# $\tilde{O}$ptimal Dynamic Time Warping on Run-Length Encoded Strings*

Itai Boneh† Shay Golan‡ Shay Mozes§ Oren Weimann¶

**Abstract**

Dynamic Time Warping (DTW) distance is the optimal cost of matching two strings when extending runs of letters is for free. Therefore, it is natural to measure the time complexity of DTW in terms of the number of runs $n$ (rather than the string lengths $N$).

In this paper, we give an $\tilde{O}(n^2)$ time algorithm for computing the DTW distance. This matches (up to log factors) the known (conditional) lower bound, and should be compared with the previous fastest $O(n^3)$ time exact algorithm and the $\tilde{O}(n^2)$ time approximation algorithm. Our method also immediately implies an $\tilde{O}(nk)$ time algorithm when the distance is bounded by $k$. This should be compared with the previous fastest $O(n^2k)$ and $O(Nk)$ time exact algorithms and the $\tilde{O}(nk)$ time approximation algorithm.

*This paper appeared in Highlights of Algorithms (HALG) 2023.

†Reichman University and University of Haifa, Israel, itai.bone@biu.ac.il

‡Reichman University and University of Haifa, Israel, golansh1@macs.biu.ac.il

§Reichman University, Israel, smozes@idc.ac.il

¶University of Haifa, Israel, oren@cs.haifa.ac.il

# 1 Introduction

Dynamic Time Warping (DTW) [38] is one of the most popular methods for comparing time-series (see e.g. [2,5,8,24,26,29,32,39,42]). It is appealing in numerous applications such as bioinformatics, signature verification, and speech recognition, where two time-series can vary in speed but still be considered similar. For example, in speech recognition, DTW can detect similarities even if one person is talking faster than the other.

To define DTW, recall that a run-length encoding $S = s_1^{\ell_1} s_2^{\ell_2} \cdots s_n^{\ell_n}$ of a string $S$ over an alphabet $\Sigma$ is a concise (length $n$) representation of the (length $N = \sum_i \ell_i$) string $S$. Here $s_i^{\ell_i}$ denotes a letter $s_i \in \Sigma$ repeated $\ell_i$ times. For example, the string $S = aaaabbbaaaaa$ is encoded as $a^4b^3a^5$. A string $S' = s_1^{\ell'_1} s_2^{\ell'_2} \cdots s_n^{\ell'_n}$ is a *time-warp* of string $S = s_1^{\ell_1} s_2^{\ell_2} \cdots s_n^{\ell_n}$ if every $\ell'_i \geq \ell_i$.

**Definition 1** (Dynamic Time Warping)**.** *For a function $\delta : \Sigma^2 \to \mathbb{R}^+$, the Dynamic Time Warping distance of two strings $S$ and $T$ over alphabet $\Sigma$ is defined as*

$$\mathsf{DTW}(S,T) = \min_{|S'|=|T'|} \sum_{i=1}^{|S'|} \delta(S'[i], T'[i]),$$

*where $S'$ and $T'$ range over all time-warps of $S$ and $T$ respectively.*

In 1968, Vintzyuk [38] gave an $O(MN)$ time dynamic programming algorithm for computing the DTW of two strings $S$ and $T$ of lengths $N$ and $M$ respectively. His algorithm is one of the earliest uses of dynamic programming and is taught today in basic algorithms courses and textbooks. Apart from logarithmic factor improvements [16], the $O(MN)$ quadratic time complexity remains the fastest known and a strongly subquadratic-time $O((MN)^{1-\varepsilon})$ algorithm is unlikely as it would refute the popular Strong Exponential Time Hypothesis (SETH) [3,9].

The complexity of DTW in terms of $N$ and $M$ is thus well understood. Special cases of DTW are also well understood. These include DTW on binary strings [22,37], approximation algorithms [4,21,41], the low distance regime [21], sparse inputs [19,30,31], and reductions to other similarity measures [21,35,36]. However, the complexity of DTW is not yet resolved in terms of $n$ and $m$ (the run-length encoding sizes of $S$ and $T$ respectively). Namely, in the (especially appealing) case where the strings contain long runs. The currently fastest algorithms are $O(Nm + Mn)$ [12,14,21] and $O(n^2m + m^2n)$ [14]. In particular, an $\tilde{O}(nm)$ time algorithm is only known to be possible if we are willing to settle for a $(1+\varepsilon)$-approximation [40]. It remained an open question whether it is possible to obtain an exact $\tilde{O}(nm)$ algorithm (which is optimal up to log factors). In this paper we answer this open question in the affirmative.

**Prior work on DTW.** The classical dynamic programming for DTW is as follows. Let $\mathsf{DTW}(i,j) = \mathsf{DTW}(S[1\ldots i], T[1\ldots j])$, then $\mathsf{DTW}(0,0) = 0$, $\mathsf{DTW}(i,0) = \mathsf{DTW}(0,j) = \infty$ for every $i > 0$ and $j > 0$, and otherwise:

$$\mathsf{DTW}(i,j) = \delta(S[i], T[j]) + \min \begin{cases} \mathsf{DTW}(i-1,j) \\ \mathsf{DTW}(i,j-1) \\ \mathsf{DTW}(i-1,j-1) \end{cases} \tag{1}$$

The above dynamic programming is equivalent to a single-source shortest path (SSSP) computation in the following grid graph. We denote $[n] = \{1, 2, \ldots, n\}$.

**Definition 2** (The Alignment Graph)**.** *The alignment graph of $S$ and $T$ is a directed weighted graph $G$ with vertices $V = [0 \dots N] \times [0 \dots M]$. Every vertex $(i, j) \in [N] \times [M]$ has three entering edges, all with weight $\delta(S[i], T[j])$: A vertical edge from $(i-1, j)$, a horizontal edge from $(i, j-1)$, and a diagonal edge from $(i-1, j-1)$.*

We denote the distance from vertex $(0, 0)$ to $(i, j)$ as $\mathsf{dist}(i, j)$.[1] Clearly, $\mathsf{DTW}(i, j) = \mathsf{dist}(i, j)$. Therefore, $\mathsf{DTW}(S, T) = \mathsf{dist}(N, M)$ and can be computed in $O(MN)$ time by an SSSP algorithm (that explicitly computes the distances from $(0, 0)$ to all the $O(MN)$ vertices of the graph). The way to beat $O(MN)$ is to only compute distances to a subset of vertices.

Namely, partition the alignment graph into *blocks* where each block is the subgraph corresponding to a single run in $S$ and a single run in $T$. Then, proceed block-by-block and for each block compute its *output* (the last row and last column) given its *input* (the last row of the block above and the last column of the block to the left). Since blocks are highly regular (i.e., all edges inside a block have the same weight), it is not difficult to compute the output in time linear in the size of the output. Since the total size of all outputs (and all inputs) is $O(Nm + Mn)$, this leads to an overall $O(Nm + Mn)$ time algorithm [12, 14, 21].

In order to go below $O(Nm + Mn)$, in [14] it was observed that we do not really need to compute the entire output. It suffices to compute only the intersection of the output with a set of $O(mn)$ diagonals. Specifically, each block contributes one diagonal starting in its top-left corner, so there are overall $O(mn)$ diagonals and each diagonal intersects with $O(m+n)$ blocks. This leads to an $O(n^2m + m^2n)$ time algorithm. In [40] it was shown that if we are willing to settle for a $(1 + \varepsilon)$-approximation, then it suffices to compute only $\tilde{O}(1)$ output values per block.

**Prior work on Edit distance.** There are many similarities between DTW and the edit distance problem: (1) like DTW, edit distance can be computed in $O(MN)$ time using the alignment graph [34, 38]. The only difference is in the edge-weights. (2) like DTW, edit distance has a lower bound prohibiting strongly subquadratic time algorithms conditioned on the SETH [3, 9, 21], and (3) like DTW, edit distance can be computed in $O(Nm + Mn)$ time by proceeding block-by-block and computing the outputs from the inputs. However, unlike DTW, it is known how to compute the edit distance of run-length encoded strings in $\tilde{O}(nm)$ time [6, 7, 10–12, 18, 25, 27, 28]. Specifically, Clifford et. al. [12] showed that the input and output of a block can be implicitly represented by a piecewise linear function, and, that the representation of the output can be computed in amortized $O(\mathsf{polylog}(mn))$ time from the representation of the input. This implies an $\tilde{O}(nm)$ time algorithm for edit distance.

In [40], Xi and Kuszmaul write about the prospects of obtaining an $\tilde{O}(nm)$ time algorithm for DTW: "*Such an algorithm would finally unify edit distance and DTW in the run-length-encoded setting*".

**Our result and techniques.** We present an $\tilde{O}(nm)$ time algorithm for DTW. This is optimal up to logarithmic factors under the SETH. Our algorithm is independent of the alphabet size $|\Sigma|$ and of the function $\delta$. In fact, $\delta$ need not even satisfy the triangle inequality.

We follow the approach for edit distance by Clifford et. al. [12] of representing and manipulating inputs and outputs with a piecewise-linear function. However, the manipulation is more challenging for several reasons which were highlighted by Xi and Kuszmaul [40]: (1) unlike edit distance, DTW does not satisfy the triangle inequality. (2) we are interested in arbitrary cost functions $\delta$ for DTW, whereas the $\tilde{O}(nm)$ algorithm for edit distance [12] works only for Levenshtein distance

[1] Abusing notation, we will later also use $\mathsf{dist}((x, y), (x', y'))$ to denote the distance from vertex $(x, y)$ to vertex $(x', y')$.

(when $\delta(\cdot,\cdot) \in \{0,1\}$). (3) in the standard setting (i.e. not the run-length encoded setting) edit distance actually reduces to DTW [21].

In Section 2, we show that the required manipulation of inputs and outputs naturally reduces to $O(nm)$ operations on a data structure that, given an array $A$ of size $M+N$ initialized to all zeros, supports the following range operations:

**Definition 3** (Range Operations)**.**

- $\mathsf{Lookup}(i)$ - *return* $A[i]$.
- $\mathsf{AddConst}(i,j,c)$ - *for every* $k \in [i \dots j]$, *set* $A[k] \leftarrow A[k] + c$.
- $\mathsf{AddGradient}(i,j,g)$ - *for every* $k \in [i \dots j]$, *set* $A[k] \leftarrow A[k] + k \cdot g$.
- $\mathsf{LeftLinearWave}(i,j,\alpha)$ - *for every* $k \in [i \dots j]$, *set* $A[k] \leftarrow \min_{t \in [i \dots k]} \big(A[t] + (k-t)\alpha\big)$.
- $\mathsf{RightLinearWave}(i,j,\alpha)$ - *for every* $k \in [i \dots j]$, *set* $A[k] \leftarrow \min_{t \in [k \dots j]} \big(A[t] + (t-k)\alpha\big)$.

In Section 3, we show our main technical contribution:

**Theorem 1.** *Performing $s$ range operations of Definition 3 can be done in amortized $O(\mathsf{polylog}(s))$ time per operation.*

The proof of Theorem 1 can be roughly described as follows: We represent the array $A$ by the line segments of the linear interpolation of $A$. This way, the range operations of Definition 3 translate to creating and deleting segments, changing their slopes, and shiftings segments up and down. For most operations, these changes apply to a single contiguous range of $A$ and are therefore quite simple to implement in polylog time. The difficult operations are LeftLinearWave and RightLinearWave. These operations may need to replace each of $\Omega(n)$ different sets of consecutive segments with a single new segment. We refer to the process of replacing a set of consecutive segments with a single new segment as a *ray shooting* process. Shooting each of these rays separately would be too costly. More accurately, a ray shooting process that replaces many segments with a single one is not problematic since its cost can be charged to the decrease in the number of segments. The challenge is in shooting rays that replace a single segment with another one, as this does not decrease the number of segments.

Our main technical contribution is a sophisticated lazy approach for handling the problematic ray shooting processes. We study the structural properties of ray shooting processes, and characterize long rays which we can afford to shoot explicitly, and short rays, which we cannot. The structure we identify allows us to divide the segments representing $A$ into mega-segments, and keep track of a single pending short ray in each mega-segment such that executing the pending ray shooting process in each mega-segment would result in the correct representation of the array $A$. While we cannot afford to actually carry out all of these pending ray shooting processes, we can afford to perform the process locally, e.g., in order to support Lookup for a specific element of $A$, or to facilitate the other range operations.

One component of our lazy approach is a data structure (sometimes called *Segment tree beats* in programming competitions) for the following problem: Maintain an array $A$ under lookup queries and two kinds of update: $\mathsf{AddConst}(i,j,c)$ - for every $k \in [i \dots j]$ set $A[k] \leftarrow A[k]+c$, and $\mathsf{Min}(i,j,c)$ - for every $k \in [i \dots j]$ set $A[k] \leftarrow \min\{A[k], c\}$. Though we are not aware of any official publication, it is known (see e.g. [1]) that this problem can be solved in amortized polylog time. In Section 4 we show a different and *worst-case* polylog time solution.[2]

[2]We note that the solution in [1] also supports range-sum queries and for such a conditional lower bound (from

**Low regime DTW.** In Section 5, we show that our $\tilde{O}(n \cdot m)$ algorithm for computing $\mathsf{DTW}(S,T)$ immediately implies an $\tilde{O}(n \cdot k)$ time algorithm where $k = \mathsf{DTW}(S,T)$. This is useful when $k$ is small. It is achieved using the standard trick of limiting the computation to blocks that are in the $k$-neighborhood of the alignment graph's main diagonal. It improves the $O(N \cdot k)$ algorithm of [21], the $\tilde{O}(n^2 \cdot k)$ algorithm of [14], and the $\tilde{O}(n \cdot k)$ time approximation algorithm of [40] (all obtained with the same $k$-neighborhood idea).

We note that for the closely related problem of low regime *edit distance*, using the same $k$-neighborhood idea, the algorithm of [12] runs in $\tilde{O}(n \cdot k)$ time (now $k$ is the edit distance between $S$ and $T$). However, unlike DTW, there is a vast literature on low regime edit distance (and the approximation algorithms inspired by it). Most notable are the celebrated $O(N + k^2)$ time algorithms of Myers [33] and Landau-Vishkin [23] for unweighted edit distance, and the very recent $O(N + k^5)$ time algorithm for weighted edit distance [13].

**Pattern matching DTW.** The pattern matching version of DTW asks to compute, for every index $j \in [1 \dots |T|]$ the value $\min_{i \in [1 \dots j]}(\mathsf{DTW}(S, T[i \dots j]))$. In [17], an $O(NM)$ algorithm was presented for pattern matching DTW. Additionally, they provided an $O(nmk)$ algorithm for the low regime version of the problem, in which the goal is to report every index $j$ such that $\min_{i \in [1 \dots j]}(\mathsf{DTW}(S, T[i \dots j])) \leq k$. The key ingredient of these algorithms (See [17, Lemma 2]) is a dynamic programming formula that is identical to Eq. (1), except for the initialization. Since our $\tilde{O}(nm)$ algorithm for DTW is obtained by implementing the dynamic programming implicitly, by changing the initialization step, our algorithm implies an $\tilde{O}(nm)$ time algorithm for pattern matching DTW. This improves upon both the $O(NM)$ algorithm for DTW pattern matching and the $O(nmk)$ algorithm for the low regime DTW pattern matching (when $k$ is super poly-logarithmic).

## 2 DTW via Range Operations

In this section we prove that Theorem 1 implies an $\tilde{O}(nm)$ algorithm for DTW. Namely, that DTW reduces to efficiently supporting the range operations of Definition 3.

