}}^{I-1}+1, j_{\mathrm{R}}^{J}, i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J})$. After this the value $D[i_{\mathrm{B}}^{I-1}+1, j_{\mathrm{R}}^{J}]$ is computed as $D[i_{\mathrm{B}}^{I-1}+1, j_{\mathrm{R}}^{J}] = \min\{Z_L, Z_T\} = \min\{\texttt{Ldeque.min}(), \texttt{Tdeque.min}()\}$. See the rightmost case in Fig. 3.

Consider next the first left boundary value $D[i_{\mathrm{B}}^{I}, j_{\mathrm{R}}^{J-1}+1]$. `Ldeque` is initialized with the two L-zone based values $D[i_{\mathrm{B}}^{I}-1, j_{\mathrm{R}}^{J-1}] + L_\delta(i_{\mathrm{B}}^{I}, j_{\mathrm{R}}^{J-1}+1, i_{\mathrm{B}}^{I}-1, j_{\mathrm{R}}^{J-1})$ and $D[i_{\mathrm{B}}^{I}, j_{\mathrm{R}}^{J-1}] + L_\delta(i_{\mathrm{B}}^{I}, j_{\mathrm{R}}^{J-1}+1, i_{\mathrm{B}}^{I}, j_{\mathrm{R}}^{J-1})$. `Tdeque` is initialized with the two T-zone based values $D[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1}] + T_\delta(i_{\mathrm{B}}^{I}, j_{\mathrm{R}}^{J-1}+1, i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1})$ and $D[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1}+1] + T_\delta(i_{\mathrm{B}}^{I}, j_{\mathrm{R}}^{J-1}+1, i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1}+1)$. After this the value $D[i_{\mathrm{B}}^{I-1}+1, j_{\mathrm{R}}^{J}]$ is computed as $D[i_{\mathrm{B}}^{I-1}+1, j_{\mathrm{R}}^{J}] = \min\{Z_L, Z_T\} = \min\{\texttt{Ldeque.min}(), \texttt{Tdeque.min}()\}$.

The preceding provides the base cases for the first steps of each boundary. Now consider how to update `Ldeque` and `Tdeque` as the computation moves from some $(i-1,j)$ to (i,j) or from some $(i,j-1)$ to (i,j). The following easily derived properties state how such steps change the minimal L- and T-path costs:

Step $(i-1,j) \to (i,j)$:

$$L_\delta(i,j,i^*,j_{\mathrm{R}}^{J-1}) = L_\delta(i-1,j,i^*,j_{\mathrm{R}}^{J-1}) + \delta(a,b) - \delta(\varepsilon,b).$$
$$T_\delta(i,j,i^*,j_{\mathrm{R}}^{J-1}) = T_\delta(i-1,j,i^*,j_{\mathrm{R}}^{J-1}) + \delta(a,\varepsilon).$$

Step $(i,j-1) \to (i,j)$:

$$L_\delta(i,j,i^*,j_{\mathrm{R}}^{J-1}) = L_\delta(i,j-1,i^*,j_{\mathrm{R}}^{J-1}) + \delta(\varepsilon,b).$$
$$T_\delta(i,j,i^*,j_{\mathrm{R}}^{J-1}) = L_\delta(i,j-1,i^*,j_{\mathrm{R}}^{J-1}) + \delta(a,b) - \delta(a,\varepsilon).$$

When the right boundary computation moves from $(i^*-1, j_{\mathrm{R}}^{J})$ to $(i^*, j_{\mathrm{R}}^{J})$, the end of L-zone will expand to cover the cell $(i^*, j_{\mathrm{R}}^{J-1})$, and all L-path costs change by $\delta(a,b) - \delta(\varepsilon,b)$. In principle all current values of `Ldeque` should be adjusted by $\delta(a,b) - \delta(\varepsilon,b)$, but the same effect is achieved by decrementing the new value by $\alpha = (i^* - i_{\mathrm{B}}^{I-1} - 1)(\delta(a,b) - \delta(\varepsilon,b))$: the value $D[i^*, j_{\mathrm{R}}^{J-1}] + \mathrm{L}_\delta(i^*, j_{\mathrm{R}}^{J}, i^*, j_{\mathrm{R}}^{J-1}) - \alpha$ is added to the end of `Ldeque`. This readjustment policy is taken into account by computing Z_L as $Z_L = \texttt{Ldeque.min}() + \alpha$. If $s_T > s_L$, the front value of `Ldeque` is removed in order to remove the cell $(i - s_L - 1, j_{\mathrm{R}}^{J-1})$ from the L-zone.

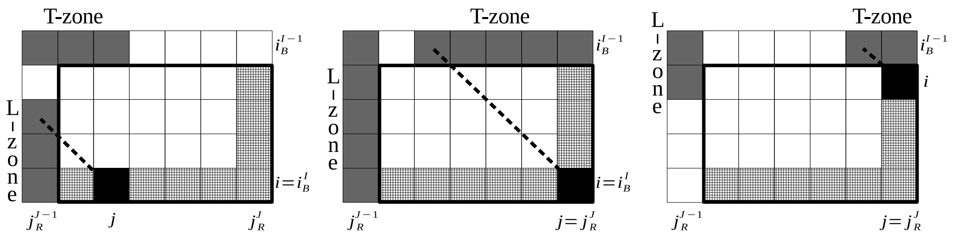

Figure 3: L- and T-zones when computing $D[i, j]$. The filled cells on the right/bottom boundary are highlighted with a grid-pattern. The cells of the L-zone and the T-zone are shown in dark grey. The dashed diagonal lines go from $D[i, j]$ to $D[i - h, j - h]$.

In the case of T-zone, the cell $D[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J} - s_T]$ will enter it if $s_T \leq s_L$. The previous values in `Tdeque` should be incremented by $\delta(a, \varepsilon)$ which is handled by decrementing the new value by $\beta = (i^* - i_{\mathrm{B}}^{I-1} - 1)\delta(a, \varepsilon)$ instead: If $s_T \leq s_L$, the value $D[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J} - s_T] + T_\delta(i^*, j_{\mathrm{R}}^{J}, i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J} - s_T) - \beta$ is added to the end of `Tdeque`, and in any case Z_T is computed as $Z_T = \texttt{Tdeque.min}() + \beta$. No cell leaves T-zone, so no value is removed from `Tdeque`. Now the desired value $D[i^*, j_{\mathrm{R}}^{J}] = \min\{Z_L, Z_T\}$ may be computed, and the whole step has taken $O(1)$ amortized time.

When the bottom boundary computation moves from $(i_{\mathrm{B}}^{J}, j^* - 1)$ to $(i_{\mathrm{B}}^{J}, j^*)$, the cell $D[i_{\mathrm{B}}^{I} - s_L, j^* - s_L]$ enters the front of L-zone if $s_L \leq s_T$. The previous values in `Ldeque` should be incremented by $\delta(\varepsilon, b)$, which is handled by decrementing the new value by $\alpha = (j^* - j_{\mathrm{R}}^{J-1} - 1)\delta(\varepsilon, b)$ instead: If $s_L \leq s_T$, the value $D[i_{\mathrm{B}}^{I} - s_L, j^* - s_L] + L_\delta(i_{\mathrm{B}}^{J}, j^*, i_{\mathrm{B}}^{I} - s_L, j^* - s_L) - \alpha$ is added to the front of `Ldeque`, and in any case Z_L is computed as $Z_L = \texttt{Ldeque.min}() + \alpha$. No cell leaves the L-zone, so no value is removed from `Ldeque`. In the case of T-zone, the cell $(i_{\mathrm{B}}^{I-1}, j^*)$ will enter it and the values of `Tdeque` should be adjusted by $\delta(a, b) - \delta(a, \varepsilon)$. This is handled by decrementing the new value by $\beta = (j^* - j_{\mathrm{R}}^{J-1} - 1)(\delta(a, b) - \delta(\varepsilon, b))$: the value $D[i_{\mathrm{B}}^{I-1}, j^*] + T_\delta(i_{\mathrm{B}}^{J}, j^*, i_{\mathrm{B}}^{I-1}, j^*) - \beta$ is added to the end of `Tdeque`. If $s_L < s_T$, the front value of `Tdeque` is removed in order to remove the cell $(i_{\mathrm{B}}^{I-1}, j^* - s_L - 1)$ from the T-zone. The value Z_T is computed as $Z_T = \texttt{Tdeque.min}() + \beta$. Now the desired value $D[i_{\mathrm{B}}^{J}, j^*] = \min\{Z_L, Z_T\}$ may be computed, and the whole step takes $O(1)$ amortized time.

Since each cell in a right/bottom boundary of a box can be computed in $O(1)$ amortized time and D has altogether $\Theta(mN + Mn)$ such cells, the overall time for computing $ed(A, B)$ is $\Theta(mN + Mn)$.

5 Deques with heap order

In this section we briefly review a simple min-deque structure of Gajewska and Tarjan [7] that supports value insertions and removals at both ends of a queue, as well as minimum value queries, in $O(1)$ amortized time per operation. The original reference describes also a more complicated variant with unamortized $O(1)$ time.

The min-deque consists of a double ended queue `deque` and two stacks, `front` and `end`. The stacks maintain information about minimum values within the "front part" and the "end part" of `queue`: `front` is updated upon an insertion or removal at the front of `deque` and `end` upon an insertion or removal at the end of `deque`.

Let v_i denote the ith value in `queue` and let f be the number of values in the front part: the front part values are $v_1, \ldots, v_f$ and the end part values are $v_{f+1}, \ldots, v_n$. Also let f_i denote the ith value in `front`, e_i the ith value in `end`, $\mathit{last}(x)$ the the largest index i where $v_i = x$, and $\mathit{first}(x)$ the smallest index i where $v_i = x$.

The values f_i in the `front` have a simple recursive definition. The first value is $f_1 = \min\{v_k \mid k \in [1..f]\}$. For $i \geq 1$, the next value f_{i+1} is defined as $f_{i+1} = \min\{v_k \mid k \in [\mathit{first}(f_i) + 1..f]\}$. The value f_{i+1} exists if and only if $\mathit{first}(f_i) < f$. Each value f_{i+1} tells the minimum within the remaining values of the front part if a removal from the front of `deque` removes the first occurrence of f_i. When a new value x is inserted to the front of `deque`, x might become a new smallest value: x is inserted to the front of `front` if it is currently empty or $x \leq f_1$. When the front value v_1 of `deque` is removed, also the first value f_1 of `front` is removed if $f_1 = v_1$. By definition, if a new f_1 (which until now was f_2) exists, it is the minimum value among the remaining values in the front part.

The end part stack `end` is symmetric. Its first value is $e_1 = \min\{v_k \mid k \in [f + 1..n]\}$, the next value e_{i+1} for $i \geq 1$ is defined as $e_{i+1} = \min\{v_k \mid k \in [f + 1..\mathit{last}(f_i) - 1]\}$, and the value e_{i+1} exists if and only if $\mathit{last}(e_i) > f + 1$. Each value e_{i+1} tells the minimum within the remaining values of the end part if values are removed from the end of `deque` up to the last occurrence of e_i. When a new value x is inserted to the end of `deque`, x is inserted to the front of `end` if the end part was empty or $x \leq e_1$. When the last value v_n of `deque` is removed, the first value e_1 is removed if $e_1 = v_n$.

The overall minimum value in `deque` is computed as $\min\{f_1, e_1\}$. Clearly each insertion, removal or minimum operation described so far takes $O(1)$ time. The only complication is if a value is removed from the front (or end) of `deque` but `front` (or `end`) is empty. In this case all n current values are either in the front or the end part. A simple solution is to move $\lceil n/2 \rceil$ of the items to the currently empty part, and then perform the deletion. The move is achieved by rebuilding both stacks from scratch as if there was a sequence of front insertions with values $\approx v_{n/2}, \ldots, v_1$ and a sequence of end insertions with values $\approx v_{n/2+1}, \ldots, v_n$. The values in `deque` remain unchanged. This takes $O(n)$ time, but the amortized cost is $O(1)$ as the cost can be spread over $\Omega(n)$ previous operations that were required in order to create the inbalance $|f - (n - f)| = n$ between the sizes of the front and end parts.

6 Dynamic edit distance table for RLE strings

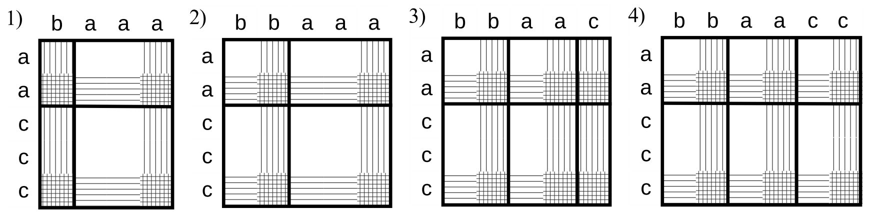

Figure 4: The figures 1) - 4) depict how DS changes when the string B evolves through the strings `baaa`, `bbaaa`, `bbaac` and `bbaaacc` by modifications to its left or right end. Cells stored in DS are shaded with a pattern: vertical pattern shows cells with U-fields, horizontal pattern cells with L-fields and grid-pattern cells with both U- and L-fields.

Let us now turn to the main topic of this paper: handling left (and right) end modifications efficiently when the strings A and B are RLE compressed. Instead of the full difference table DR, we will maintain a "sparse" table DS that contains only those columns and rows that coincide with the right/bottom boundaries of the boxes $\mathcal{B}^{I,J}$. To be more precise, DS stores the values $\{DR.U[i, j^J_{\mathrm{R}}] \mid i \in [1..m], J \in [1..N]\}$ and $\{DR.L[i^I_{\mathrm{B}}, j] \mid I \in [1..M], j \in [1..n]\}$. Note that the stored columns contain only the U-fields and the stored rows only the L-fields. The cells at the intersections of these columns and rows contain both fields. See Fig. 4 for an example. Assume that DS corresponds to $ed(A, B)$ and that B has been changed into B' by a modification to its left or right end. Let DS' denote DS after it has been updated to correspond to $ed(A, B')$. Our goal is to find an efficient way to update DS into DS'.