**Blocks in the alignment graph.** Let $S[i_1 \dots i_2]$ and $T[j_1 \dots j_2]$ be the $i$'th run in $S$ and the $j$'th run in $T$ respectively. The block $B_{i,j}$ in the alignment graph is the set of vertices $(a,b)$ with $a \in [i_1 \dots i_2]$ and $b \in [j_1 \dots j_2]$. All of the edges entering any vertex in block $B_{i,j}$ have the same weight $\delta(S[i_1], T[j_1])$, which we denote by $c_{B_{i,j}}$. We call the blocks $B_{i-1,j}$, $B_{i,j-1}$ and $B_{i-1,j-1}$ the *entering blocks* of $B_{i,j}$. The *input* of a block consists of all vertices belonging to the first row or first column of the block. The *output* of a block consists of all vertices belonging to the last row or last column of the block. The following structural lemma was also used implicitly in previous works (see formal proof in the appendix).

**Lemma 1.** *Let $B$ be a block.*

- *If $(x,y),(x,y+1) \in B$ then there is a shortest path from $(0,0)$ to $(x+1,y+1)$ that does not visit $(x,y+1)$.*
- *If $(x,y),(x+1,y) \in B$ then there is a shortest path from $(0,0)$ to $(x+1,y+1)$ that does not visit $(x+1,y)$.*

the Online Matrix-Vector Multiplication (OMV) problem) is known [15]. The lower bound implies that *worst-case* operations unlikely to be possible in $O(n^{1/2-\varepsilon})$ time. We are able to circumvent this lower bound because we only support lookups, but not range-sum queries.

- *If* $(x, y), (x+1, y+1) \in B$ *then there is a shortest path from* $(0, 0)$ *to* $(x+1, y+1)$ *that goes through* $(x, y)$.

**Frontiers in the alignment graph.** Our algorithm for DTW processes all blocks in the alignment graph. At every step, the algorithm can processes any block $B$ as long as all its entering blocks have already been processed. When block $B$ is processed, the algorithm computes $\mathsf{dist}(x, y)$ for every output vertex $(x, y)$ of $B$. After processing block $B$, we say that the output vertices of $B$ are *resolved*. At every step of the algorithm, the *frontier* is the set of resolved vertices with an outgoing edge to a block that was not yet processed. Observe that, at any given time in the execution of the algorithm, for every value $d \in [-N \dots M]$, the frontier includes exactly one vertex $(x, y)$ such that $y - x = d$. At every step $t$ of the algorithm, we will maintain an array $\mathsf{F}_t[-N \dots M]$ where $F_t[d] = \mathsf{dist}(x, y)$ such that vertex $(x, y)$ belongs to the current frontier and $y - x = d$. In Appendix A, we prove the following lemma.

**Lemma 2.** *$F_{t+1}$ can be obtained by using $O(1)$ range operations (Definition 3) on $F_t$.*

In the rest of this section, we prove that Lemma 2 implies our main result:

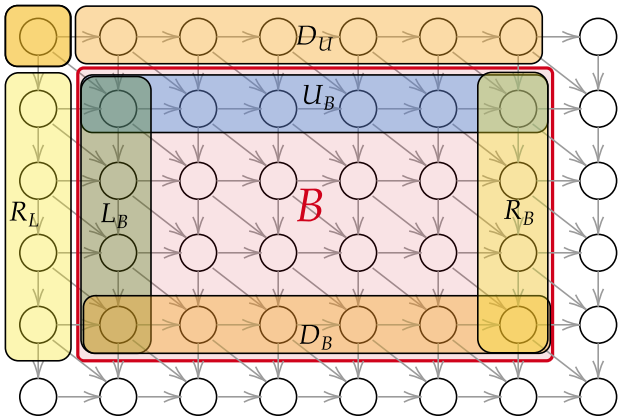


Figure 1: A block $B$, its inputs $L_B \cup U_B$ and its outputs $D_B \cup R_B$. The entering edges to $L_B$ are from $R_L$, the corner of $C$, and the leftmost node of $D_U$. The entering edges to $U_B$ are from $D_U$, the corner of $C$, and the topmost node of $R_L$.

**Theorem 2.** *The Dynamic Time Warping distance of two run-length encoded strings $S$ and $T$ with $n$ and $m$ runs respectively can be computed in $\tilde{O}(nm)$ time.*

*Proof.* We initialize the data structure of Theorem 1 as an array of length $N + M + 1$. We treat the indices of $A$ as if they are in $[-N \dots M]$[3]. Initially, the frontier consists of the vertices $(x, 0)$ with $x \in [0 \dots N]$ and $(0, y)$ with $y \in [M]$. We start by turning $A$ into $\mathsf{F}_0$. According to Eq. (1), we need to set $A[0] = 0$ and $A[i] = \infty$ for $i \neq 0$. This can be done by applying $\mathsf{AddConst}(1, M, \infty)$ and $\mathsf{AddConst}(-N, -1, \infty)$.

The algorithm runs in $nm$ iterations. At the beginning of iteration $t$, we have $A = \mathsf{F}_t$. The algorithm picks any block $B$ whose entering blocks have already been processed, and applies $O(1)$

[3]When a gradient update $\mathsf{AddGradient}(i, j, c)$ affects a value $A[k]$, we would like $A[k]$ to be increased by $k \cdot c$ with $k \in [-N \dots M]$ being the 'simulated' index rather then the actual index $k + N + 1$. This can be achieved by applying an additional operation $\mathsf{AddConst}(i, j, (-N-1) \cdot c)$.

range operations (due to Lemma 2) on $A$ in order to obtain $A = \mathsf{F}_{t+1}$. After the last iteration, it is guaranteed that the block $B_{n,m}$ has been processed. Therefore, $\mathsf{F}_{nm}[M-N] = \mathsf{DTW}(S,T)$. Every iteration requires $O(1)$ range operations each in $O(\mathsf{polylog}(nm))$ time, so overall the algorithm performs $O(nm)$ operations in total $\tilde{O}(nm)$ time. □

# 3 Implementing the Range Operations

In this section, we prove Theorem 1. We view the array $A$ as a piecewise linear function. We associate with $A$ a set $\mathcal{P} = \{p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots\}$ of points satisfying $A[x_i] = y_i$. The set $\mathcal{P}$ is uniquely defined by $A$ as the endpoints of the maximal linear segments of the linear interpolation of $A$. Note that the first point of $\mathcal{P}$ is always $(1, A[1])$ and the last point is $(n, A[n])$.[4] Let $\ell_i(x) = \alpha_i x + \beta_i$ be the line segment between $p_i$ and $p_{i+1}$. Our representation will maintain the $\alpha_i$'s and $\beta_i$'s. With this representation we can retrieve $A[x]$ for any $x \in [1, n]$ from $\alpha_i$ and $\beta_i$ where $x_i$ is predecessor of $x$ in the sequence $(x_1, x_2, \ldots)$. Upon initialization, $A$ is represented as one linear segment, with $\alpha_1 = 0$, and $\beta_1 = 0$.

We will use the following simple data structure.[5]

**Lemma 3** (Interval-add Data Structure)**.** *There is a data structure supporting the following operations in $O(\log n)$ time per operation on a set of $n$ points with distinct first coordinates.*

- $\mathsf{Lookup}(x)$ - *return the second coordinate of the point with first coordinate $x$, if exists.*
- $\mathsf{Insert}(x, y)$ - *insert the point $(x, y)$.*
- $\mathsf{Remove}(x)$ - *remove the point with first coordinate $x$, if exists.*
- $\mathsf{AddToRange}(i, j, c)$ - *for every point (x,y) with $x \in [i \ldots j]$ set $y \leftarrow y + c$.*
- $\mathsf{nextGT}(x, y)$ - *return the point $p' = (x', y')$ with smallest $x' > x$ among points with $y' > y$.*
- $\mathsf{prevLT}(x, y)$ - *return the point $p' = (x', y')$ with largest $x' < x$ among points with $y' < y$.*

## 3.1 A Warmup Algorithm

We first present a naive and inefficient implementation of a range operations data structure. We maintain the sequence $\mathcal{P}$ in a predecessor/successor data structure over the sequence $(x_1, x_2, \ldots)$. With a slight abuse of notation we shall also use $\mathcal{P}$ to refer to this data structure. We maintain the $\alpha_i$'s and $\beta_i$'s using two Interval-add data structures $D_\alpha$ and $D_\beta$, respectively. The parameters $\alpha_i, \beta_i$ of the linear segment $\ell_i$ starting at $x_i$ are represented by points $(x_i, \alpha_i)$ in $D_\alpha$ and $(x_i, \beta_i)$ in $D_\beta$. In what follows, whenever we say we add a point $p = (x, y)$ to $\mathcal{P}$ we mean that $(x, y)$ is inserted into the predecessor/successor data structure $\mathcal{P}$, and that points with first coordinate $x$ are inserted into $D_\alpha$ and $D_\beta$, with their second coordinates appropriately set to reflect the parameters $\alpha, \beta$ of the segment ending at $p$ and the segment starting at $p$. This process requires $O(1)$ operations on $\mathcal{P}, D_\alpha$ and $D_\beta$.

The effect of $\mathsf{AddConst}(i, j, c)$ (see Fig. 2) is to break the segment containing $i$ into at most 3 linear segments (a prefix ending at $i-1$, a segment $[i-1, i]$, and a suffix starting at $i$), and

[4] Here we use $n$ to denote the size of the array $A$.

[5] The data structure can be implemented using a balanced search tree with a delta-representation (where the value of a node is represented by the sum of values of its ancestors), and having every node also store the minimal and maximal values in its subtree. See e.g. [20].

similarly for the segment containing $j$. Thus, to apply $\mathsf{AddConst}(i, j, c)$, we first replace the segments containing $i$ and $j$ with these $O(1)$ new segments by inserting or updating the endpoints of the segments in $\mathcal{P}, D_\alpha$, and $D_\beta$. We then invoke $\mathsf{AddToRange}(i, j, c)$ on $D_\beta$ to shift all segments between $i$ and $j$ by $c$. Next, we set the parameters for the segment $[i-1, i]$ and for the segment $[j, j+1]$ by $O(1)$ additional calls to $\mathsf{AddToRange}$ on $D_\alpha$ and $D_\beta$. Finally, we check if any of the new segments we inserted has the same slope as its adjacent segments and, if so, we merge them into a single segment by removing their common point from $\mathcal{P}, D_\alpha$ and $D_\beta$. This guarantees that the set $\mathcal{P}$ we maintain is indeed the set $\mathcal{P}$ defined by $A$. Supporting $\mathsf{AddGradient}(i, j, g)$ is similar. The only difference is that we invoke $\mathsf{AddToRange}(i, j, g)$ on $D_\alpha$ instead of on $D_\beta$ because the slope of the segments is shifted rather than their values.

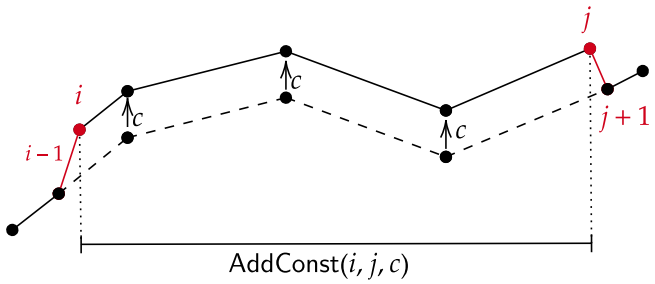


Figure 2: An illustration of applying the $\mathsf{AddConst}(i, j, c)$ operation. The dashed line represents the segments before the operation. After the operation, new points are created with $x$ coordinates $i-1, i, j$ and $j+1$ and the segments in $[i \dots j]$ are shifted by $c$.

The challenge is thus in supporting $\mathsf{LeftLinearWave}(i, j, \alpha)$. We first describe its effect and then describe how it is implemented. We assume without loss of generality that $i$ and $j$ are both endpoints of segments (otherwise we break the segments containing them into $O(1)$ segments as above). Let $p_a = (i, A[i])$ and $p_{b+1} = (j, A[j])$ be the points corresponding to $i$ and $j$. Thus, the segments contained within $[i \dots j]$ are $\ell_a, \ell_{a+1} \dots \ell_b$.

If $\alpha_a \leq \alpha$ then the segment $\ell_a$ is not affected by the linear wave. This is because for every $k \in [x_a \dots x_{a+1}]$, the linear wave assigns $A[k] \leftarrow \min_{x_a \leq t \leq k}(A[t] + (k-t)\alpha) =$

$$= \min_{x_a \leq t \leq k}(A[k] - (k-t)\alpha_a + (k-t)\alpha) = \min_{x_a \leq t \leq k}(A[k] + (k-t)(\alpha - \alpha_a)) = A[k].$$

Let $z \in [a \dots b]$ be the minimum index such that $\alpha_z > \alpha$. By the same reasoning, none of the segments $\ell_a, \ell_{a+1}, \dots, \ell_{z-1}$ is affected by the linear wave. Let $r_z(x)$ be the (positive) ray with slope[6] $\alpha$ starting at $p_z$. Since $\alpha_z > \alpha$, the ray $r_z$ is below the linear segment $\ell_z$. Hence, the segment $\ell_z$ starting at $p_z$ is affected by the linear wave; its slope changes from $\alpha_z$ to $\alpha$, and it extends beyond $x_{z+1}$ as long as $A[x] \geq r_z(x)$. We describe this effect of $\mathsf{LeftLinearWave}$ by a *ray shooting* process from $p_z$ (See Fig. 3). This process identifies the new endpoint $p'$ of $\ell_z$, and removes all the existing segments between $p_z$ and $p'$, as follows.

Let $z' \in [z+1..b+1]$ be the minimum index with $y_{z'} < r_z(x_{z'})$, i.e. the first point in $\mathcal{P}$ that lies strictly below the ray $r_z$. Let $p^* = (x^*, y^*)$ be the intersection point of the ray $r_z$ with $\ell_{z'-1}$ (if $z'$ does not exist, then $p^* = p_b$). The new endpoint of $\ell_z$ is the point $p' = (x', y') = (\lfloor x^* \rfloor, r_z(\lfloor x^* \rfloor)$,

[6]Note that in Fig. 3 and in all subsequent figures we indicate the slope $\alpha$ of the ray $r_z$ by drawing an angle $\alpha$ between the ray and the positive direction of the $x$-axis. However, formally $\alpha$ is the slope of the ray, not the indicated angle.

and it replaces all the points $p_w$ for $w \in (z \dots z')$. If $x^*$ is not an integer (or if $z'$ does not exist) then a new segment is formed between $p'$ and $p'' = (x' + 1, A[x' + 1])$.

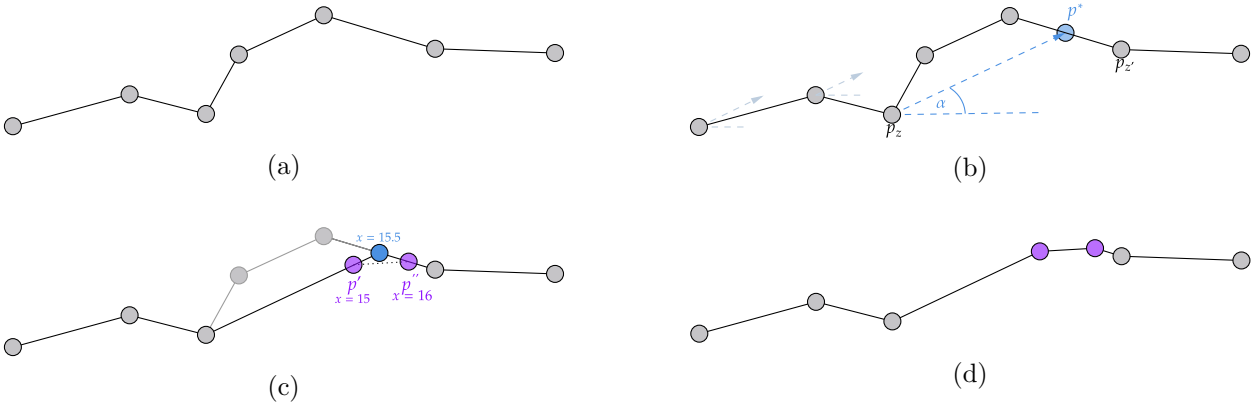


Figure 3: The effect of LeftLinearWave$(i, j, \alpha)$. The segments before $p_z$ are not affected. The segments between $p_z$ and $p_{z'}$ are affected. Namely, a ray $r_z$ with slope $\alpha$ (in dashed blue) is shot from $p_z$ and intersects at point $p^* = (x^*, y^*)$. The new endpoint of $\ell_z$ becomes $p'$ and all the segments between $p_z$ and $p'$ are removed. Since $x^* = 15.5$ is not an integer, a new segment is formed between $p'$ (with $x$ coordinate 15) and $p''$ (with $x$ coordinate 16).

The effect of LeftLinearWave$(i, j, \alpha)$ on the remaining part of $A$, namely on $A[x_{z'} \dots j]$ is analyzed in the same way as above, this time starting from $p_{z'}$ instead of from $p_a$. In particular, the prefix of segments with slopes less than $\alpha$ is not affected, and a ray with slope $\alpha$ is shot from the next $p_w$ with $\alpha_w > \alpha$, and so on. In the appendix (Lemma 13) we formally prove that the above characterization indeed represents the new values of $A[i \dots j]$.

We now describe a naive, non-efficient implementation of LeftLinearWave$(i, j, \alpha)$ according to the description above. Recall that $i$ and $j$ are assumed to be endpoints $p_a$ and $p_{b+1}$ of segments. We begin by finding the first $p_z$ with $x_z \in [i \dots j]$ and $\alpha_z > \alpha$ by querying $D_\alpha$.nextGT$(i, j, \alpha)$. A ray shooting process is then performed from $p_z$ (if $p_z$ exists) as follows: Recall that $r_z(x)$ denotes the positive ray with slope $\alpha$ shot from $p_z$. We scan the successor points of $p_z$ one by one in order, and for every point $p_w$ we check whether the ray $r_z(x_w) \le y_w$. If so, $p_w$ is removed by removing $x_w$ from $\mathcal{P}$, $D_\beta$ and $D_\alpha$. Otherwise, we compute $p^* = (x^*, y^*)$, the intersection point of $r_z$ and $\ell_{w-1}$, and from it the points $p' = (x', y') = (\lfloor x^* \rfloor, r_z(\lfloor x^* \rfloor)$ and, if $x^*$ is not an integer, also $p'' = (\lceil x^* \rceil, \ell_{w-1}(\lceil x^* \rceil))$. Then, we insert the new points $p'$ and $p''$ just before $p_w$, as discussed above for AddConst. The scanning then continues with another nextGT query from $p_w$, and so on. If, at the end of the process, the last point $p_b$ is removed since it is above some $r_z$, we insert a new point $(x_b, r_z(x_b))$.

We now analyze the time complexity of this naive implementation. Each AddConst and AddGradient operation requires $O(1)$ operations on the Interval-add data structures, and therefore takes $O(\text{polylog}|\mathcal{P}|)$ time per operation, with $|\mathcal{P}|$ being the cardinality of $\mathcal{P}$ when the operation is applied.

Regarding LeftLinearWave operations, one might hope that the cost of each ray shooting process can be charged to the removal of points from $\mathcal{P}$ during the process. However, each ray shooting process might also add up to two new points, which might result in the size of $\mathcal{P}$ increasing. Indeed, a LeftLinearWave operation may give rise to many such ray shooting processes, and hence may significantly increase the size of $\mathcal{P}$ and take too much time. This is the main technical challenge we need to address.

The idea is to distinguish between *long* ray shootings for which we can globally charge the new

insertions, and *short* ray shootings for which we cannot. We handle the long rays as in the naive solution and devise a separate lazy mechanism that delays the application of all the short rays stemming from a single LeftLinearWave operation using a constant number of updates to a separate data structure that keeps track of the delayed rays.

**Symmetry of RightLinearWave.** The discussion so far was focused on the LeftLinearWave operation. We note that the analysis of RightLinearWave is symmetric. In particular, the execution of RightLinearWave$(i, j, \alpha)$ can be described as a sequence of ray shootings with *negative* rays. The first point from which a ray is shot is $p_z$ with largest $z \in [a \dots b]$ such that $\alpha_{z-1} < -\alpha$ ($p_z$ is found using $D_\alpha$.prevLT). Note that the condition for starting a ray shooting process for RightLinearWave is on $\alpha_{z-1}$ rather than $\alpha_z$ since the slope of the segment to the left of $p_z$ is $\alpha_{z-1}$. To simplify the presentation, we will keep describing only LeftLinearWave, and will comment at the very end about the minor adjustments required to also handled the symmetric RightLinearWave.

### 3.2 Active and Passive Points, Long and Short Rays

On our way to formally define long rays and short rays we first observe that ray shootings only occur at points where slopes increase. We call such points *active* points.

**Definition 4** (Active and Passive points)**.** *A point $p_z$ in $\mathcal{P}$ is called* active *if $z \in \{1, |\mathcal{P}|\}$ or $\alpha_z > \alpha_{z-1}$. A point that is not active, is called* passive*. We denote the sets of active points by $\mathcal{P}_{\mathsf{active}}$.*

**Lemma 4.** *Ray shootings stemming from* LeftLinearWave$(i, j, \alpha)$ *occur either at point $p_a = (i, A[i])$ or at active points.*

*Proof.* Assume to the contrary that a ray shooting process starts at a passive point $p_z \neq p_a$. If $p_z$ is the first point where a ray shooting starts, then $z$ is the minimal index in $[a \dots b]$ with $\alpha_z > \alpha$. But since $p_z$ is passive, we have $\alpha < \alpha_z \leq \alpha_{z-1}$, contradicting the minimality of $z$ (note that $p_z \neq p_a$ so $z - 1 \in [a \dots b]$).

Otherwise, let $p_q$ be the last point before $p_z$ from which a ray shooting process occurred. Let $p_{q'}$ be the first point below the ray shot from $p_q$. Since $p_z$ is the next point from which a ray is shot, $z$ is the first point in $[q \dots b]$ with $\alpha_z \geq \alpha$. Since $p_z$ is passive, we have $\alpha < \alpha_z \leq \alpha_{z-1}$. If $z \neq q'$, we have $z - 1 \in [q' \dots b]$, a contradiction to the minimality of $z$. Otherwise, $p_z = p_{q'}$ is the first point below the ray with slope $\alpha$ shot from $p_q$. It follows that $p_{z-1}$ is above the ray, and $\alpha_{z-1} > \alpha$. It must be the case that $p_{q'}$ is above the ray, a contradiction. □

Similarly, we provide a proof in Appendix A for the following symmetric claim regarding RightLinearWave

**Lemma 5.** *Ray shootings stemming from* RightLinearWave$(i, j, \alpha)$ *occur either at point $p_b = (j, A[j])$ or at active points.*

Let $\mathcal{P}_{\mathsf{active}} = q_1, q_2 \dots$ be the restriction of the sequence $\mathcal{P}$ to the active points. We can think of the active points as defining a piecewise linear function whose segments are a coarsening of the segments of $A$. We refer to these segments as *mega-segments*. Let $\gamma_z$ denote the slope of the mega-segment whose endpoints are $q_z$ and $q_{z+1}$. The following lemma asserts that the segments of $A$ are never below their corresponding mega-segments, and that the slope of a segment starting at an active point is never smaller than the slope of the mega-segment starting at the same point.

**Lemma 6.** *Let $q_z = p_w$ and $q_{z+1} = p_{w'}$ be two consecutive active points. For every $k \in [w \dots w']$, the passive point $p_k$ is not below the mega-segment connecting $q_z$ and $q_{z+1}$. Furthermore, $\alpha_w \geq \gamma_z$.*

*Proof.* (See Fig. 4) Assume by contradiction that there is a point below the mega-segment, and let $k' \in (w \dots w')$ be the smallest index of such a point. Since $p_{k'-1}$ is not below the mega-segment and $p_{k'}$ is below the mega-segment, we must have $\alpha_{k'-1} < \gamma_z$. Moreover, since the points $p_k$ with $k \in [k' \dots w')$ are passive, the slopes are non-increasing and therefore every $\alpha_k \leq \gamma_z$. This means that all these points and in particular $p_{w'}$ are below the mega-segment. In contradiction to $p_{w'}$ lieing on the mega-segment. □

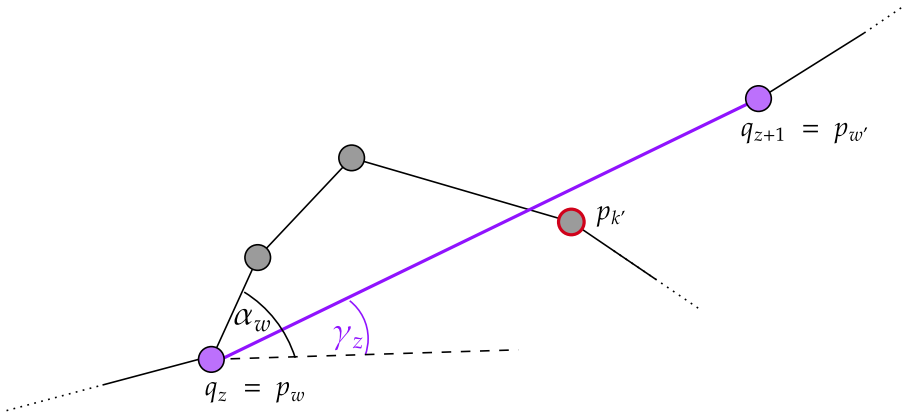


Figure 4: The impossible configuration in Lemma 6. The points $q_z$ and $q_{z+1}$ are represented by the purple points and the mega-segment connecting them is represented by a thick purple line. The first point $p_{k'}$ below the segment is marked with red stroke. Since the points strictly within the mega-segment are passive, the points following $p_{k'}$ within the mega-segment (and in particular $q_{z+1}$) must remain below the mega-segment.

We next show that if a ray shooting process starts at an active point $q_z$ with $\gamma_z < \alpha$ then the process ends before $q_{z+1}$, and the only affected points are the passive points between $q_z$ and $q_{z+1}$. On the other hand, if $\gamma_z \geq \alpha$ then as a result of the process $q_{z+1}$ ceases to be an active point, so $|\mathcal{P}_{\mathsf{active}}|$ decreases.

**Lemma 7.** *Consider a ray shooting process starting from point $p_w = q_z \in \mathcal{P}_{\mathsf{active}}$ during the application of* LeftLinearWave$(i, j, \alpha)$. *Let $q_{z+1} = p_{w'}$.*

1. *No new active points $p = (x, y)$ with $x \neq j$ are created in this process.*
2. *If $\gamma_z < \alpha$ then the points that are deleted by this process are the (passive) points $p_k$ with $k \in [w + 1 \dots r]$ for some $w < r < w'$. No other points between $p_w$ and $p_{w'}$ are deleted by* LeftLinearWave$(i, j, \alpha)$.
3. *If $\gamma_z \geq \alpha$ then $q_{z+1}$ is either deleted or becomes passive.*

*Proof.* Let $\ell$ be the ray starting from $p_w = q_z$. Assume the process terminates by finding the first point $p^* = (x^*, y^*)$ below $\ell$ (the only process that does not end this way is the one that ends by reaching $(j, A[j])$). The ray shooting process adds at most two new points $p'$ and $p''$ with decreasing slopes, so no new active points are created by the process. The slope of $p'$ is decreasing because the segment entering $p'$ is (a sub-segment of) $\ell$ and the segment leaving $p'$ is to a point below $\ell$.

The slope of $p''$ is decreasing because the line segment entering $p''$ is a line from $p'$ (a point on $\ell$) and the line segment leaving $p''$ is to the suffix of a line segment below $\ell$.

Consider the case $\gamma_z < \alpha$. Then $q_{z+1}$ is below the ray with slope $\alpha$ starting at $q_z$. Hence the ray shooting process terminates at a point after $p_{w+1}$ and before $q_{z+1}$. Since no active points are created, the next ray will be shot from $q_{z+1}$ or later, so no other points between $q_z$ and $q_{z+1}$ are deleted by $\mathsf{LeftLinearWave}(i, j, \alpha)$.

Now consider the case $\gamma_z > \alpha$. Then the mega-segment between $q_z$ and $q_{z+1}$ is above the ray with slope $\alpha$ shot from $q_z$. By Lemma 6, all the (passive) points between $q_z$ and $q_{z+1}$ are also above this ray. Hence $q_{z+1}$ is deleted by the ray shooting process.

Finally, consider the case $\gamma_z = \alpha$. Then the mega-segment between $q_z$ and $q_{z+1}$ coincides with the ray with slope $\alpha$ shot from $q_z$. By Lemma 6, all the (passive) points between $q_z$ and $q_{z+1}$ will be deleted by the ray shooting process. Let $w'$ be such that $q_{z+1} = p_{w'}$. If $\alpha_{w'} \geq \alpha$ then $q_{z+1}$ will be deleted by the process. Otherwise, $\alpha_{w'} < \alpha$, so the ray shooting process terminates at $q_{z+1}$. Since all the passive points between $q_z$ and $q_{z+1}$ were deleted, $q_z$ and $q_{z+1}$ become consecutive in $\mathcal{P}$, and the slope of the corresponding segment is $\gamma_z = \alpha$. But the slope of the segment starting at $q_{z+1}$ is $\alpha_{w'} < \alpha$, so $q_{z+1}$ becomes passive. □

We call rays with $\gamma_z > \alpha$ *long* rays, and those with $\gamma_z \leq \alpha$ *short* rays. Since long rays decrease $\mathcal{P}_{\mathsf{active}}$ we can handle them explicitly as in the warmup, charging the deletion of passive points during the process to their creation, and charging the insertion of the at most two passive points at the end of the process to the decrease in $|\mathcal{P}_{\mathsf{active}}|$. The short rays, which do not decrease $|\mathcal{P}_{\mathsf{active}}|$, will be handled lazily. Namely, instead of explicitly shooting a short ray in the mega-segment starting at an active point $q_z$, we only store the slope of the ray and postpone its execution until it is required (e.g., by a $\mathsf{Lookup}$ operation). Note that subsequent short rays shot in this mega-segment may further change the stored slope, and subsequent long rays may also affect it. We explain this in detail next.

### 3.3 The Data Structure

Since our data structure is lazy, the sequence of points it maintains will be different than the sequence $\mathcal{P}$ that would have been maintained had we used the warmup algorithm from Section 3.1. We will therefore use $\tilde{\mathcal{P}}$ to denote the set of points actually maintained by the data structure. The points $\tilde{\mathcal{P}}$ define linear segments $\tilde{\ell}_i(x)$ in the usual way. For $x \in [1, n]$ we denote by $\tilde{A}[x]$ the value $\tilde{\ell}_i(x)$, where $\tilde{\ell}_i$ is the segment containing $x$. We stress that our algorithm does not maintain $\mathcal{P}$. However, for the sake of description and analysis only we shall keep referring to the original $\mathcal{P}$, and array $A$. The definition of active and passive points, of the slopes $\gamma$ of mega-segments, and of short and long rays are now with respect to the slopes of the $\tilde{\ell}_i$'s.[7] However, we shall maintain that the set of active points with respect to $\mathcal{P}$ and $\mathcal{P}_{\mathsf{active}}$ is the same:

**Invariant 1.** $\tilde{\mathcal{P}}_{\mathsf{active}} = \mathcal{P}_{\mathsf{active}}$.

Following Section 3.1, we maintain $\tilde{\mathcal{P}}$ in a predecessor/successor data structure, as well as the Interval-add data structures $D_\alpha$ and $D_\beta$ representing the parameters of the linear segments $\tilde{\ell}_i(x)$ defined by the points of $\tilde{\mathcal{P}}$. By implementing $\mathsf{AddConst}$, $\mathsf{AddGradient}$ and long ray shootings similarly to Section 3.1 (the exact details will be spelled out below), we shall maintain the invariant that this part of the data structure correctly represents the values of active points.[8]

[7] It would have been more accurate to use $\tilde{\alpha}, \tilde{\beta}$, and $\tilde{\gamma}$, but this would be too cumbersome, so we stick to using $\alpha, \beta, \gamma$.