First we note that even though we discuss only the case of editing B explicitly, the goal is to also allow left or right end edits to A. This means, among other things, that we should be able to efficiently add/remove rows or columns to/from DS when updating it to correspond to the DS'. A suitable solution (like in [12]) is to store DS as a linked structure where each cell $DS[i, j]$ has a pointer to its four neighbours (left, up, right and down). Here we define a "neighbour" to be the nearest cell that actually exists in DS, effectively hopping over those cells of the boxes $\mathcal{B}^{I,J}$ that do not reside on the right/bottom boundary of any $\mathcal{B}^{I,J}$. Such a linked sparse table DS can be stored using $\Theta(mN + Mn)$ space and adding or removing a column or row can be done in $\Theta(n)$ or $\Theta(m)$ time, respectively.

Fig. 4 shows examples of how the form of DS (which cells are stored in it) may change when the left or right end of B is modified. For example if a character is inserted, it either expands the current boxes (step 1 $\rightarrow$ 2 in Fig. 4) or adds completely new boxes (imagine the situation of step 3 without the last character `c` in B). Performing such changes to DS is straight-forward in $O(m)$ time. We assume that when we start to update DS into DS', the preprocessing step of changing the form of DS, if necessary, has already been done. For convenience, we will already refer to this preprocessed (but not yet fully updated) table as DS' (or DR', as the two tables differ only in that the former is a partial representation of the latter).

6.1 Updating DS' after a right end modification

The case of a right end modification essentially corresponds to (re)computing right boundaries of the boxes in at most two rightmost box columns of DS', as e.g. Fig. 4 illustrates. These boundaries can be handled in $O(m)$ time by a straight-forward application of the algorithm of Mäkinen et al. [15] that was discussed in section 4.

6.2 Updating DS' after a left end modification

Our approach for updating DS' after a left end modification is to process DS as if it contained all values of DR'. The algorithm will process essentially the same set of values of DR' as the uncompressed dynamic edit distance table algorithm of Hyyrö et al. [11]. Any value $DR'[i, j]$ that is needed during the update process, but which is not present in the sparse DS' table, will be computed on the fly and forgotten once it is no longer needed.

We update the table DS' one box at a time in a column-wise manner, starting from box column $J = 1$. According to Lemma 1, we need to update a box $B^{I,J}$ only if its left/top boundary contains a changed difference value; otherwise all values within the box are already known to be correct. Recall that the left boundary of $B^{I,J}$ is the right boundary of $B^{I,J-1}$ and the top boundary of $B^{I,J}$ is the bottom boundary of $B^{I-1,J}$. Let *prevBoxΔ* be an ordered list that contains the box row index I of each box $B^{I,J-1}$ whose right boundaries were changed while processing box column $J - 1$. The preceding rule states that a box $B^{I,J}$ needs to be updated only if $I \in$ *prevBoxΔ* or if some difference value on the top boundary of the box $B^{I,J}$ was changed while updating the top neighbor box $B^{I-1,J}$. We assume that a modification to the left end of B may change any box $B^{I,1}$, and so the list *prevBoxΔ* $= \{1, \ldots, M\}$ is used in the first box column $J = 1$. When starting to process column J, we initialize an empty auxiliary list *currBoxΔ*. If updating a box $B^{I,J}$ changes some difference on its right boundary, the box row index I is added to the end of the list *currBoxΔ*. After box column J has been processed, the contents of the lists *currBoxΔ* and *prevBoxΔ* are interchanged: now *prevBoxΔ* is ready for use in the next box column $J + 1$ as it contains the box row indices of all boxes whose right boundaries were changed in box column J.

When updating a box, we want to focus on only those cells that need to be updated according to Lemma 1. To this end we also build the following two lists ${\Delta_\mathrm{B}}^{I,J}$ and ${\Delta_\mathrm{R}}^{I,J}$ for each box $\mathcal{B}^{I,J}$ whose right/bottom boundaries have been changed. The list ${\Delta_\mathrm{B}}^{I,J}$ records in increasing order of j the position-value-value triplets $(j, D'[i^I_\mathrm{B}, j], D[i^I_\mathrm{B}, j - \ell])$ for all cells (i^I_B, j) on the bottom boundary of

$\mathcal{B}^{I,J}$ where the inequality $DR'[i^I_{\mathrm{B}}, j].L \neq DR[i^I_{\mathrm{B}}, j-\ell].L$ holds. In a similar manner, each list ${\Delta_{\mathrm{R}}}^{I,J}$ records in increasing order of i the position-value-value triplets $(i, D'[i, j^J_{\mathrm{R}}], D[i, j^J_{\mathrm{R}} - \ell])$ for all cells (i, j^J_{R}) on the right boundary of $\mathcal{B}^{I,J}$ where the inequality $DR'[i, j^J_{\mathrm{R}}].U \neq DR[i, j^J_{\mathrm{R}} - \ell].U$ holds. Note that the position-value pairs record concrete distance values $D'[i,j]$ and $D[i, j-\ell]$ instead of difference values. The positions may be accompanied by pointers to allow direct reference in a linked DS'. The lists ${\Delta_{\mathrm{B}}}^{I,J}$ and ${\Delta_{\mathrm{R}}}^{I,J}$ serve a similar purpose as $prev\Delta$ in HNI: when updating box $B^{I,J}$, we consult the lists ${\Delta_{\mathrm{B}}}^{I-1,J}$ and ${\Delta_{\mathrm{R}}}^{I,J-1}$ to deduce which cells on the topmost row $i^{I-1}_{\mathrm{B}} + 1$ and the leftmost column $j^{J-1}_{\mathrm{R}} + 1$ of $\mathcal{B}^{I,J}$ need to be recomputed based on Lemma 1.

We will also maintain a length-m array $DRcol$, a length-n array $DRrow$ and a length-M array TL for which the following conditions hold when we are processing a box $\mathcal{B}^{I,J}$:

- $DRcol[i] = (J-1, DR[i, j^{J-1}_{\mathrm{R}} - \ell].U)$, if the left boundary cell $(i, j^{J-1}_{\mathrm{R}})$ was changed when processing $\mathcal{B}^{I,J-1}$.
- $DRrow[j] = (I-1, DR[i^{I-1}_{\mathrm{B}}, j-\ell].L)$, if the top boundary cell $(i^{I-1}_{\mathrm{B}}, j)$ was changed when processing $\mathcal{B}^{I-1,J}$.
- $TL[I] = (D'[i^{I-1}_{\mathrm{B}}, j^{J-1}_{\mathrm{R}}], D[i^{I-1}_{\mathrm{B}}, j^{J-1}_{\mathrm{R}} - \ell])$, if at least one left boundary cell $(i, j^{J-1}_{\mathrm{R}})$ was changed when processing $\mathcal{B}^{I,J-1}$, or if at least one top boundary cell $(i^{I-1}_{\mathrm{B}}, j)$ was changed when processing $\mathcal{B}^{I-1,J}$.

$DRcol$ and $DRrow$ preserve the values of DR before their potential changes, and TL provides the top-left corner distance values for the current box.

Initially we assume that a left end modification to B potentially changes all rows in column 0 of DR': before the box column $J = 1$ is processed, each previous box column list ${\Delta_{\mathrm{R}}}^{I,0}$ is initialized to contain the triplets $(i, D'[i,0], D[i,-\ell])$ for $i = i^{I-1}_{\mathrm{B}} + 1, \ldots, i^I_{\mathrm{B}}$. In addition the array $DRcol$ is initialized with values $DRcol[i] = (0, DR[i, -\ell].U)$ for $i = 1, \ldots, m$, and the array TL is initialized with values $TL[I] = (D'[i^{I-1}_{\mathrm{B}}, 0], D[i^{I-1}_{\mathrm{B}}, -\ell])$ for $I = 1, \ldots, M$. All these initializations involve at most two first columns of D' or D and can thus be computed in $O(m)$ time using recurrence (1). The array $DRrow$ is initialized in $O(n)$ time with $(\texttt{null}, \texttt{null})$ values.

6.2.1 Processing a box $B^{I,J}$

When updating a box $\mathcal{B}^{I,J}$, we will perform the following tasks:

- Recompute all difference values on the right/bottom boundary that change.
- For each changed right boundary cell (i, j^J_{R}):
  - Insert the position-value-value triplet $(i, D'[i, j^J_{\mathrm{R}}], D[i, j^J_{\mathrm{R}} - \ell])$ into ${\Delta_{\mathrm{B}}}^{I,J}$.
  - Set $DRcol[i] = (J, DR[i, j^J_{\mathrm{R}} - \ell].U)$.
- For each changed bottom boundary cell (i^J_{B}, j):

  - Insert the position-value-value triplet $(j, D'[i_{\mathrm{B}}^{I}, j], D[i_{\mathrm{B}}^{I}, j-\ell])$ into ${\mathit{\Delta}_{\mathrm{R}}}^{I,J}$.
  - Set $\mathit{DRrow}[j] = (I, \mathit{DR}[i_{\mathrm{B}}^{I}, j-\ell].L)$.

- If at least one right boundary difference changed:
  - Set $TL[I] = (D'[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J}], D[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J} - \ell])$.
  - Add the box row index I to the end of $\mathit{currBox\Delta}$.

- If at least one bottom boundary cell changed and $I < M$:
  - Set $TL[I+1] = (D'[i_{\mathrm{B}}^{I}, j_{\mathrm{R}}^{J-1}], D[i_{\mathrm{B}}^{I}, j_{\mathrm{R}}^{J-1} - \ell])$.

If we are able to complete the first task of updating the changed boundary cells, the remaining task are trivial to incorporate into the process. Therefore we will concentrate on the first task. Fig. 5 depicts the initial situation when starting to process a box $B^{I,J}$. The grey cells mark changed cells on the left/top boundary, as specified by the lists ${\mathit{\Delta}_{\mathrm{B}}}^{I-1,J}$ and ${\mathit{\Delta}_{\mathrm{R}}}^{I,J-1}$. Our strategy is to traverse difference changes within the current box $B^{I,J}$ by depth-first-search (DFS). This partially resembles how the KP algorithm [12] traces changes in DR. The DFS will also maintain L-zone and T-zone min-deques `Ldeque`$'$, `Ldeque`, `Tdeque`$'$ and `Tdeque` that allow computing values of D' and D inside the current box $\mathcal{B}^{I,J}$. We start a separate DFS from each position listed in ${\mathit{\Delta}_{\mathrm{B}}}^{I-1,J}$ or ${\mathit{\Delta}_{\mathrm{R}}}^{I,J-1}$. Each DFS traces cells of DR' with difference changes as long as possible while still remaining inside the current box. The search advances according to the conditions of Lemma 1. If a DFS is currently in cell $\mathit{DR}'[i,j]$, it will first try to proceed one step right to the cell $\mathit{DR}'[i,j+1]$ if the condition $\mathit{DR}'[i,j].U \neq \mathit{DR}[i,j-\ell].U$ holds, and later one step down to the cell $\mathit{DR}'[i+1,j]$ if the condition $\mathit{DR}'[i,j].L \neq \mathit{DR}[i,j-\ell].L$ holds. Fig. 5 shows examples of the first step from the left column or top row.

When a DFS moves into a cell (i,j), we want to achieve the following five goals:

1. `Ldeque`$'$ will be set to represent L-zone values of D' for the cell (i,j).
2. `Ldeque` will be set to represent L-zone values of D for the cell $(i, j-\ell)$.
3. `Tdeque`$'$ will be set to represent T-zone values of D' for the cell (i,j).
4. `Tdeque` will be set to represent T-zone values of D for the cell $(i, j-\ell)$.
5. The values of D' and D will be known for the four cells $(i-1, j-1)$, $(i-1, j)$, $(i, j-1)$ and (i,j).

Let us first concentrate on the first four goals: updating the min-deques. Before a DFS begins, all four min-deques are initialized to be empty.

It is useful to first note that all values $\mathit{DR}'[i,j]$ and $\mathit{DR}[i,j-\ell]$ on the left/top boundary of the current box are available in $O(1)$ time, if we have a pointer to within $O(1)$ positions of the cell (i,j) in DS'. The values $\mathit{DR}'[i,j]$ are available directly in DS', as the top/left boundaries have already been updated. Each left boundary value $DR[i, j_{\mathrm{R}}^{J-1} - \ell]$ is stored in the second position of the pair $DRcol[i]$, if the first position of that pair is $J-1$, and otherwise the equality $DR[i, j_{\mathrm{R}}^{J-1} - \ell] = DR'[i, j_{\mathrm{R}}^{J-1}]$ holds (that boundary value did not change). In a similar manner each top boundary value $DR[i_{\mathrm{B}}^{I-1}, j - \ell]$ is stored in the second position of the pair $DRrow[j]$, if the first position of that pair is $I-1$, and otherwise the equality $DR[i_{\mathrm{B}}^{I-1}, j - \ell] = DR'[i_{\mathrm{B}}^{I-1}, j]$ holds.

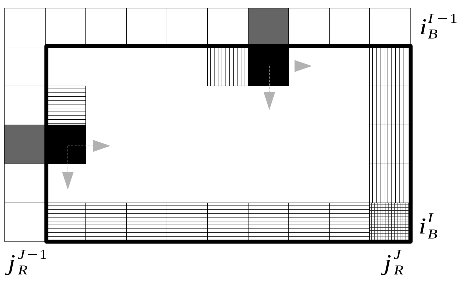

Figure 5: The first steps of a DFS when starting from the left or the top boundary. The current cell (i, j) is shown as black, and the previous cell in grey. The arrows show the possible directions for next steps.

In order to make the discussion more concise, we will discuss only the min-deques `Ldeque'` and `Tdeque'` in detail. The min-deques `Ldeque` and `Tdeque` will be subjected to otherwise identical operations but using values of form $D[i, j-\ell]$ or $DR[i, j-\ell]$ instead of $D'[i, j]$ or $DR'[i, j]$. A similar convention applies to the notion of L- and T-zones: discussion about an L- or T-zone cell (i, j) will refer to the cell $(i, j - \ell)$ in case of D. We start by considering the first step of a DFS. Fig. 5 illustrates the situation.