[8] See Invariant 3 and the note following it.

We maintain the set of active points $\tilde{\mathcal{P}}_{\mathsf{active}} = (q_1, q_2, \ldots)$ using a predecessor/successor structure on their $x$-coordinates. For each $q_z \in \tilde{\mathcal{P}}_{\mathsf{active}}$, we maintain the slope $\gamma_z$ of the mega-segment starting at $q_z$ in an Interval-add data structure $D_\gamma$. In addition, we maintain a pending short ray $r_z$ with slope $\rho_z$ passing through $q_z$ (see Fig. 5) by maintaining $\rho_z$ in a data structure $D_\rho$. This data structure, which we call the Add-min data structure is summarized below and described in detail in Section 4.

**Lemma 8** (Add-min Data Structure)**.** *There exists a data structure supporting the following operations in $O(\mathsf{polylog}n)$ time on a set of points $S$.*

1. $\mathsf{Insert}(x, y)$ - *insert the point $(x, y)$ to $S$.*
2. $\mathsf{Remove}(x)$ - *remove the a point $p = (x, y)$ from $S$, if such a point exists.*
3. $\mathsf{Lookup}(x)$ - *Return $y$ such that $p = (x, y)$ is in $S$, or report that there is no such point.*
4. $\mathsf{AddToRange}(i, j, c)$ - *for every $p = (x, y) \in S$ with $x \in [i \ldots j]$ set $y \leftarrow y + c$.*
5. $\mathsf{Min}(i, j, c)$ - *for every $p = (x, y) \in S$ with $x \in [i \ldots j]$ set $y \leftarrow \min(y, c)$.*

Note that storing $\rho_z$ suffices to compute $r_z(x)$ since the active point $q_z$ that determines the free coefficient of $r_z$ is correctly represented by $D_\alpha$ and $D_\beta$. We shall show that storing a single pending ray suffices to represent all the pending changes in a mega-segment. This property will rely on maintaining the following invariant.

**Invariant 2.** *For every active point $q_z$ we have $\rho_z > \gamma_z$. (Recall that $\gamma_z$ is the slope of the mega-segment connecting $q_z$ and $q_{z+1}$.)*

The idea is that with this representation, for any $x$, the value of $A[x]$ is given by the minimum of the value $\tilde{A}[x] = \ell_w(x)$ of the segment of $\tilde{\mathcal{P}}$ containing $x$, and the value $r_z(x)$ of the pending short ray for the mega-segment containing $x$. This is captured by the following main invariant maintained by the data structure.

**Invariant 3.** *Let $x \in [1, n]$, and let $p_w$ and $q_z$ be the predecessor of $x$ in $\tilde{\mathcal{P}}$ and in $\tilde{\mathcal{P}}_{\mathsf{active}}$, respectively. It holds that $A[x] = \min(\tilde{\ell}_w(x), r_z(x))$. Furthermore, if $p = (x, A[x])$ is an active point in $\mathcal{P}$, then $A[x] = \tilde{A}[x]$.*

Note that the first part of Invariant 3, together with Invariant 1 implies the second part of Invariant 3. This is because the predecessor of $x$ for an active point $p = (x, A[x])$ in $\mathcal{P}_{\mathsf{active}}$ is itself. Since $\mathcal{P}_{\mathsf{active}} = \tilde{\mathcal{P}}_{\mathsf{active}}$ we have that $p \in \tilde{\mathcal{P}}_{\mathsf{active}}$ is the predecessor of $x$ in $\tilde{\mathcal{P}}_{\mathsf{active}}$ as well. By definition $r_z$ goes through $p = q_z$, so $r_z(x) = \tilde{A}[x]$, and $\tilde{A}[x] = \tilde{\ell}_w(x)$ by definition. Hence, when proving that the invariants are maintained, we will not need to explicitly establish the second statement in Invariant 3.

Initially, $\tilde{\mathcal{P}} = \mathcal{P} = \{(1, 0), (|A|, 0)\}$, and $\rho_1 = \rho_2 = \infty$. Indeed, $A[x] = \min(\tilde{A}[x], r_1(x)) = \min(0, \infty) = 0$ and Invariant 3 is satisfied. It remains to specify the implementation of the various operations supported by the data structure, to prove that the invariants are maintained, and to analyze the running times.

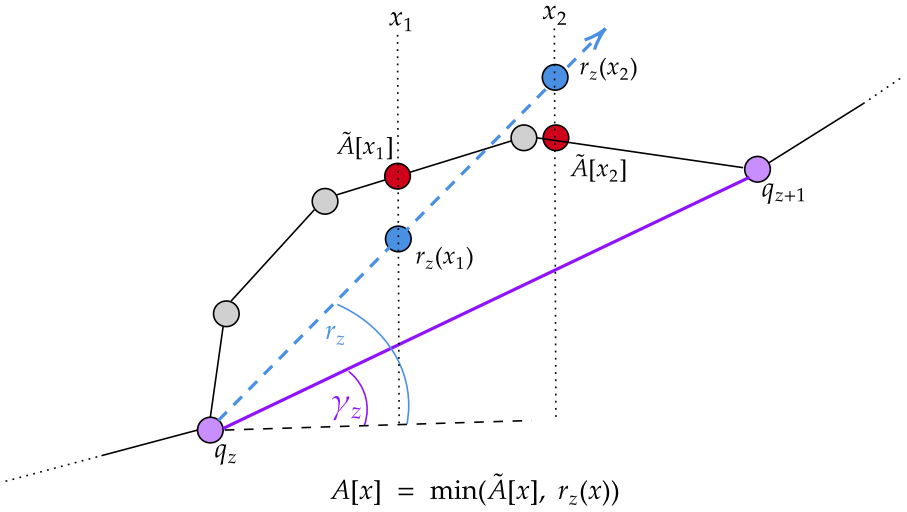


Figure 5: An illustration of the data stored for a mega-segment between two consecutive active points $q_z$ and $q_{z+1}$ (purple points). The slope $\gamma_z$ is the slope of the mega-segment. The slope $\rho_z > \gamma_z$ stored in $q_z$ represents a pending ray $r_z$ (dashed blue) that should be shot from $q_z$. The value of $A[x]$ is the minimum between $r_z(x)$ (a blue point) and $\tilde{A}[x]$ (a red point), the value of the piece-wise linear function defined by $\tilde{\mathcal{P}}$ (in grey).

**The flush Operation.** We first describe a service operation $\mathsf{flush}(q_z)$ which explicitly shoots the pending short ray in the mega-segment starting at the active point $q_z$. It will be useful to invoke flush before serving Lookup operations, but also when serving the other operations in order to guarantee that the lazy implementation properly follows the explicit implementation in the warmup. This is particularly important in operations which may create $O(1)$ new active points and thus change the partition into mega-segments, but is also useful to streamline the proof of correctness. Recall that the reason we avoided shooting local rays in the first place was that there could be many of them, and we could not afford to pay for the possible creation of $O(1)$ new passive points at the end of each of them. We can afford, however, to perform $O(1)$ flush operations before each Lookup, AddConst or AddGradient operation, because the cost of adding the $O(1)$ new points can be charged to the operation itself.

A flush of $q_z = p_w \in \mathcal{P}_{\mathsf{active}}$ is performed as follows. Starting from $p_{w+1}$, we scan the points in $\tilde{\mathcal{P}}$. When scanning $p = (x, y)$, we compare $y$ and $r_z$. If $r_z(x) \leq y$, we remove $p$ from $\tilde{\mathcal{P}}$. Otherwise, the scan halts. Let $p_{\mathsf{end}}$ be the point on which the scan halts. If no point was deleted throughout the scan, we set $\rho_z = \infty$ and terminate. Otherwise, let $p_{\mathsf{del}}$ be the last point deleted by the scan. We compute the intersection $p^* = (x^*, y^*)$ of $r_z$ and the line $\tilde{\ell}$ between $p_{\mathsf{del}}$ and $p_{\mathsf{end}}$. Finally, we insert $p' = (\lfloor x^* \rfloor, r_z(\lfloor x^* \rfloor))$ and $p'' = (\lceil x^* \rceil, \tilde{\ell}(\lceil x^* \rceil))$ to $\tilde{\mathcal{P}}$ (as in the warmup algorithm of Section 3.1), update $D_\alpha$ and $D_\beta$ with the new parameters of the segments ending and starting at $p'$ or at $p''$, and set $\rho_z = \infty$.

**Lemma 9.** *Applying* flush *to an active point $q_z \in \mathcal{P}_{\mathsf{active}}$ preserves Invariants 1 to 3. Furthermore, it guarantees that the restriction of $\mathcal{P}$ and $\tilde{\mathcal{P}}$ to the (passive) points between $q_z$ and $q_{z+1}$ is identical, and that for every $x \in [x_z \dots x_{z+1}]$, $A[x] = \tilde{A}[x]$.*

*Proof.* Invariant 2 is maintained because the flush operation sets $\rho_z$ to $\infty$. Since $\rho_z > \gamma_z$, it is guaranteed by Lemma 7 that the scan of flush ends at $q_{z+1}$ or before $q_{z+1}$. It follows that Invariant 1 is maintained because $\mathcal{P}_{\mathsf{active}}$ does not change and flush only deletes passive points of

$\tilde{\mathcal{P}}$. We proceed to prove that Invariant 3 is maintained. Note that $\rho_z$ is set to $\infty$ by the end of flush, and that $q_z$ remains the predecessor active point of every $x \in [x_z \dots x_{z+1}]$, so we need to show $A[x] = \tilde{A}[x]$. Let $x \in [x_z \dots x_{z+1}]$. If $x \le x^*$, then before flush was applied, we had $\tilde{A}[x] \ge r_z(x)$, and therefore by Invariant 3 $A[x] = \min(\tilde{A}[x], r_z(x)) = r_z(x)$. Since flush sets the value of $\tilde{A}[x]$ to be $r_z(x)$ for $x < x^*$, Invariant 3 still holds. If $x > x^*$, the value of $\tilde{A}[x]$ is not changed by flush. Since the line $\tilde{\ell}$ between $p_{\mathsf{del}}$ and $p_{\mathsf{end}}$ starts not below the $r_z$ and ends below $r_z$, its slope is smaller than $\rho_z$. Since the points between $p_{\mathsf{end}}$ and $q_z$ (excluding $q_z$) are passive, the slopes of the corresponding segments are also lower than $\rho_z$ and therefore $(x, \tilde{A}[x])$ is below $r_z$ for every $x \in (x' \dots x_{z+1}]$. Due to Invariant 3 before the application of flush, we have $A[x] = \min(\tilde{A}[x], r_z(x)) = \tilde{A}[x]$. Therefore, assigning $\rho_z \leftarrow \infty$ and not changing $\tilde{A}[x]$ satisfies Invariant 3. □

### 3.4 Implementing the Data Structure

**Lookup$(k)$.** To perform Lookup$(k)$ we retrieve the predecessor $q_z$ of $k$ in $\tilde{\mathcal{P}}_{\mathsf{active}}$, and invoke flush$(q_z)$. We then retrieve the predecessor $p_w$ of $k$ in $\tilde{\mathcal{P}}$ and return $\tilde{\ell}_w(k)$ which is correct by Lemma 9. All three invariants are clearly maintained by this operation.

**AddConst$(i, j, c)$.** Similar to the implementation in the warmup algorithm (Section 3.1), we first perform Lookup queries to retrieve $A[x]$ for $x \in \{i-1, i, j, j+1\} = \mathcal{B}$. Let $\mathcal{P}_{\mathcal{B}}$ be the resulting set of $O(1)$ points. We assume that no point of $\mathcal{P}_{\mathcal{B}}$ was previously in $\tilde{\mathcal{P}}$ (the other cases are handled similarly). We insert the points of $\mathcal{P}_{\mathcal{B}}$ into $\tilde{\mathcal{P}}$ and update $D_\beta$ and $D_\alpha$ accordingly using $O(1)$ operations. Note that at this point all $O(1)$ mega-segments containing points in $\mathcal{P}_{\mathcal{B}}$ are flushed (because of the calls to Lookup). Next, we apply $D_\beta$.AddToRange$(i, j, c)$. Note that this changes the linear segments between $i-1$ and $i$ and between $j$ and $j+1$. We update $D_\alpha$ and $D_\beta$ to reflect these changes using $O(1)$ additional operations.

Next, for every $p_k = (x_k, y_k) \in \mathcal{P}_{\mathcal{B}}$, if $p_k$ just became active, then we insert $x_k$ to $\tilde{\mathcal{P}}_{\mathsf{active}}$ and set the $\rho$ value of $x_k$ in $D_\rho$ to be $\infty$. Otherwise, if $p_k$ just became passive, then we remove $x_k$ from $\tilde{\mathcal{P}}_{\mathsf{active}}$ and $D_\gamma$ if necessary. Finally, if $\alpha_k = \alpha_{k-1}$, we merge the two segments by removing $p_k$ from $\tilde{\mathcal{P}}, D_\alpha$ and $D_\beta$.

**Lemma 10.** *Applying* AddConst$(i, j, c)$ *preserves Invariants 1 to 3.*

*Proof.* The only points of $\mathcal{P}$ that become active or passive due to AddConst$(i, j, c)$ are the points in $\mathcal{P}_{\mathcal{B}}$. Since the mega-segments containing points in $\mathcal{P}_{\mathcal{B}}$ are flushed, Lemma 9 and the explicit handling by AddConst of the points of $\mathcal{P}_{\mathcal{B}}$ that become active or passive guarantee that Invariant 1 is maintained.

Similarly, the only mega-segments whose $\gamma$ value is changed by AddConst$(i, j, c)$ are those containing points of $\mathcal{P}_{\mathcal{B}}$. The values $\rho$ for all these mega-segments are set to $\infty$ either by flushing or explicitly. Hence Invariant 2 holds.

To establish Invariant 3, let $k \in [1 \dots |A|]$ and let $q_z$ be the active point prior to the application of AddConst$(i, j, c)$ such that $k \in [x_z \dots x_{z+1}]$.

- If $[x_z \dots x_{z+1}] \cap [i-1 \dots j+1] = \emptyset$, then both $A[k]$ and $\tilde{A}[k]$ are not affected by the update. Moreover, the predecessor active point of $k$ remains $q_z$ after the update, and $\rho_z$ was not affected by the update. It follows that $\min(\tilde{A}[k], r_z(k))$ is not changed.

- If $[x_z \dots x_{z+1}] \subseteq [i+1 \dots j-1]$, then notice that $q_z$ remains active after the update since $\alpha_z$ and $\alpha_{z-1}$ are not affected by the update. The value of $\tilde{A}[k]$ and the $y$ coordinate of $q_z$ were increased by $c$ via the $D_\beta$.AddToRange$(i, j, c)$ operation. The value $\rho_z$ was not changed,

so $r_z(k)$ was increased by $c$ as well. It follows that $\min(\tilde{A}[k], r_z(k))$ was increased by $c$, as required.