When the first step of a DFS is a right step from the left boundary cell $(i, j_{\mathrm{R}}^{J-1})$ to the cell $(i, j_{\mathrm{R}}^{J-1}+1)$, the list $\Delta_{\mathrm{R}}{}^{I,J-1}$ must have contained the triplet $(i, D'[i, j_{\mathrm{R}}^{J-1}], D[i, j_{\mathrm{R}}^{J-1} - \ell])$. The L-zone consists of the cells $(i-1, j_{\mathrm{R}}^{J-1})$ and $(i, j_{\mathrm{R}}^{J-1})$. All L-zone values become available after computing $D'[i-1, j_{\mathrm{R}}^{J-1}] = D'[i, j_{\mathrm{R}}^{J-1}] - DR'[i, j_{\mathrm{R}}^{J-1}].U$ and $D[i-1, j_{\mathrm{R}}^{J-1} - \ell] = D[i, j_{\mathrm{R}}^{J-1} - \ell] - DR[i, j_{\mathrm{R}}^{J-1} - \ell].U$. The T-zone cells are $(i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1})$ and $(i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1}+1)$. Now $TL[I]$ provides the values $D'[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1}]$ and $D[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1} - \ell]$, and the remaining T-zone values are computed as $D'[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1}+1] = D'[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1}] + DR'[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1}+1].L$ and $D[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1} + 1 - \ell] = D[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1} - \ell] + DR[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1} + 1 - \ell].L$. These values are inserted into the corresponding min-deques in accordance with Lemma 2. `Ldeque'` gets the values $D'[i-1, j_{\mathrm{R}}^{J-1}] + L_\delta(i, j_{\mathrm{R}}^{J-1}+1, i-1, j_{\mathrm{R}}^{J-1})$ and $D'[i, j_{\mathrm{R}}^{J-1}] + L_\delta(i, j_{\mathrm{R}}^{J-1}+1, i, j_{\mathrm{R}}^{J-1})$, and `Tdeque'` gets the values $D'[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1}] + T_\delta(i, j_{\mathrm{R}}^{J-1}+1, i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1})$ and $D'[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1}+1] + T_\delta(i, j_{\mathrm{R}}^{J-1}+1, i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1}+1)$.

When the first step of a DFS is a down step from the top boundary cell $(i_{\mathrm{B}}^{I-1}, j)$ to the cell $(i_{\mathrm{B}}^{I-1}+1, j)$, the list $\Delta_{\mathrm{B}}{}^{I-1,J}$ must have contained the triplet $(j, D'[i_{\mathrm{B}}^{I-1}, j], D[i_{\mathrm{B}}^{I-1}, j - \ell])$. The L-zone consists of the cells $(i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1})$ and $(i_{\mathrm{B}}^{I-1}+1, j_{\mathrm{R}}^{J-1})$. Again $TL[I]$ provides the values $D'[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1}]$ and $D[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1} - \ell]$, and the remaining L-zone values are computed as $D'[i_{\mathrm{B}}^{I-1}+1, j_{\mathrm{R}}^{J-1}] = D'[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1}] + DR'[i_{\mathrm{B}}^{I-1}+1, j_{\mathrm{R}}^{J-1}].U$ and $D[i_{\mathrm{B}}^{I-1}+1, j_{\mathrm{R}}^{J-1} - \ell] = D[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1} - \ell] + DR[i_{\mathrm{B}}^{I-1}+1, j_{\mathrm{R}}^{J-1} - \ell].U$. The T-zone consists of the cells $(i_{\mathrm{B}}^{I-1}, j-1)$ and $(i_{\mathrm{B}}^{I-1}, j)$. All T-zone values will become known after computing $D'[i_{\mathrm{B}}^{I-1}, j-1] = Dp[i_{\mathrm{B}}^{I-1}, j] - DR'[i_{\mathrm{B}}^{I-1}, j].L$ and $D[i_{\mathrm{B}}^{I-1}, j-1-\ell] = D[i_{\mathrm{B}}^{I-1}, j-\ell] - DR[i_{\mathrm{B}}^{I-1}, j - \ell].L$. `Ldeque'` gets the values $D'[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1}] + L_\delta(i_{\mathrm{B}}^{I-1}+1, j, i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^{J-1})$ and $D'[i_{\mathrm{B}}^{I-1}+1, j_{\mathrm{R}}^{J-1}] + L_\delta(i_{\mathrm{B}}^{I-1}+1, j, i_{\mathrm{B}}^{I-1}+1, j_{\mathrm{R}}^{J-1})$, and `Tdeque'` gets the values

$D'[i_{\mathrm{B}}^{I-1}, j-1] + T_\delta(i_{\mathrm{B}}^{I-1}+1, j, i_{\mathrm{B}}^{I-1}, j-1)$ and $D'[i_{\mathrm{B}}^{I-1}, j] + T_\delta(i_{\mathrm{B}}^{I-1}+1, j, i_{\mathrm{B}}^{I-1}, j)$.

In summary, all four min-deques $\texttt{Ldeque}'$, $\texttt{Ldeque}$, $\texttt{Tdeque}'$ and $\texttt{Tdeque}$ can be initialized in $O(1)$ time in the beginning of the first step to represent the L- and T-zone values of both D' and D. Later updates to the min-deques will again use a value readjusting mechanism in order to account for changes in the values $h(i, j, i^*, j^*)$, $v(i, j, i^*, j^*)$ and $d(i, j, i^*, j^*)$ as we move to compute a different cell. But because a DFS moves in a non-linear manner, we will maintain the two adjustment values, one per zone, in separate variables α and β. Initially $\alpha = \beta = 0$. The values Z_L and Z_T will always be computed as $Z_L = \texttt{Ldeque}'.\texttt{min}() + \alpha$ and $Z_T = \texttt{Tdeque}'.\texttt{min}() + \beta$.

Now consider how the L- and T-zones change if the DFS makes a later step into the cell (i, j). The situation is illustrated in Fig. 6. Throughout this discussion we assume that the values $s_L = j - j_{\mathrm{R}}^{J-1}$, $s_T = i - i_{\mathrm{B}}^{I-1}$ and $s = \min\{s_L, s_T\}$ have been computed to correspond to the current cell (i, j). We consider both forward and reverse steps, as the DFS can also backtrack.

The case of a right step from $(i, j-1)$ to (i, j): The cell $(i-s, j_{\mathrm{R}}^{J-1})$ is added to the front of L-zone if $s = s_L \le s_T$, and otherwise L-zone remains unchanged. The cell $(i_{\mathrm{B}}^{I-1}, j)$ is added to the end of T-zone, and if $s = s_T < s_L$, the cell $(i_{\mathrm{B}}^{I-1}, j-s-1)$ is removed from the front of T-zone. The adjustment values are updated to be $\alpha \leftarrow \alpha + \delta(\varepsilon, b)$ and $\beta \leftarrow \beta + \delta(a, b) - \delta(a, \varepsilon)$.

The reverse left step from $(i, j+1)$ to (i, j): if $s = s_L < s_T$, the cell $(i-s-1, j_{\mathrm{R}}^{J-1})$ is removed from the front of L-zone, and otherwise L-zone remains unchanged. The cell $(i_{\mathrm{B}}^{I-1}, j+1)$ is removed from the end of T-zone, and if $s = s_T \le s_L$, the cell $(i_{\mathrm{B}}^{I-1}, j-s)$ is added to the front of T-zone. The adjustment values are updated to be $\alpha \leftarrow \alpha - \delta(\varepsilon, b)$ and $\beta \leftarrow \beta - \delta(a, b) + \delta(a, \varepsilon)$.

The case of a down step from $(i-1, j)$ to (i, j): The cell $(i, j_{\mathrm{R}}^{J-1})$ is added to the end of L-zone, and if $s = s_L < s_T$, the cell $(i-s-1, j_{\mathrm{R}}^{J-1})$ is removed from the front of L-zone. If $s = s_T \le s_L$, the cell $(i_{\mathrm{B}}^{I-1}, j-s)$ is added to the front of T-zone, and otherwise T-zone remains unchanged. The adjustment values are updated to be $\alpha \leftarrow \alpha + \delta(a, b) - \delta(\varepsilon, b)$ and $\beta \leftarrow \beta + \delta(a, b) - \delta(a, \varepsilon)$.

The reverse up step from $(i+1, j)$ to (i, j): The cell $(i+1, j_{\mathrm{R}}^{J-1})$ is removed from the end L-zone, and if $s = s_L \le s_T$, the cell $(i-s, j_{\mathrm{R}}^{J-1})$ is added to the front of L-zone. If $s = s_T < s_L$, the cell $(i_{\mathrm{B}}^{I-1}, j-s-1)$ is removed from the front of T-zone, and otherwise T-zone remains unchanged. The adjustment values are updated to be $\alpha = \alpha - \delta(a, b) + \delta(\varepsilon, b)$ and $\beta = \beta - \delta(a, b) + \delta(a, \varepsilon)$.

Let $\texttt{deque}.\texttt{front}()$ and $\texttt{deque}.\texttt{end}()$ refer to the first and last values of a min-deque $\texttt{deque}$, respectively. We also assume that the adjustment values α and β have already been updated as described above. We now provide the details of how the min-deques can be updated to accommodate the zone changes described above. Removals are trivial, as removing a cell from the front or end of a zone corresponds to removing the front or end value of the corresponding min-deque. Therefore we consider only additions, which concern the following cells:

Adding $(i-s, j_{\mathrm{R}}^{J-1})$: The minimal path from (i, j) to $(i-s, j_{\mathrm{R}}^{J-1})$ uses only diagonal steps and has cost $s\delta(a, b)$. The minimal path from (i, j) to $(i-s+1, j_{\mathrm{R}}^{J-1})$ uses $s-1$ diagonal steps and one horizontal step, costing $\delta(\varepsilon, b) + (s-1)\delta(a, b)$. Therefore $\texttt{Ldeque}'.\texttt{front}()$ corresponds to $D'[i-s+$

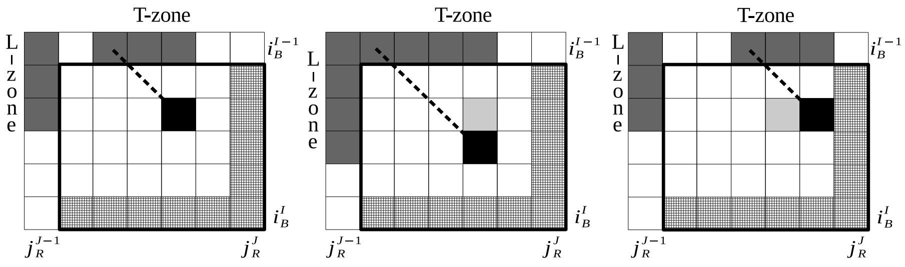

Figure 6: An example of how the zones change upon a down or right step. The black cell is the current position. The light grey cell is the previous position.

$1, j_{\mathrm{R}}^{J-1}] + \delta(\varepsilon, b) + (s-1)\delta(a,b) - \beta$, and we would like to add the value $D'[i - s, j_{\mathrm{R}}^{J-1}] + s\delta(a,b) - \beta$ for the new cell $(i-s, j_{\mathrm{R}}^{J-1})$. This is achieved by adding the value $\texttt{Ldeque}'.\texttt{front}() + \delta(a,b) - \delta(\varepsilon,b) - DR'[i-s+1, j_{\mathrm{R}}^{J-1}].U$ to the front of $\texttt{Ldeque}'$.

Adding $(i, j_{\mathrm{R}}^{J-1})$: The minimal path from (i,j) to $(i, j_{\mathrm{R}}^{J-1})$ uses only horizontal steps and has cost $s_L\delta(\varepsilon,b)$. The minimal path from (i,j) to $(i-1, j_{\mathrm{R}}^{J-1})$ uses $s_L - 1$ horizontal steps and one diagonal step, costing $(s_L-1)\delta(\varepsilon,b) + \delta(a,b)$. Therefore $\texttt{Ldeque}'.\texttt{end}()$ corresponds to $D'[i-1, j_{\mathrm{R}}^{J-1}] + (s_L-1)\delta(\varepsilon,b) + \delta(a,b) - \beta$, and we would like to add the value $D'[i, j_{\mathrm{R}}^{J-1}] + s_L\delta(\varepsilon,b) - \beta$ for the new cell $(i, j_{\mathrm{R}}^{J-1})$. This is achieved by adding the value $\texttt{Ldeque}'.\texttt{end}() - \delta(a,b) + \delta(\varepsilon,b) + DR'[i, j_{\mathrm{R}}^{J-1}].U$ to the end of $\texttt{Ldeque}'$.

Adding $(i_{\mathrm{B}}^{I-1}, j-s)$: The minimal path from (i,j) to $(i_{\mathrm{B}}^{I-1}, j-s)$ uses only diagonal steps and has cost $s\delta(a,b)$. The minimal path from (i,j) to $(i_{\mathrm{B}}^{I-1}, j-s+1)$ uses $s-1$ diagonal steps and one vertical step, costing $\delta(a,\varepsilon) + (s-1)\delta(a,b)$. Therefore $\texttt{Tdeque}'.\texttt{front}()$ corresponds to $D'[i_{\mathrm{B}}^{I-1}, j-s+1] + \delta(a,\varepsilon) + (s-1)\delta(a,b) - \beta$, and we would like to add the value $D'[i_{\mathrm{B}}^{I-1}, j-s] + s\delta(a,b) - \beta$ for the new cell $(i_{\mathrm{B}}^{I-1}, j-s)$. This is achieved by adding the value $\texttt{Tdeque}'.\texttt{front}() + \delta(a,b) - \delta(a,\varepsilon) - DR'[i_{\mathrm{B}}^{I-1}, j-s+1].L$ to the front of $\texttt{Tdeque}'$.

Adding $(i_{\mathrm{B}}^{I-1}, j)$: The minimal path from (i,j) to $(i_{\mathrm{B}}^{I-1}, j)$ uses only vertical steps and has cost $s_T\delta(a,\varepsilon)$. The minimal path from (i,j) to $(i_{\mathrm{B}}^{I-1}, j-1)$ uses $s_T - 1$ vertical steps and one diagonal step, costing $(s_T-1)\delta(a,\varepsilon) + \delta(a,b)$. Therefore $\texttt{Tdeque}'.\texttt{end}()$ corresponds to $D'[i_{\mathrm{B}}^{I-1}, j-1] + (s_T-1)\delta(a,\varepsilon) + \delta(a,b) - \beta$, and we would like to add the value $D'[i_{\mathrm{B}}^{I-1}, j] + s_T\delta(a,\varepsilon) - \beta$ for the new cell $(i_{\mathrm{B}}^{I-1}, j)$. This is achieved by adding the value $\texttt{Tdeque}'.\texttt{end}() - \delta(a,b) + \delta(a,\varepsilon) + DR'[i_{\mathrm{B}}^{I-1}, j].L$ to the end of $\texttt{Ldeque}'$.

The preceding min-deque updates take $O(1)$ amortized time. Let us now consider the fifth goal: that of ensuring that the values of D' and D are known for the four cells $(i-1, j-1)$, $(i-1, j)$, $(i, j-1)$ and (i,j).