- If $[x_z \dots x_{z+1}] \cap \mathcal{B} \neq \emptyset$, then note that we applied a flush operation on $q_z$, so we have $\rho_z = \infty$ and $\tilde{A}[k] = A[k]$ prior to the application of $D_\beta$.AddToRange$(i, j, c)$. Thus, after the flush operation, we have $A[k] = \min(\tilde{A}[k], r_z(k)) = \min(\tilde{A}[k], \infty) = \tilde{A}[k]$. After applying the update AddConst$(i, j, c)$, the predecessor active point of $k$ is either $q_z$, some point in $\mathcal{P}_\mathcal{B}$ (if a point in $\mathcal{P}_\mathcal{B}$ became active as a result of the operation), or the predecessor active point of a point in $\mathcal{P}_\mathcal{B}$ (if a point in $\mathcal{P}_\mathcal{B}$ was $q_z$, and became passive as a result of the update). In all these cases, the predecessor active point $q_a$ of $k$ in the updated representation has $\rho_a = \infty$ (since either it is a new active point, or it is an existing active point on which a flush was applied). The value of $\tilde{A}[k]$ was increased by $c$ via the operation $D_\beta$.AddToRange$(i, j, c)$. In conclusion, we have $\min(\tilde{A}[k], r_a(k)) = \min(\tilde{A}[k], \infty) = \tilde{A}[k]$. Since $\tilde{A}[k]$ was increased by $c$ if it was necessary, it is now representing the value of $A[k]$ after the AddConst$(i, j, c)$ operation. □

**AddGradient$(i, j, c)$.** As was the case in the warmup algorithm, the implementation of AddGradient is similar to that of AddConst except that rather than applying AddToRange$(i, j, c)$ to to $D_\beta$, it is applied to $D_\alpha$ to increase the slope of all line segments between $i$ and $j$ by $c$. In the same manner we increase the slope of the corresponding mega-segments by $c$ using $O(1)$ calls to AddToRange on $D_\gamma$ (the mega-segments containing $i$ and $j$ need a special treatment since their slope might increase by less than $c$). Finally, we increase the slope of the pending rays using AddToRange on $D_\rho$.

**Lemma 11.** *Applying* AddGradient$(i, j, c)$ *preserves Invariants 1 to 3.*

*Proof.* The proof for Invariant 1 is identical to that in Lemma 10. Invariant 2 holds since the only mega-segments whose $\gamma$ and $\rho$ change by different values are those containing points of $\mathcal{P}_\mathcal{B}$, and those are flushed and handled explicitly by the implementation.

As for Invariant 3, let $k \in [1 \dots |A|]$ and let $q_z$ be the active point such that $k \in [x_z \dots x_{z+1}]$ prior to the application of AddGradient$(i, j, c)$.

- If $[x_z \dots x_{z+1}] \cap [i-1 \dots j+1] = \emptyset$, then no changes occurs, just like in the proof of Lemma 10.
- If $[x_z \dots x_{z+1}] \subseteq [i+1 \dots j-1]$, notice that $q_z$ is still active since $\alpha_z$ and $\alpha_{z-1}$ were both increased by $c$. The value of $\tilde{A}[k]$ was increased by $k \cdot c$ via the $D_\alpha$.AddToRange$(i, j, c)$ operation. The $y$ coordinate of $q_z$ was increased by $x_z \dots c$ via the same operation. The value $\rho_z$ was increased by $c$ as well, so $r_z(k)$ was increased by $x_z + (k - x_z) \cdot c = k \cdot c$. It follows that $\min(\tilde{A}[k], r_z)$ was increased by $k \cdot c$, as required.
- If $[x_z \dots x_{z+1}] \cap B \neq \emptyset$, then $A[k] = \tilde{A}[k]$ by the same argument as in the proof of the corresponding case in Lemma 10. □

**LeftLinearWave$(i, j, c)$.** In the description of the algorithm we will say that it *explicitly* performs a ray shooting process from some point $p$. By this we mean the following. First, the mega-segment containing $p$ is flushed. We then scan the points of $\tilde{\mathcal{P}}$ starting in $p$ using Successor queries in $\tilde{\mathcal{P}}$. Similarly to the warmup algorithm of Section 3.1, we delete the scanned point $p_k = (x_k, y_k)$ from $\tilde{\mathcal{P}}$ if $y_k \geq \ell(x)$ with $\ell$ being the ray with slope $c$ shot from $p$. If the scan reaches an active point $q_w = (x_w, y_w)$, and finds that $q_w$ is not below $\ell$ - we perform a flush operation on $q_w$ before deleting $x_w$ from $\tilde{\mathcal{P}}, D_\alpha, D_\beta$. We also delete $x_w$ from $\tilde{\mathcal{P}}_{\text{active}}, D_\gamma, D_\rho$, and update the slopes of the predecessors of $q_w$ in $\tilde{\mathcal{P}}$ and in $\tilde{\mathcal{P}}_{\text{active}}$ accordingly. Upon reaching a point $p_k$ that is below the ray

$\ell$ (or when reaching $p_b$), we add to $\tilde{\mathcal{P}}$ the points $p'$ and $p''$ (delete $p_b$ and add $(j, \ell(j))$ if necessary) as described in Section 3.1, and the ray shooting process terminates.

We now describe the algorithm for LeftLinearWave$(i, j, c)$. (Refer to Fig. 6). First, as in the AddConst and AddGradient operations, we add the points in $\mathcal{P}_{\mathcal{B}}$ to $\tilde{\mathcal{P}}$, and flush every mega-segment containing a point from $\mathcal{P}_{\mathcal{B}}$ . If $\alpha_a > c$, we explicitly perform a ray shooting process from $p_a$. Let $p$ be the point at which the explicit ray shooting process terminated, or $p = p_a$ if $\alpha_a \leq c$.

Let $q_w = (x_w, y_w)$ be the first active point (weakly) after $p$. We use $D_\gamma$.nextGT$(x_w, c)$ to obtain the point $q_z = (x_z, y_z)$ from which the next long ray shooting should start. We then implicitly shoot short rays in every mega-segment starting at an active point $q_t = (x_t, y_t)$ with $x_t \in [x_w \dots x_z)$. This is done by applying a single operation, Min$(x_w, x_{z-1}, c)$ on the Add-min data structure $D_\rho$ maintaining the $\rho_z$ values. This operation has the effect of setting $\rho_t \leftarrow \min(\rho_t, c)$ for all such $t$'s. Next, we explicitly perform the long ray shooting from $q_z$.

We keep repeating the above paragraph with the point at which the last explicit long ray shooting process terminated taking the role of $p$. We stop if an explicit ray shooting reaches $j$ or if $D_\gamma$.nextGT$(x_w, c)$ returns 'null' or a point beyond $j$. In the latter case, let $q_{b'}$ be the starting point of the mega-segment containing $j$. We implicitly shoot all the short rays in all the mega-segments starting not earlier than $p$ and ending no later than $q_{b'}$. Finally, we call flush$(q_{b'})$, and if $\alpha_{b'} > c$ we explicitly perform a ray shooting process from $q_{b'}$.

To finalize we also need to update the effects around $i$ and $j$. We check for every $p_k \in \mathcal{P}_{\mathcal{B}}$ if $p_k$ is active, and update $D_\gamma$ accordingly (similar to this update in AddConst).

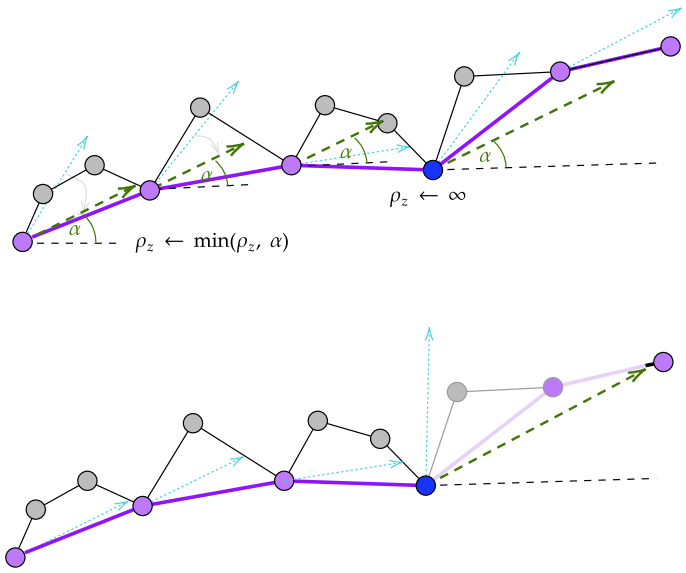


Figure 6: A demonstration of the first explicit long ray shooting process. First, the leftmost active point $q_z$ with $\gamma_z$ larger than $\alpha$ (blue) is identified (assuming that a ray shooting process from $p_a$ is not required). Then, the $\rho$ values (light blue rays) of the active points preceding $q_z$ is assigned $\rho \leftarrow \min(\rho, \alpha)$. Finally, a ray shooting process is explicitly applied from $q_z$. Since an explicit ray shooting process starts by applying flush to the mega-segment of $q_z$, the value of $\rho_z$ is set to $\infty$.

**Lemma 12.** *Applying* LeftLinearWave$(i, j, c)$ *preserves Invariants 1 to 3.*

*Proof.* We start by proving the following claim.

**Claim 1.** *The implementation of* LeftLinearWave *performs exactly the same* long *ray shooting processes as the warmup algorithm of Section 3.1 (same starting points, ending points, and points deleted).*

*Proof.* Since the mega-segment containing $p_a$ is flushed, our algorithm shoots a ray from $p_a$ if and only if the warmup algorithm also has. If we did not shoot a ray from $p_a$, our algorithm shoots a long ray from the first active point $q_z$ with $\gamma_z > c$. Since $\mathcal{P}_{\mathsf{active}} = \tilde{\mathcal{P}}_{\mathsf{active}}$ (Invariant 1), the $\gamma$ slopes stored in $D_\gamma$ are the slopes of the mega-segments of $\mathcal{P}$. Therefore, the first long ray shot by the warmup algorithm does not start before $q_z$. It is possible that the warmup algorithm executed several short ray shooting processes from active points $q_w$ with $w < z$, but by Lemma 7, the effect of these short rays is confined to the mega-segment starting at $q_w$, and does not affect any active points. Therefore, it is guaranteed that the first long ray shot by the warmup algorithm is also from $q_z$.

Consider the first long ray shooting process applied from $q_z$ (or from $p_a$, depending on $\alpha_a$). Throughout the scan, we always perform a flush operation on a mega-segment before scanning points in the mega-segment. Therefore, by Lemma 9, it is guaranteed that we see the exact same points as in the scan of the warmup algorithm. It follows that the scan terminates by creating exactly the same point as the warmup algorithm, and deletes all the points between those points and $q_z$). Our algorithm then proceeds to find where the next long ray should start using $D_\gamma$.nextGT. The claim follows inductively by repeating the reasoning above on all subsequent long rays. □

**Invariant 1.** By Claim 1, every active point that is deleted from $\mathcal{P}_{\mathsf{active}}$ as a result of a ray shooting process is deleted from $\tilde{\mathcal{P}}_{\mathsf{active}}$ as well (and only these points). By Lemma 7, the only point that may become active as a result of a ray shooting is $p = (j, A[j]) \in P_{\mathcal{B}}$. This is explicitly handled by the algorithm by flushing the mega-segment containing $j$, which guarantees that the slopes of the segments ending and starting at $p$ are identical in $\mathcal{P}$ and in $\tilde{\mathcal{P}}$. Hence, $p$ is active in $\tilde{\mathcal{P}}$ if and only if it is active in $\mathcal{P}$, and Invariant 1 is satisfied.

**Invariant 2.** Let $q_z$ be an active point in $\tilde{\mathcal{P}}_{\mathsf{active}}$ after the application of LeftLinearWave$(i, j, c)$. Note that since a ray shooting does not create any new active points (except possibly a point $(j, A[j])$), the point $q_z$ was active also before the application of LeftLinearWave$(i, j, c)$ (or $q_z$ is a point in $\mathcal{P}_B$). If flush$(q_z)$ was invoked then $\rho_z$ is set to $\infty$ and Invariant 2 is satisfied.

Otherwise, it must be the case that a long ray shooting process was not executed from $q_z$. Furthermore, no long ray shooting deleted a point in the mega-segment of $q_z$. It follows that $q_{z+1}$ was also not affected by the update, since a ray shooting starting in $q_w$ with $w \geq z + 1$ does not change the value of $q_{z+1}$, and a short ray shooting process that may have been applied from $q_z$ does not change $q_{z+1}$ as well. Therefore, $\gamma_z$ is unchanged by LeftLinearWave$(i, j, c)$. Since a long ray did not start at $q_z$, and $q_z$ was not deleted by a long ray shooting, we must have that $\gamma_z < c$. Before the application of LeftLinearWave, we had $\gamma_z < \rho_z$. Either $\rho_z$ is unchanged by the algorithm, or it was set to $\min(c, \rho_z)$ by a short ray. In both cases, Invariant 2 is maintained.

**Invariant 3.** Let $x \in [n]$. We distinguish between two cases regarding the mega-segment $[x_z \dots x_{z+1}]$ containing $x$ prior to the application of LeftLinearWave$(i, j, c)$.

- If $q_z$ was deleted by an explicit long ray shooting. Let $q_w$ be the point from which the long ray started. Hence, after the application of LeftLinearWave$(i, j, c)$, $q_w$ is the predecessor active point of $x$, and the mega-segment starting at $q_w$ (and containing $x$) is flushed. Hence the

algorithm sets $\rho_w = \infty$, applies on this segment exactly the same changes as the warmup algorithm $\tilde{A}[x] = A[x] = \min(\tilde{A}[x], r_w(x))$ as required.

- If $q_z$ was not deleted by an explicit long ray shooting. It follows from Claim 1 and from the correctness of the warmup algorithm that if the value of $A[x]$ needs to be modified, it is as a result of a short ray shooting. If the warmup algorithm does not perform a ray shooting process from $q_z$, it must be the case that $\gamma_z < c$. In this case, the assignment $\rho_z \leftarrow \min(\rho_z, c)$ does not change $\rho_z$, so $\min(\tilde{A}[x], r_z)$ is not changed, as required. We proceed to treat the case in which a local ray $r$ with slope $c$ is shot from $q_z$ by the warmup algorithm. Let $A_{\mathsf{before}}[x]$ and $A_{\mathsf{after}}[x]$ be the values of $A[x]$ before and after $\mathsf{LeftLinearWave}(i, j, c)$ is applied, respectively. The value of $A_{\mathsf{after}}[x]$ is $A_{\mathsf{after}}[x] = \min(A_{\mathsf{before}}[x], r(x))$. Let $r_z^{\mathsf{before}}$ (resp. $r_z^{\mathsf{after}}$) be the pending ray with slope $\rho_z$ (resp. $\min(\rho_z, c)$) at $q_z$ before (resp. after) applying $\mathsf{LeftLinearWave}(i, j, c)$. Note that $r_z^{\mathsf{after}}(x) = \min(r_z^{\mathsf{before}}(x), r(x))$, so

$$\min(\tilde{A}[x], r_z^{\mathsf{after}}(x)) = \min(\tilde{A}[x], r_z^{\mathsf{before}}(x), r(x)) =$$
$$\min(\min(\tilde{A}[x], r_z^{\mathsf{before}}(x)), r(x)) = \min(A_{\mathsf{before}}[x], r(x)) = A_{\mathsf{after}}[x]$$

as required. □

**Complexity.** We start by showing that the number of active points added to $\tilde{\mathcal{P}}_{\mathsf{active}}$ throughout a sequence of $s$ operations is $O(s)$. This is because $\mathsf{flush}$ operations do not add active points to $\tilde{\mathcal{P}}_{\mathsf{active}}$, and each invocation of $\mathsf{AddConst}$, $\mathsf{AddGradient}$ or $\mathsf{LeftLinearWave}$ may create $O(1)$ active points.

Consider a sequence of $s$ operations. We use a standard charging argument to prove that the amortized time per operation is $\tilde{O}(1)$. The only difficulty is in charging the time of ray shootings that are performed explicitly in any call to $\mathsf{flush}$ and during $\mathsf{LeftLinearWave}$, and the time of the implicit short ray shootings during $\mathsf{LeftLinearWave}$.

We charge to each operation the $\tilde{O}(1)$ time of handling the mega-segments containing the points of $\mathcal{P}_{\mathcal{B}}$, including the time to insert the $O(1)$ new points in $\mathcal{P}_{\mathcal{B}}$ but excluding calls to $\mathsf{flush}$ on these mega-segments. Similarly, we charge the $\tilde{O}(1)$ time update $D_\alpha$ and $D_\beta$ during $\mathsf{AddConst}$ and $\mathsf{AddGradient}$ to the operation itself.

Each call to $\mathsf{flush}$ may remove many passive points from $\tilde{\mathcal{P}}$ and, in addition, takes $\tilde{O}(1)$ time to insert $O(1)$ new passive points into $\tilde{\mathcal{P}}$. The time to delete each point $p$ of $\tilde{\mathcal{P}}$ is charged to the insertion of $p$. Calls to $\mathsf{flush}$ on a mega-segments containing points in $\mathcal{P}_{\mathcal{B}}$, or to the mega-segment containing the point $p_a$ in $\mathsf{LeftLinearWave}$ charge the additional $\tilde{O}(1)$ required time to the calling operation. All other calls to $\mathsf{flush}$ occur during explicit long ray shootings in $\mathsf{LeftLinearWave}$, and will be charged next.

Every mega-segment $[q_z, q_{z+1}]$ that is encountered during an explicit long ray shooting, except the last one, results in deleting the active point $q_{z+1}$ from $\tilde{\mathcal{P}}_{\mathsf{active}}$. For each such mega-segment we charge $\tilde{O}(1)$ time for inspecting $q_z$ and calling $\mathsf{flush}(q_z)$ to the the deletion of $q_{z+1}$ from $\tilde{\mathcal{P}}_{\mathsf{active}}$. For the last mega-segment, by Lemma 4, $q_{z+1}$ is either deleted or becomes inactive, so we charge $\tilde{O}(1)$ time to $|\tilde{\mathcal{P}}_{\mathsf{active}}|$ decreasing by 1. Note that handling this last segment may include the insertion of $O(1)$ new passive points to $\tilde{\mathcal{P}}$, which is within the $\tilde{O}(1)$ charged budget.

Finally, we charge the time of calls to $D_\rho.\mathsf{Min}$ in implementing short ray shootings implicitly. Each of these calls results from some $\mathsf{LeftLinearWave}$ operation. We charge $O(1)$ such calls to the $\mathsf{LeftLinearWave}$ operation itself, and each of the remaining calls to the long ray shooting preceding it.

Each operation and each decrease in $|\tilde{\mathcal{P}}_{\mathsf{active}}|$ was charged $\tilde{O}(1)$ time. Since the sequence consists

of $O(s)$ operations and since, as we have shown, $O(s)$ points ever become active, the total time for serving the entire sequence is $\tilde{O}(s)$.

**Handling** RightLinearWave**.** As we had mentioned above, handling RightLinearWave is symmetric to LeftLinearWave with the algorithm proceeding right-to-left, starting from $p_b$, and shooting negative rays from active points (the definition of active points remains unchanged). To keep track of pending short negative rays we maintain an additional Add-min data structure $D'_\rho$, and now $A[x]$ is obtained as $\min(\tilde{A}[x], r_z(x), r'_{z+1}(x))$, where $r'_{z+1}$ is the pending negative ray going through $q_{z+1}$. The proof of correctness, maintenance of invariants, and analysis of complexity are completely symmetric to those of LeftLinearWave. With that, the proof of Theorem 1 is complete.

# 4 The Add-min Data Structure

In this section we describe the Add-min data structure of Lemma 8. To explain the main idea of the data structure we assume that no points are added or removed and focus on supporting just $\mathsf{Min}(i, j, c)$ and $\mathsf{AddToRange}(i, j, c)$. We consider the points $p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots$ ordered by their $x$-coordinates. The Add-min data structure is recursive. It consists of an Interval-add data structure $D$, and of a recursive instance of Add-min $R$, which only stores a constant fraction $(2/3)$ of the points.

The points are partitioned into pairs of consecutive points. The *representative* of a point $p = (x, y)$ is the first point in the pair that $p$ belongs to. Let $M(x)$ denote the $x$-coordinate of the representative of $p$. Initially $D$ stores all the points $p_k = (x_k, y_k)$, and $R$ stores only the representatives, which are initialized to $\infty$. Namely, $(x_1, \infty), (x_3, \infty), (x_5, \infty), \ldots$. We maintain the invariant that $y_k = \min(D.\mathsf{Lookup}(x_k), R.\mathsf{Lookup}(M(x))$, so a $\mathsf{Lookup}(x_k)$ query on the data structure can be served with a single Lookup on $D$ and a single recursive Lookup on $R$, which would take total polylogarithmic time.

We use a service operation $\mathsf{Assign}(x_k, c)$, that assigns $y_k \leftarrow c$ if there is a point $p = (x_k, y_k) \in S$. The operation $\mathsf{Assign}(x_k, c)$ is implemented by making $D$ store the value $c$ as follows (note that doing Assign on $D$ is trivial using Remove and Insert). Let $p_{k'}$ be the other point in the pair with $p_k$. We obtain the value of $y_{k'}$ using a Lookup operation, and call $D.\mathsf{Assign}(x_k, c)$, $D.\mathsf{Assign}(x_{k'}, y_{k'})$, and recursively call $R.\mathsf{Assign}(M(x_k), \infty)$.

To implement $\mathsf{Min}(i, j, c)$, let $p_a$ (resp. $p_b$) be the first point with $x_a \geq i$ (resp. $x_b \leq j$). Assume first that $p_a$ is a representative (i.e., $M(x_a) = x_a$) and $p_b$ is not, so the effected range $[i, j]$ exactly corresponds to a range of consecutive pairs of points. We simply invoke $R.\mathsf{Min}(i, j, c)$, which clearly correctly implements the Min operation while maintaining the invariant. If $p_a$ is not a representative then we need to handle the pair containing $p_a$ differently since we do not want to affect the value of $p_{a-1}$. We obtain the values of $y_{a-1}$ and $y_a$ before the update using two Lookup operations, and assign these values by calling $D.\mathsf{Assign}(x_{a-1}, y_{a-1})$, $D.\mathsf{Assign}(x_a, \min(y_a, c))$, $R.\mathsf{Assign}(x_{a-1}, \infty)$ (this last call is a recursive assignment). This maintains the invariant and guarantees that both $p_{a-1}$ and $p_a$ are correctly represented. We handle similarly the case where $p_b$ is a representative.

We implement $\mathsf{AddConst}(i, j, c)$ in a similar spirit. If $p_a$ is a representative and $p_b$ is not, we simply invoke $D.\mathsf{AddToRange}(i, j, c)$ and $R.\mathsf{AddToRange}(i, j, c)$. Otherwise, we handle the endpoints of the intervals explicitly in the manner described for Min.

To support insertions and deletions of points we can no longer work with the rigid partition into consecutive pairs. Instead, we shall use the standard technique of partitioning the points into segments consisting of a single point or two consecutive points. Only the first point from each segment is represented in the recursive structure. We make sure to merge consecutive segments

whenever both contain just a single point. This guarantees that the number of segments is at most 2/3 the number of points, and hence the recursive structure is sufficiently small.

To keep track of the partition into segments we maintain the representatives in a predecessor data structure $M$. The representative of a point $p_k$ is then given by the predecessor of $x_k$ in $M$. The invariant now becomes $y_k = \min(D.\mathsf{Lookup}(x_k), R.\mathsf{Lookup}(M.\mathsf{Predecessor}(x_k))$. We denote by $D[x]$ (resp. $R[x]$) the value of the $y$ coordinate of the point with $x$ coordinate in $D$ (resp. in $R$) if such a point exists. We denote by $M[x]$ the value of $M.\mathsf{Predecessor}(x)$. With this notation the invariants we maintain is:

**Invariant 4.** *For every $p = (x, y)$ in the data structure, we have $y = \min(D[x], R[M[x]])$.*

We denote the segments by $s_1, s_2 \dots s_r$. The invariant on the segments is:

**Invariant 5.** *For every $i \in [1 \dots r-1]$ either $s_i$ is of length two or $s_{i+1}$ is of length two.*

Note that a direct consequence of Invariant 5 is that $r \leq \lceil \frac{2n}{3} \rceil$. Also note that, for every $i \in [1 \dots |D|]$, $M[x_i]$ is either $x_i$ or $x_{i-1}$.

For completeness we give all the details of the data structure. Upon initialization, the data structure contains no elements and $D, M$, and $R$ are empty.

**$\mathsf{Lookup}(x)$.** We perform a $\mathsf{Lookup}(x)$ query on $D$, and a (recursive) $\mathsf{Lookup}(M[x])$ query on $R$ and return the minimum of the two.

**$\mathsf{Assign}(x, y)$.** Let $x_{i'} = M[x_i]$ and let $s_a$ be the segment containing the point $p$ with $x$ value of $x$. We assign $D[x] \leftarrow y$. If there is another point $p' = (x', y')$ in $s_a$ we also assign $D[x'] \leftarrow y'$. Note that we can identify the two candidates for being $p'$ using predecessor and successor queries, and confirm which is in $s_a$ using $M$. We conclude by (recursively) assigning $R[x_{i'}] \leftarrow \infty$.

**$\mathsf{Shift}(x, x')$.** We introduce a service operation that would be useful for maintaining the invariants throughout insertions and deletions. The input for $\mathsf{Shift}$ is an $x$ value of a point $p_i = (x, y_i)$ in $D$, and a new value $x'$ satisfying $x_{i-1} \leq x' \leq x_{i+1}$. The operation $\mathsf{Shift}$ replaces $p_i$ with $p = (x', y_i)$. Note that due to the constraint on $x'$, the new point $p$ can enter the segment from which $p_i$ is removed. We implement $\mathsf{Shift}(x, x')$ as follows. First, we remove $p_i$ from $D$ and insert $p = (x', y_i)$ instead (a $\mathsf{Lookup}$ operation is required to acquire $y_i$). If $M[x_i] = x_i$, we also remove $x_i$ from $M$ and add $x'$ instead. Finally, if $x_i$ was replaced in $M$ we also recursively apply $R.\mathsf{Shift}(x, x')$.

**$\mathsf{Insert}(x, y)$.** Let $x_p$ and $x_s$ be the predecessor and the successor $x$ values of $x$ in the data structure, respectively. Let $x_a = M[x_p]$ and $x_b = M[x_s]$.

- if $x_a \neq x_b$ we apply $D.\mathsf{Insert}(x, y)$. Then, we update $M$ and $R$ as follows:
  - If the segments containing $x_s$ and $x_p$ are both of length two, then we create a new segment $s = [(x, y)]$ and apply $R.\mathsf{Insert}(x, \infty)$ (recursively). We also add $x$ to $M$.
  - If one of the segments containing $x_p$ and $x_s$ are of length one, assume that the segment $s$ containing $x_p$ is the segment of length one. We update the value of $D[x_p]$ to be the proper value of $x_p$ in our data structure by applying $D[x_p] \leftarrow \mathsf{Lookup}(x_p)$. Moreover, we apply $R[x_a] \leftarrow \infty$ to guarantee Invariant 4. Note that in this operation, we add a point to a segment. If in this process the added point $(x, y)$ becomes the first point of the segment previously containing a single point $p' = (x', y')$, we need to update $M$ s.t. $x'$ is mapped to $x$ and update $R$ to contain a point with $x$ coordinate $x'$ instead of $x$. This is achieved by replacing $x$ with $x'$ in $M$ and applying $R.\mathsf{Shift}(x, x')$.

- If $x_a = x_b$, then inserting some point $(x', y')$ right before the segment $s$ containing $x_p$ is a case that we already covered. Thus, instead of inserting $(x, y)$ to $s$, we replace $x_p$ with $(x, y)$ and insert $x_p$ before $s$ following the previous cases. Let $p_p = (x_p, y_p)$ (obtained via $\mathsf{Lookup}(x_p)$). We replace $p_p$ with $p = (x, y)$ by removing the point with $x$ value $x_p$ from $D$ and insert $(x, y)$ instead. We also remove $x_p$ from $M$ and add $x$ instead. Recall that $p_s = (x_s, y_s)$ is also in the segment and we assign $\mathsf{Lookup}(x_s)$ to $D[x_s]$ and $\infty$ to $\mathbb{R}[x_a]$. We also apply $R.\mathsf{Shift}(x_p, x)$. With that, we have replaced $p_p$ with $p = (x, y)$. We proceed to insert $(x_p, y_p)$ via one of the previous cases.

$\mathsf{Remove}(x)$. Let $x' = M[x]$. Let $s_i$ be the segment containing $x$.

- If both $s_{i-1}$ and $s_{i+1}$ are of length two or do not exist, then we remove $x$ from $D$ by applying $D.\mathsf{Remove}(x)$. If $s_i$ is of length two, then it may be the case that the first point in $s_i$ was removed and is now $p' = (x', y')$. If it is the case, we replace $x$ with $x'$ in $M$ and apply $R.\mathsf{Shift}(x, x')$. If $s_i$ is of length one, the segment $s$ should be removed. We remove $R[x']$ by applying $R.\mathsf{Remove}(x')$. We also remove $x'$ from $M$.

- If $s_{i-1}$ or $s_{i+1}$ is of length one (assume that $s_{i-1}$ is of length one). Notice that by Invariant 5 $s_i$ is of length two so $s_{i-1} \cup s_i$ has exactly 3 points. Let $p'$ be the point in $s_{i-1}$ and $p''$ be the other point in $s_i$ other than $(x, y)$. We remove the point $p' = (x', y')$ from $s_{i-1}$ as described in previous cases. We then manipulate the $x$ and $y$ values of the points in $s_i$ via operations on $D$ and assign and shift operations on $R$ (as described $\mathsf{Insert}$) to replace them with $p'$ and $p''$. Notice that we already described how to remove elements in a segment of length one (due to Invariant 5 it must be the case that $s_{i-2}$ is of length two, if it exists). If the first point in $s_i$ is changed as a result of the deletion, we update $M$ and $R$ accordingly as in $\mathsf{Insert}$.

$\mathsf{AddConst}(i, j, c)$. Let $p_p = (x_p, y_p)$ and $p_s = (x_s, y_s)$ be the $x$ successor of $i$ and the $x$ predecessor of $j$ in the data structure, respectively. Let $x_a = M[x_p]$ and $x_b = M[x_s]$. Let $s_{a'}$ and $s_{b'}$ be the segments containing $x_p$ and $x_s$, respectively. First, we update the values of all (at most 4) elements in $s_{a'}$ and $s_{b'}$ that their value need to be changed. This is done via $\mathsf{Assign}(x, \mathsf{Lookup}(x)+c)$ operation for every point $p = (x, y)$ in $s_{a'}$ or in $s_{b'}$ with $x \in [i \dots j]$. Let $p_1 = (x_1, y_1)$ be the first point in $s_{a'+1}$, and $p_2 = (x_2, y_2)$ be the last point in $s_{b'-1}$, it remains to add $c$ to all elements in the range $[x_1 \dots x_2]$. Since the $y$ value of every point $(x, y)$ is represented by $\min(D[x], R[M[x]])$, we add $c$ to the two parts of the representation as follows. We add $c$ to the $y$ value for every point $p = (x, y)$ of $D$ with $x \in [x_1 \dots x_2]$. In addition, we (recursively) add $c$ to the corresponding segments via a $R.\mathsf{AddConst}(M[x_1], M[x_2], c)$ operation.