We will first note in an inductive manner that the cells $(i-1, j-1)$ and $(i, j-1)$ will be known already, if the current step moved right from $(i, j-1)$ to (i,j), and the cells $(i-1, j-1)$ and $(i-1, j)$ will be known already, if the current step moved down from $(i-1, j)$ to (i,j).

If the first step was a right step from the left boundary cell $(i, j_{\mathrm{R}}^{J-1})$ to the cell $(i, j_{\mathrm{R}}^{J-1}+1)$, then the list ${\Delta_{\mathrm{R}}}^{I,J-1}$ provides the previous column values $D'[i, j_{\mathrm{R}}^{J-1}]$ and $D[i, j_{\mathrm{R}}^{J-1}-\ell]$ directly, and the values $D'[i-1, j_{\mathrm{R}}^{J-1}]$ and $D[i-1, j_{\mathrm{R}}^{J-1}-\ell]$ can be computed as $D'[i-1, j_{\mathrm{R}}^{J-1}] = D'[i, j_{\mathrm{R}}^{J-1}] - DR'[i, j_{\mathrm{R}}^{J-1}].U$ and $D[i-1, j_{\mathrm{R}}^{J-1}-\ell] = D[i, j_{\mathrm{R}}^{J-1}-\ell] - DR[i, j_{\mathrm{R}}^{J-1}-\ell].U$.

If the first step was a down step from the top boundary cell $(i_{\mathrm{B}}^{I-1}, j)$ to the cell $(i_{\mathrm{B}}^{I-1}+1, j)$, then the list ${\Delta_{\mathrm{B}}}^{I-1,J}$ provides the previous row values $D'[i_{\mathrm{B}}^{I-1}, j]$ and $D[i_{\mathrm{B}}^{I-1}, j-\ell]$ directly, and the values $D'[i_{\mathrm{B}}^{I-1}, j-1]$ and $D[i_{\mathrm{B}}^{I-1}, j-1-\ell]$ can be computed as $D'[i_{\mathrm{B}}^{I-1}, j-1] = D'[i_{\mathrm{B}}^{I-1}, j] - DR'[i_{\mathrm{B}}^{I-1}, j].L$ and $D[i_{\mathrm{B}}^{I-1}, j-1-\ell] = D[i_{\mathrm{B}}^{I-1}, j-\ell] - DR[i_{\mathrm{B}}^{I-1}, j-\ell].L$.

Therefore the claim holds initially. Also, if each previous step has fulfilled the fifth goal, then the claim holds also for the current (i, j). Fulfilling the fifth goal also for the current cell (i, j) requires us to compute values for two unknown cells: either the cells $(i-1, j)$ and (i, j) or the cells $(i, j-1)$ and (i, j).

As the min-deques are already updated for the cell (i, j), the values $D'[i, j]$ and $D[i, j-\ell]$ can be computed as $D'[i, j] = \min\{\texttt{Ldeque}'.\texttt{min}()+\alpha, \texttt{Tdeque}'.\texttt{min}()+\beta\}$ and $D[i, j\ell] = \min\{\texttt{Ldeque}.\texttt{min}()+\alpha, \texttt{Tdeque}.\texttt{min}()+\beta\}$. Let (i', j') be the single remaining unknown cell: it is either $(i, j-1)$ or $(i-1, j)$. The cell (i', j') can be handled by making a temporary reverse step into it. This step is either a reverse left step from (i, j) to $(i, j-1)$, or a reverse up step from (i, j) to $(i-1, j)$. Once the temporary step has updated the min-deques and the adjusting values α and β, the values for the cell (i', j') are computed as $D'[i', j'] = \min\{\texttt{Ldeque}'.\texttt{min}()+\alpha, \texttt{Tdeque}'.\texttt{min}()+\beta\}$ and $D[i', j'\ell] = \min\{\texttt{Ldeque}.\texttt{min}()+\alpha, \texttt{Tdeque}.\texttt{min}()+\beta\}$. Then we make a normal right or down step from (i', j') back to (i, j). The whole process takes $O(1)$ amortized time.

Before performing a forward step to the right or down, we let the DFS store the distance values of the current cells $(i-1, i-1)$, $(i-1, j)$, $(i, j-1)$ and (i, j) onto a stack. This ensures that the preceding invariant about two known values will hold also for the next step that moves right or down from (i, j); it can look up the values from the top of the stack. Fig. 5 illustrates the four neighboring cells involved in the process. Also note that each backtracking step will update the min-deques and adjusting values α and β in $O(1)$ time according to the rules given for the reverse left and up steps, and read and remove the last stored distance values from the top of the stack.

In summary, all five goals of a DFS step can be achieved in $O(1)$ amortized time. Let us now turn back to the high level organization of the DFS. We will perform a separate DFS search from each position listed in ${\Delta_{\mathrm{B}}}^{I-1,J}$ or ${\Delta_{\mathrm{R}}}^{I,J-1}$ in such order, that first the positions in the top boundary list ${\Delta_{\mathrm{B}}}^{I-1,J}$ are processed in reversed order of decreasing j, and after that the position in the left boundary list ${\Delta_{\mathrm{R}}}^{I,J-1}$ are processed in the normal increasing order of i. We maintain a length-n table *bottom* where the value *bottom*$[j]$ tells the largest row index i that any DFS has visited in column j. The table is initialized with zeros before any boxes have been processed, and when a forward DFS step reaches a cell (i, j), the update *bottom*$[j] = i$ is performed.

When a DFS is currently in the cell (i, j) and has the values of D' and D in the four neighboring cells $(i-1, j-1)$, $(i-1, j)$, $(i, j-1)$ and (i, j) available, the differences $DR'[i,j].U$, $DR'[i,j].L$, $DR[i,j-\ell].U$ and $DR[i,j-\ell].L$ can be computed and possible boundary changes recorded. If the current cell resides on the right or bottom boundary, the DFS will update ${\Delta_{\mathrm{B}}}^{I,J}$, ${\Delta_{\mathrm{R}}}^{I,J}$, *DRcol* and *DRrow* as described in the beginning of this section. One delicacy is that each right boundary position-value-value triplet will be added to the end of the list ${\Delta_{\mathrm{R}}}^{I,J}$, and each bottom boundary position-value-value triplet is added to the front of ${\Delta_{\mathrm{B}}}^{I,J}$. The reason for this order becomes clear soon.

The DFS decides according to Lemma 1 if it should proceed right and/or down, or if the next step must backtrack. If the current cell is on the right/bottom boundary, the DFS will always backtrack. Otherwise the DFS will make a right step to the cell (i, j) if and only if $DR'[i,j].U \neq DR[i,j-\ell].U$ and $bottom[j+1] < i$, and (possibly after backtracking from the right) a down step to the cell $(i+1, j)$ if and only if $DR'[i,j].L \neq DR[i,j-\ell].L$.