$\mathsf{Min}(i, j, c)$. As before, let $p_p = (x_p, y_p)$ and $p_s = (x_s, y_s)$ be the $x$ successor of $i$ and the $x$ predecessor of $j$ in the data structure, respectively. Let $x_a = M[x_p]$ and $x_b = M[x_s]$. Let $s_{a'}$ and $s_{b'}$ be the segments containing $x_p$ and $x_s$, respectively. First, we update the values of all (at most 4) points in $s_{a'}$ and $s_{b'}$ that their value need to be changed. This is done via $\mathsf{Assign}(\min(x, \mathsf{Lookup}(x)), x)$ for every point $(x, y)$ in $s_{a'}$ and $s_{b'}$ with $x \in [i \dots j]$. Let $p_1 = (x_1, y_1)$ be the first point in $s_{a'+1}$, and $p_2 = (x_2, y_2)$ be the last element in $s_{b'-1}$, it remains to apply the $\mathsf{Min}$ operation to all the points $(x, y)$ with $x \in [x_1 \dots x_2]$. Since the representation of the $y$ value of a point $p = (x, y)$ is $y = \min(D[x], R[M[x]])$, it is sufficient to apply $R.\mathsf{Min}(M[x_1], M[x_2], c)$. This is due to the equation $\min(\min(x, y), z) = \min(x, \min(y, z))$.

**Complexity.** A Lookup performs $O(1)$ operations on Interval-add and predecessor data structures and a recursive call to $R$. Thus, $T_{\mathsf{Lookup}}(n) = T_{\mathsf{Lookup}}(\lceil \frac{2n}{3} \rceil) + O(\log n)$, and therefore the time complexity of Lookup operation is $O(\log^2 n)$.

An Assign (resp. Shift) operation is performed using $O(1)$ operations on Interval-add and predecessor data structures, a constant number of Lookup operations and a recursive call of Assign (resp. Shift) to $R$. Thus, $T_{\mathsf{Assign}}(n) = T_{\mathsf{Assign}}(\lceil \frac{2n}{3} \rceil) + O(\log^2 n)$, and therefore the time complexity of Assign (resp. Shift) is $O(\log^3 n)$.

An operation Insert, Remove, AddConst or Min is performed by applying $O(1)$ operations on interval-add and predecessor data structures, a constant number of Lookup, Assign and Shift operations and a recursive call to $R$. Thus, for all these operations $T(n) = T(\lceil \frac{2n}{3} \rceil) + O(\log^3 n)$, and therefore their time complexity is $O(\log^4 n)$.[9]

# 5 Bounded DTW

In this section, we study the $k$-bounded version of DTW. In this version, every $\delta(a, b) \geq 1$, and we wish to compute $\mathsf{DTW}_k(S, T) = \min(\mathsf{DTW}(S, T), k + 1)$ for a given integer $k$. In this section, we prove the following theorem:

**Theorem 3.** *The Dynamic Time Warping distance of two run-length encoded strings $S$ and $T$ with $n$ and $m$ runs respectively can be computed in $\tilde{O}(nk)$ time if $\mathsf{DTW}(S, T) \leq k$.*

The key structural insight for Theorem 3 is that there is a set of $O(nk)$ blocks containing all the vertices $(x, y)$ with $\mathsf{dist}(x, y) \leq k$. Therefore, it is sufficient to process only those blocks instead of the entire grid. Informally, the set of $O(nk)$ blocks is a band of width $\Theta(k)$ around the main diagonal of blocks. This property holds since a path to a vertex outside the band requires $\Omega(k)$ orthogonal steps between blocks. Note that since every $\delta(a, b) \geq 1$, at least one of any two orthogonally adjacent blocks is a non-zero block, and the part of the path that goes through this block must incur a cost of at least 1. Formally:

**Claim 2.** *Let $(x, y)$ be a vertex in the alignment graph. Let $B_{i,j}$ be the block containing $(x, y)$. If $|j - i| > 2k$, then $\mathsf{dist}(x, y) > k$.*

*Proof.* We assume without loss of generality that $j - i > 2k$. Let $p$ be a path from $(0, 0)$ to $(x, y)$. Let $P = B_{i_1,j_1}, B_{i_2,j_2} \dots B_{i_{|P|},j_{|P|}}$ be the sequence of blocks visited by $p$ (where $B_{i_1,j_1} = B_{0,0}$ and $B_{i_{|P|},j_{|P|}} = B_{i,j}$). Note that for every $a \in [1 \dots |P| - 1]$ we have $(i_{a+1}, j_{a+1}) \in \{(i_a + 1, j_a), (i_a, j_a + 1), (i_a + 1, j_a + 1)\}$. Since $j_{|P|} - i_{|P|} > 2k$, and $i_1 - j_1 = 0$, there must be at least $2k + 1$ values of $a \in [1 \dots |P| - 1]$ such that $(i_{a+1}, j_{a+1}) = (i_a, j_a + 1)$.

Consider a value of $a$ with this property. Since the $j_a$'th run and the $(j_a + 1)$'th run in $T$ are adjacent, they must consist of different symbols. It follows that either $c_{B_{i_a,j_a}} \geq 1$ or $c_{B_{i_a,j_a+1}} \geq 1$. Let $B' \in \{B_{i_a,j_a}, B_{i_a,j_a+1}\}$ be the block with non-zero weight. The fragment of $p$ that goes through $B'$ incurs a weight of at least 1. Note that every block may be associated with at most 2 different values of $a$ - once when $p$ enters the block and once when it leaves the block. Therefore, $2k + 1$ different values of $a$ indicate that the cost of $p$ is at least $k + 1$. □

We denote the set of blocks $B_{i,j}$ such that $|j - i| \leq 2k$ as the *band*. It follows directly from Claim 2 that if $\mathsf{dist}(x, y) \leq k$ for some vertex $(x, y)$ then there is a shortest path from $(0, 0)$ to $(x, y)$ that uses only the vertices of the band. Therefore, we can set the weight of every block outside of the band to $\infty$. Then, instead of processing all the blocks, we only process the blocks of the band.

[9] We did not attempt to optimize the degree of the polylog. Some of the log factors can be easily avoided.

Any block not in the band is considered vacuously processed. Before processing a block with an input that is not in the band, we initialize the values in the frontier corresponding to this input to $\infty$. This is implemented by an $\mathsf{AddConst}(i, j, \infty)$ for the appropriate interval $[i, j]$. Note that with this assignment, the inputs have the same values as if the algorithm would have processed all the blocks. Finally, the algorithm reports $\mathsf{DTW}_k(S, T) = \min(\mathsf{dist}(N, M), k + 1)$.

## References


[1] Segment tree beats. https://codeforces.com/blog/entry/57319.

[2] John Aach and George M. Church. Aligning gene expression time series with time warping algorithms. *Bioinformatics*, 17(6):495–508, 2001.

[3] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In *56th FOCS*, pages 59–78, 2015.

[4] Pankaj K. Agarwal, Kyle Fox, Jiangwei Pan, and Rex Ying. Approximating dynamic time warping and edit distance for a pair of point sequences. In *32nd SoCG*, pages 14–18, 2016.

[5] Saeed Reza Aghabozorgi, Ali Seyed Shirkhorshidi, and Ying Wah Teh. Time-series clustering - A decade review. *Inf. Syst.*, 53:16–38, 2015.

[6] Alberto Apostolico, Gad M. Landau, and Steven Skiena. Matching for run-length encoded strings. *Journal of Complexity*, 15(1):4–16, 1999.

[7] 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.

[8] Anthony J. Bagnall, Jason Lines, Aaron Bostrom, James Large, and Eamonn J. Keogh. The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances. *Data Min. Knowl. Discov.*, 31(3):606–660, 2017.

[9] Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *56th FOCS*, pages 79–97. IEEE, 2015.

[10] Horst Bunke and János Csirik. Edit distance of run-length coded strings. In *1992 ACM/SIGAPP Symposium on Applied computing: Technological challenges of the 1990's*, pages 137–143, 1992.

[11] 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.

[12] Raphaël Clifford, Pawel Gawrychowski, Tomasz Kociumaka, Daniel P. Martin, and Przemyslaw Uznanski. RLE edit distance in near optimal time. In *44th MFCS*, pages 66:1–66:13, 2019.

[13] Debarati Das, Jacob Gilbert, MohammadTaghi Hajiaghayi, Tomasz Kociumaka, and Barna Saha. Weighted edit distance computation: Strings, trees, and dyck. In *55th STOC*, pages 377–390, 2023.

[14] Vincent Froese, Brijnesh J. Jain, Maciej Rymar, and Mathias Weller. Fast exact dynamic time warping on run-length encoded time series. *Algorithmica*, 85(2):492–508, 2022.

[15] Pawel Gawrychowski and Yanir Edri. private communication. 2016.

[16] Omer Gold and Micha Sharir. Dynamic time warping and geometric edit distance: Breaking the quadratic barrier. In *44th ICALP*, volume 80, pages 25:1–25:14, 2017.

[17] Garance Gourdel, Anne Driemel, Pierre Peterlongo, and Tatiana Starikovskaya. Pattern matching under DTW distance. In *29th SPIRE*, pages 315–330, 2022.

[18] Guan-Shieng Huang, Jia Jie Liu, and Yue-Li Wang. Sequence alignment algorithms for run-length-encoded strings. In *14th COCOON*, volume 5092, pages 319–330, 2008.

[19] Youngha Hwang and Saul B. Gelfand. Fast sparse dynamic time warping. In *26th ICPR*, pages 3872–3877, 2022.

[20] Philip N. Klein and Shay Mozes. Optimization algorithms for planar graphs. http://planarity.org. Book draft.

[21] William Kuszmaul. Dynamic time warping in strongly subquadratic time: Algorithms for the low-distance regime and approximate evaluation. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th ICALP*, volume 132, pages 80:1–80:15, 2019.

[22] William Kuszmaul. Binary dynamic time warping in linear time. arXiv preprint, 2021.

[23] Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. *J. Comput. Syst. Sci.*, 37(1):63–78, 1988.

[24] T. Warren Liao. Clustering of time series data - a survey. *Pattern Recognit*, 38(11):1857–1874, 2005.

[25] 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.

[26] Alexander De Luca, Alina Hang, Frederik Brudy, Christian Lindner, and Heinrich Hussmann. Touch me once and i know it's you!: implicit authentication based on touch screen patterns. In *SIGCHI Conference on Human Factors in Computing Systems*, pages 987–996. ACM, 2012.

[27] Veli Mäkinen, Esko Ukkonen, and Gonzalo Navarro. Approximate matching of run-length compressed strings. *Algorithmica*, 35(4):347–369, 2003.

[28] J. Mitchell. *A geometric shortest path problem, with application to computing a longest common subsequence in run-length encoded strings.* Tech-nical Report, Department of Applied Mathemat-ics, SUNY StonyBrook, NY, 1997.

[29] Lindasalwa Muda, Mumtaj Begam, and Irraivan Elamvazuthi. Voice recognition algorithms using mel frequency cepstral coefficient (MFCC) and dynamic time warping (DTW) techniques. arXiv preprint, 2010.

[30] Abdullah Mueen, Nikan Chavoshi, Noor Abu-El-Rub, Hossein Hamooni, and Amanda J. Minnich. Awarp: Fast warping distance for sparse time series. In *16th ICDM*, pages 350–359, 2016.

[31] Abdullah Mueen, Nikan Chavoshi, Noor Abu-El-Rub, Hossein Hamooni, Amanda J. Minnich, and Jonathan MacCarthy. Speeding up dynamic time warping distance for sparse time series data. *Knowl. Inf. Syst.*, 54(1):237–263, 2018.

[32] Mario E. Munich and Pietro Perona. Continuous dynamic time warping for translation-invariant curve alignment with applications to signature verification. In *7th ICCV*, pages 108–115, 1999.

[33] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[34] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[35] Yoshifumi Sakai and Shunsuke Inenaga. A reduction of the dynamic time warping distance to the longest increasing subsequence length. In *31st ISAAC*, pages 6:1–6:16, 2020.

[36] Yoshifumi Sakai and Shunsuke Inenaga. A faster reduction of the dynamic time warping distance to the longest increasing subsequence length. 2022.

[37] Nathan Schaar, Vincent Froese, and Rolf Niedermeier. Faster binary mean computation under dynamic time warping. In *31st CPM*, pages 28:1–28:13, 2020.

[38] Taras K. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics*, 4(1):52–57, 1968.

[39] Xiaoyue Wang, Abdullah Mueen, Hui Ding, Goce Trajcevski, Peter Scheuermann, and Eamonn J. Keogh. Experimental comparison of representation methods and distance measures for time series data. *Data Min. Knowl. Discov.*, 26(2):275–309, 2013.

[40] Zoe Xi and William Kuszmaul. Approximating dynamic time warping distance between run-length encoded strings. In *30th ESA*, pages 90:1–90:19, 2022.

[41] Rex Ying, Jiangwei Pan, Kyle Fox, and Pankaj K. Agarwal. A simple efficient approximation algorithm for dynamic time warping. In *24th ACM SIGSPATIAL*, pages 21:1–21:10, 2016.

[42] Yunyue Zhu and Dennis Shasha. Warping indexes with envelope transforms for query by humming. In *22nd ACM SIGMOD*, pages 181–192, 2003.

# A Missing Proofs

**Proof of Lemma 1.** We prove the case where $(x, y), (x, y+1) \in B$, the other two cases are similar. All edges entering $(x+1, y+1)$ have the same weight $c$. We claim that the values $\mathsf{dist}(x, y)$ are weakly monotone along every row and column in a block. This implies that $\mathsf{dist}(x, y) + c$ (the path to $(x+1, y+1)$ through $(x, y)$)) is not larger than $\mathsf{dist}(x, y+1) + c$ (the path to $(x+1, y+1)$ through $(x, y+1)$)).

To see why the values $\mathsf{dist}(x, y)$ are weakly monotone along every row and column in a block, consider two vertices $(x, y), (x, y+1)$ in the same block $B$. We show that $\mathsf{dist}(x, y) \leq \mathsf{dist}(x, y+1)$ (a symmetric proof shows that $\mathsf{dist}(x, y) \leq \mathsf{dist}(x+1, y)$). Let $P$ be a shortest path from $(0, 0)$ to $(x, y+1)$. Let $(x', y)$ be the last vertex in $P$ with second coordinate $y$. If $x' = x$ then clearly

$\mathsf{dist}(x,y) \le \mathsf{dist}(x,y+1)$. Otherwise, $x' < x$. Then, we can assume that the suffix of $P$ starting from $(x',y)$ is composed of a single diagonal edge followed by zero or more vertical edges (since a horizontal edge followed by a vertical edge is always not shorter than just using the diagonal edge). Now consider the path $P'$ from $(0,0)$ to $(x,y)$ that is identical to $P$ until $(x',y)$ and from $(x',y)$ continues vertically. Paths $P'$ and $P$ only differ in the suffix from $(x',y)$. But in this suffix they both use the same number of edges $x$-$x'$ and the same edge weights (since all edges in a block have the same weight). This means that $P'$ and $P$ have the same length and thus $\mathsf{dist}(x,y) \le \mathsf{dist}(x,y+1)$).

**Lemma 13.** *For every $x \in [i \dots j]$ the procedure* $\mathsf{LeftLinearWave}(i,j,\alpha)$ *described in Section 3.1 assigns* $A[x] \leftarrow \min_{t \le x}(A[t] + (x-t)\alpha)$.

*Proof.* Let $L(k)$ be the value assigned to $A[k]$ by $\mathsf{LeftLinearWave}(i,j,\alpha)$, we first claim that:

$$L(k) = \begin{cases} A[i] & k = i \\ \min(A[k], L(k-1)+\alpha) & k \in [i+1 \dots j] \end{cases}$$

We prove this by induction on $k-i$. For $k-i=0$, $\mathsf{LeftLinearWave}(i,j,\alpha)$ assigns $A[i] \leftarrow \min_{t\in[i\dots i]}(A[t]+(i-t)\alpha) = A[i] = L(i)$ as required. For $k-i>0$:

$$\begin{aligned} A[k] &\leftarrow \min_{t\in[i\dots k]}(A[t]+(k-t)\alpha) = \min\big(\min_{t\in[i\dots k-1]}(A[t]+(k-t)\alpha), A[k]\big) \\ &= \min\big(\min_{t\in[i\dots k-1]}(A[t]+(k-1-t)\alpha)+\alpha, A[k]\big) = \min(L(k-1)+\alpha, A[k]) = L(k). \end{aligned}$$

By the above, we need to prove that after $\mathsf{LeftLinearWave}$ is applied, $A[x] = L[x]$ for every $x \in [i \dots j]$ (clearly, the operation $\mathsf{LeftLinearWave}$ does not change $A[x]$ for $x \notin [i \dots j]$). We prove this claim by induction on $x \in [i \dots j]$. For $x = i$ the claim holds since $L[i] = A[i]$ and indeed, $\mathsf{LeftLinearWave}$ does not change $A[i]$. Otherwise, assume that the claim holds for $x-1 \in [i \dots j-1]$. Let $z \in [q \dots b]$ be the maximal value such that a ray shooting process started from $p_z = (x_z, y_z)$ and $x_z < x$. Let $w \in [z+1 \dots |\mathcal{P}|$ be minimal index such that $p_w$ is below the ray $r_z$ with slope $\alpha$ shot from $p_z$. Let $p' = (x', y')$ be the intersection point of $\ell_{w-1}$ and the ray with slope $\alpha$ shot from $p_z$. We distinguish between two cases.

- **Case 1:** $x' \ge x-1$**.** In this case, the linear segment containing $x-1$ after $\mathsf{LeftLinearWave}(i,j,\alpha)$ is applied is a sub-segment of the ray $r_z$. Therefore, the value assigned to $A[x-1]$ by the procedure is $r_z(x-1)$. According to the induction hypothesis, we have $r_z(x-1) = L[x-1]$. We consider two cases regarding the value $A[x]$ before the update is applied.
  - $A[x] \ge L[x-1]+\alpha$. In this case, the point $(x, A[x])$ is not below the ray $r_z$ and $p'$ must be to the left of $(x, A[x])$. It follows that after the procedure is applied, $x$ is also on a linear segment that is a sub-segment of $r_z$ and therefore $A[x] = r_z(x)$. Indeed, in this case $L[x] = \mathsf{Min}(L[x-1]+\alpha, A[x]) = L[x-1]+\alpha = r_z(x-1)+\alpha = r_z(x)$ as required.
  - $A[x] < L[x-1]+\alpha$ Note that in this case, we have $p_w = (x, A[x])$. Recall that when $p_w$ is met in the ray shooting process, a segment connecting $(x-1, r_z(x-1))$ and $(x, A[x])$ is created. Therefore, the value of $A[x]$ is not changed by the ray shooting process. Indeed, in this case we have $L[x] = \min(L[x-1]+\alpha, A[x]) = A[x]$.
- **Case 2:** $x' < x-1$**.** In this case, the procedure $\mathsf{LeftLinearWave}(i,j,\alpha)$ did not change the value of $x-1$ and we have $L[x-1] = A[x-1]$ from the induction hypothesis. Specifically, the ray shooting process from $q_z$ terminated, and the next ray shooting process, if a necessary one exists, is from a point $p_{z'}$ with $x_{z'} > x-1$. This implies that the slope of the linear segment

containing $x-1$ is at most $\alpha$. Therefore, we must have $A[x] < A[x-1]+\alpha = L[x-1]+\alpha$ and as a result $L[x] = A[x]$. Whether or not a ray shooting starts from $(x, A[x])$, the value of $A[x]$ is not changed by $\mathsf{LeftLinearWave}(i, j, \alpha)$. If a ray shooting does not start from $(x, A[x])$ - the linear segment containing $x$ is not affected by $\mathsf{LeftLinearWave}(i.j.\alpha)$ as no ray shooting process interacted with it. If a ray shooting process starts from $(x, A[x])$, the linear segment containing $x$ after the update is applied will be a sub-segment of a ray $\ell^*$ starting from $(x, A[x])$, and clearly $\ell^*(x) = A[x]$ as required. □

**Proof for Lemma 5.** Assume to the contrary that a ray shooting process starts at a passive point $p_z \neq p_b$. If $p_z$ is the first point from the right where a ray shooting starts, then $z$ is the maximal index in $[a \dots b]$ with $\alpha_{z-1} < -\alpha$. But since $p_z$ is passive, we have $-\alpha > \alpha_{z-1} \geq \alpha_z$, contradicting the maximality of $z$ (note that $p_z \neq p_b$ so $z+1 \in [a \dots b]$).

Otherwise, let $p_q$ be the last point before $p_z$ from which a ray shooting process occurred. Let $p_{q'}$ be the first point below the ray shot from $p_q$. Since $p_z$ is the next point from which a ray is shot, $z$ is the rightmost point in $[a \dots q']$ with $\alpha_{z-1} < -\alpha$. Since $p_z$ is passive, we have $-\alpha > \alpha_{z-1} \geq \alpha_z$. If $z \neq q'$, we have $z+1 \in [a \dots q']$, a contradiction to the maximality of $z$. Otherwise, $p_z = p_{q'}$ is the first point below the ray with slope $-\alpha$ shot from $p_q$. It follows from $\alpha_z < -\alpha$ that $p_{z+1}$ is below the ray as well, and a contradiction to $p_z = p_{q'}$ being the first point below the ray.

*Proof of Lemma Lemma 2.* We obtain $\mathsf{F}_{t+1}$ from $F_t$ in two phases. Let $B$ be the block processed at step $t$, and suppose $B$ corresponds to runs $S[i_1 \dots i_2]$ and $T[j_1 \dots j_2]$. Then $\mathsf{F}_t$ and $\mathsf{F}_{t+1}$ differ only in the range $[a \dots b]$ where $a = j_1 - i_2$ and $b = j_2 - i_1$ (see Fig. 7). In phase I we apply a sequence of range operations on $\mathsf{F}_t$ in order to obtain $\mathsf{F}$, which is defined to be identical to $\mathsf{F}_t$ except that the inputs of $B$ replace the corresponding outputs of $B$'s entering blocks. Formally, $\mathsf{F}[d] = \mathsf{F}_t[d]$ for every $d \notin [a \dots b]$, and for $d \in [a \dots b]$, $\mathsf{F}[d] = \mathsf{dist}(x, y)$ where $(x, y)$ is the input node of $B$ with $y - x = d$. In phase II we apply another sequence of range operations on $\mathsf{F}$ to obtain $\mathsf{F}_{t+1}$, which is identical to $\mathsf{F}$ except that the inputs of $B$ are replaced by the outputs of $B$.

The height of $B$ is denoted $h = i_2 - i_1 + 1$ and the width of $B$ is $w = j_2 - j_1 + 1$. We denote the first row of $B$ as $U_B$, the first column of $B$ as $L_B$, the last row of $B$ as $D_B$, and the last column of $B$ as $R_B$ (see Fig. 1). We note that the input nodes of $B$ are $L_B \cup U_B$ and the output nodes are $D_B \cup R_B$. We denote the entering blocks of $B$ as $L = B_{i,j-1}$, $C = B_{i-1,j-1}$ and $U = B_{i-1,j}$. We define $R_L$ as the last column of $L$, and $D_U$ as the last row of $U$. Notice that the values of $\mathsf{dist}(x, y)$ for vertices of $R_L$ are stored in $\mathsf{F}_t[a-1 \dots a+h-2]$, and the values of $\mathsf{dist}(x, y)$ for vertices of $D_U$ are stored in $\mathsf{F}_t[b-w+2 \dots b+1]$ (see Fig. 7).

**Phase I - computing $\mathsf{F}$ from $\mathsf{F}_t$.** We begin by computing $\mathsf{dist}(i_1, j_1)$. Recall that $\mathsf{dist}(i_1-1, j_1-1)$ is stored in $\mathsf{F}_t[z]$ where $z = a+h-1 = b-w+1 = j_1 - i_1$. By Eq. (1), we have $\mathsf{dist}(i_1, j_1) = c_B + \min\{\mathsf{F}_t[z-1], \mathsf{F}_t[z], \mathsf{F}_t[z+1]\}$. Let $\tilde{\mathsf{F}}_t$ be $\mathsf{F}_t$ with the assignment $\tilde{\mathsf{F}}_t[z] \leftarrow \mathsf{dist}(i_1, j_1) - c_B$. The definition of $\tilde{\mathsf{F}}$ is motivated by the following lemma.

**Lemma 14.** *For every $k \in [z \dots b]$, $\mathsf{F}[k] = c_B + \min_{i \in [z \dots k]} \left(\tilde{\mathsf{F}}_t[i] + (k-i)c_B\right)$.*

*Proof.* For $k \in [z \dots b]$, let $y = k - z$. Note that $\mathsf{F}[k]$ should be assigned $\mathsf{dist}(i_1, j_1 + y)$. We prove the claim by induction on $y$. For $y = 0$, we need to prove that $\mathsf{F}[z] = \mathsf{dist}(i_1, j_1) = c_B + \tilde{\mathsf{F}}_t[z]$. This follows from the fact that $\tilde{\mathsf{F}}_t[z] = \mathsf{dist}(i_1, j_1) - c_B$. For the inductive step, we need to show that $\mathsf{F}[k] = \mathsf{dist}(i_1, j_1 + y) = \min_{i \in [z \dots k]} \left(\tilde{\mathsf{F}}_t[i] + (k-i)c_B\right) + c_B$.

By Lemma 1, there are two options: (i) The shortest path to $(i_1, j_1+y)$ goes diagonally through $(i_1 - 1, j_1 + y - 1)$. Then, its length is $\mathsf{dist}(i_1 - 1, j_1 + y - 1) + c_B = \tilde{\mathsf{F}}_t[k] + c_B$ from the definition

of $\tilde{\mathsf{F}}_t$. (ii) The shortest path to $(i_1, j_1 + y)$ goes horizontally through $(i_1, j_1 + y - 1)$, and it follows that is length is $c_B + \mathsf{F}[k-1]$. Then, by the induction hypothesis, the length of this path is

$$\left(c_B + \min_{i \in [z \ldots k-1]} \left(\tilde{\mathsf{F}}_t[i] + ((k-1) - i)c_B\right)\right) + c_B = \min_{i \in [z \ldots k-1]} \left(\tilde{\mathsf{F}}_t[i] + (k-i)c_B\right) + c_B.$$

Taking the minimum between (i) and (ii) yields the lemma. □

It directly follows from Lemma 14 that $\mathsf{F}_t[z \ldots b]$ can be turned into $\mathsf{F}[z \ldots b]$ by applying the following range operations, in order:

1. $\mathsf{AddConst}(z, z, \mathsf{dist}(i_1, j_1) - c_B - \mathsf{F}_t[z])$.
2. $\mathsf{LeftLinearWave}(z, b, c_B)$.
3. $\mathsf{AddConst}(z, b, c_B)$.

The first operation turns $\mathsf{F}_t$ into $\tilde{\mathsf{F}}_t$ and the other two operations turn $\tilde{\mathsf{F}}_t[z \ldots b]$ into $\mathsf{F}$ by applying the formula given in Lemma 14. In a similar way, we can prove that $\mathsf{F}[a \ldots z]$ can be obtained from $\mathsf{F}_t$. This time, using $\mathsf{RightLinearWave}(i, j, c)$.

**Phase II - computing $\mathsf{F}_{t+1}$ from $\mathsf{F}$.** The following lemma shows that $\mathsf{F}_{t+1}$ can be obtained by applying $O(1)$ range operations on $\mathsf{F}$.

**Lemma 15.** *Let $d_1 = j_1 - i_1$ and $d_2 = j_2 - i_2$. For every $d \in [a \ldots b]$:*

$$\mathsf{F}_{t+1}[d] = \begin{cases} \mathsf{F}[d] + (d-a)c_B, & \text{if } d \in [a \ldots \min(d_1, d_2)) \\ \mathsf{F}[d] + \min(w, h)c_B, & \text{if } d \in [\min(d_1, d_2) \ldots \max(d_1, d_2)] \\ \mathsf{F}[d] + (b-d)c_B, & \text{if } d \in (\max(d_1, d_2) \ldots b] \end{cases}$$

*Proof.* We begin with the following claim:

**Claim 3.** *Let $(x, y)$ be a vertex in block $B$ and let $(x', y')$ be the input vertex of $B$ with $y' - x' = y - x$, then $\mathsf{dist}(x, y) = \mathsf{dist}(x', y') + c_B \cdot (x - x')$.*

*Proof.* We prove by induction on $d = x - x'$ that there is a shortest path from $(0, 0)$ to $(x, y)$ that visits $(x', y')$ and then uses $x' - x$ consecutive diagonal edges. If $d = 0$, this holds trivially. Otherwise, since $d \geq 1$, the vertices $(x, y)$ and $(x-1, y-1)$ are both in the same block $B$. It follows from Lemma 1 that there is a shortest path to $(x, y)$ via $(x-1, y-1)$, which by the induction hypothesis goes through $(x', y')$ and then uses only diagonal edges. □

Let $(x, y)$ be an output of $B$ on diagonal $d = y - x$. Let $(x', y')$ be the input of $B$ on the same diagonal $d$. Therefore, the following holds (see Fig. 7):

1. If $d \in [a \ldots \min(d_1, d_2))$, we have $x - x' = d - a$.
2. If $d \in [\min(d_1, d_2) \ldots \max(d_1, d_2)]$, we have $x - x' = \min(w, h)$.
3. If $d \in (\max(d_1, d_2) \ldots b]$, we have $x - x' = b - d$.

Combined with Claim 3, this proves Lemma 15. □

From Lemma 15, $\mathsf{F}$ can be turned into $\mathsf{F}_{t+1}$ by applying:

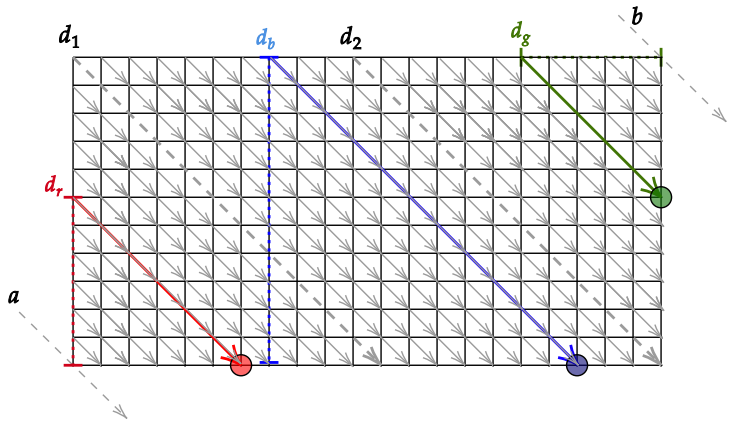


Figure 7: A block $B$. The red vertex is an output vertex on a diagonal $d_r$ with $d_r \in [a \dots d_1]$, and the number of diagonal steps from the matching input on the same diagonal to the red vertex is $d_r - a$. Similarly, the blue vertex is on a diagonal $d_b \in [d_1 \dots d_2]$, and the green vertex is on a diagonal $d_g \in [d_2 \dots b]$.

1. $\mathsf{AddConst}(a, \min(d_1, d_2) - 1, -a \cdot c_B)$.
2. $\mathsf{AddGradient}(a, \min(d_1, d_2) - 1, c_B)$.
3. $\mathsf{AddConst}(\min(d_1, d_2), \max(d_1, d_2), \min(w, h)c_B)$.
4. $\mathsf{AddConst}(\max(d_1, d_2) + 1, b, b \cdot c_B)$.
5. $\mathsf{AddGradient}(\max(d_1, d_2) + 1, b, -c_B)$.

This concludes Phase II and the proof of Lemma 2. As for the parameters of the required range operations, the values $d_1, d_2, w, h, a, b, c_B$, and $z$ can all be calculated in advance for every block $B$ in $O(nm)$ time in a straightforward manner. □