The table *bottom* prevents two different DFS searches from visiting the same cell (i, j). The order in which we start the different DFS searches from the left/top boundary ensures that if a previous DFS has visited a cell (i, j), then no column j cell above it will need to be visited. The previous DFS traversed a path from some (i^*, j^*) to (i, j), and the start point (i', j') of the current DFS must have either $j' < j^*$ or $j' = j^*$ and $i' < i^*$. In both cases a path from (i', j') to a column j cell above (i, j) would meet the earlier DFS path from (i^*, j^*) to (i, j), making the path redundant from that meeting point onwards. A similar argument also guarantees that the DFS searches will build the lists ${\Delta_{\mathrm{B}}}^{I,J}$ and ${\Delta_{\mathrm{R}}}^{I,J}$ in correct order. Once a DFS has reached a right boundary cell (i, j_{R}^J), the update $bottom[j_{\mathrm{R}}^j] = i$ will prevent redundant visits to the cells above it. And in the same way, once a DFS has reached a bottom boundary cell (i_{B}^J, j), the update $bottom[j] = i_{\mathrm{B}}^J$ will prevent redundant visits to any cell to the right.

Once all DFS searches for the current box have ended, we still need to update the *currBoxΔ* and *TL*, as described in the beginning of this section. The case of *currBoxΔ* is trivial, so consider *TL*. We need to store into $TL[i_{\mathrm{B}}^I]$ the bottom left corner values $(D'[i_{\mathrm{B}}^I, j_{\mathrm{R}}^{J-1}], D[i_{\mathrm{B}}^I, j_{\mathrm{R}}^{J-1} - \ell])$, if a bottom boundary cell changed, and into $TL[i_{\mathrm{B}}^{I-1}]$ the top right corner values $(D'[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^J]$ and $D[i_{\mathrm{B}}^{I-1}, j_{\mathrm{R}}^J - \ell])$, if a right boundary cell changed. The bottom left corner values are computed by starting from the concrete distance values provided by the last triplet of the list ${\Delta_{\mathrm{R}}}^{I,J-1}$, or the top-left corner distances still given by $TL[i_{\mathrm{B}}^{I-1}]$, if ${\Delta_{\mathrm{R}}}^{I,J-1}$ is empty. Let this starting cell be $(i^*, j_{\mathrm{R}}^{J-1})$: the bottom left corner distances can be computed in $i_{\mathrm{B}}^J - i^*$ steps by using the difference values $DR'[i, j_{\mathrm{R}}^{J-1}]$ and $DR[i, j_{\mathrm{R}}^{J-1}]$. The cost of these $i_{\mathrm{B}}^I - i^*$ steps can be charged to the DFS that reached the bottom boundary: its start row must have been at or above row i^*. The top right corner values are handled in similar way: starting from the concrete distance values provided by the last triplet of the list ${\Delta_{\mathrm{B}}}^{I-1,J}$, or the top-left corner distances in $TL[i_{\mathrm{B}}^{I-1}]$, if ${\Delta_{\mathrm{B}}}^{I-1,J}$ is empty. Let this starting cell be $(i_{\mathrm{B}}^{I-1}, j^*)$: the top right corner distances can be computed in $j_{\mathrm{R}}^J - j^*$ steps by using the difference values $DR'[i_{\mathrm{B}}^{I-1}, j]$ and $DR[i_{\mathrm{B}}^{I-1}, j]$. The cost of these $j_{\mathrm{R}}^J - j^*$ steps can be charged to the DFS that reached the right boundary: its start column must have been at or to the left of column j^*.

Let $\#^{I,J}$ be the number of cells in box $\mathcal{B}^{I,J}$ of DR' that differ from DR. The overall number of cells visited by the DFS searches is $\Theta(\#^{I,J})$, as the DFS searches proceed only to changed cells and no cell is visited more than $O(1)$ times. The work per DFS step is $O(1)$ amortized time. Hence the overall work per box is $\Theta(\#^{I,J})$. is $O(1)$ amortized time. Hence the overall work per box is $\Theta(\#^{I,J})$.

Theorem 3 *Let c be the maximum weight of the cost function δ. DS can be updated to DS' in $O(c(m+n))$ time.*

Proof The initializing step takes $O(m+n)$ time. Then, the theorem follows from Theorem 1 and the fact that each box can be processed in $O(\#^{I,J})$ time. $\square$

7 Conclusions

In this paper, we proposed a compact representation of the dynamic edit distance table of $\Theta(mN+Mn)$ space which can be updated in $O(c(m+n))$ time per left or right modification under weighted costs, where m, n, M, N are respectively the lengths of strings A and B to compare, and the sizes of the RLEs of A and B, and c is the maximum weight of the cost function. This generalizes the result of the previous version [10] of this paper which deals only with the unit costs, and reduces the space usage of the algorithm by Hyyrö et al. [11] which uses $\Theta(mn)$ space for dynamic edit distance tables with weighted costs. We remark that our proposed method is asymptotically as fast as these previous methods.

References

[1] H. Ann, C. Yang, C. Tseng and C. Hor, A fast and simple algorithm for computing the longest common subsequence of run-length encoded strings, *Inf. Process. Lett.* **108**(6) (2008) 360–364.

[2] A. Apostolico, G. M. Landau and S. Skiena, Matching for run-length encoded strings, *J. Complexity* **15**(1) (1999) 4–16.

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

[4] C. Barton, C. S. Iliopoulos and S. P. Pissis, Average-case optimal approximate circular string matching, *Proc. LATA 2015*, (2015), pp. 85–96.

[5] H. Bunke and J. Csirik, An improved algorithm for computing the edit distance of run-length coded strings, *Inf. Process. Lett.* **54**(2) (1995) 93–96.

[6] K. Chen and K. Chao, A fully compressed algorithm for computing the edit distance of run-length encoded strings, *Algorithmica* **65**(2) (2013) 354–370.

[7] H. Gajewska and R. E. Tarjan, Deques with heap order, *Inf. Process. Lett.* **22**(4) (1986) 197–200.

[8] P. Hsu, K. Chen and K. Chao, Finding all approximate gapped palindromes, *Int. J. Found. Comput. Sci.* **21**(6) (2010) 925–939.

[9] G. Huang, J. J. Liu and Y. Wang, Sequence alignment algorithms for run-length-encoded strings, *COCOON 2008*, (2008), pp. 319–330.

[10] H. Hyyrö and S. Inenaga, Compacting a dynamic edit distance table by RLE compression, *SOFSEM 2016*, (2016), pp. 302–313.

[11] H. Hyyrö, K. Narisawa and S. Inenaga, Dynamic edit distance table under a general weighted cost function, *J. Disc. Algo.* **34** (2015) 2–17.

[12] S.-R. Kim and K. Park, A dynamic edit distance table, *J. Disc. Algo.* **2** (2004) 302–312.

[13] P. Krusche and A. Tiskin, String comparison by transposition networks, *Proc. London Algorithmics Workshop 2008*, *Texts in Algorithmics* **11** (2009), pp. 184–204.

[14] G. M. Landau, E. W. Myers and J. P. Schmidt, Incremental string comparison, *SIAM J. Comp.* **27**(2) (1998) 557–582.

[15] V. Mäkinen, G. Navarro and E. Ukkonen, Approximate matching of run-length compressed strings, *Algorithmica* **35**(4) (2003) 347–369.

[16] Y. Sakai, Computing the longest common subsequence of two run-length encoded strings, *ISAAC 2012*, (2012), pp. 197–206.

[17] J. P. Schmidt, All highest scoring paths in weighted grid graphs and their application in finding all approximate repeats in strings, *SIAM J. Comp.* **27**(4) (1998) 972–992.