Optimal Dynamic Time Warping on Run-Length Encoded Strings

Itai Boneh
Reichman University, Herzliya, Israel
University of Haifa, Israel

Shay Golan
Reichman University, Herzliya, Israel
University of Haifa, Israel

Shay Mozes
Reichman University, Herzliya, Israel

Oren Weimann
University of Haifa, Israel

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 \( O(n^3k) \) and \( O(Nk) \) time exact algorithms and the \( \tilde{O}(nk) \) time approximation algorithm.

2012 ACM Subject Classification Theory of computation → Pattern matching; Theory of computation → Shortest paths

Keywords and phrases Dynamic time warping, Fréchet distance, edit distance, run-length encoding

Digital Object Identifier 10.4230/LIPIcs.ICALP.2024.30

Category Track A: Algorithms, Complexity and Games


Funding Israel Science Foundation grant 810/21

1 Introduction

Dynamic Time Warping (DTW) [39] is one of the most popular methods for comparing time-series (see e.g. [2, 5, 8, 25, 27, 30, 33, 40, 43]). 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 \( S = aaaaabbbbaaaaaa \) 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

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

where \( S' \) and \( T' \) range over all time-warps of \( S \) and \( T \) respectively.

In 1968, Vintzyuk [39] 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 [17], 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) [3,10].

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 [23,38], approximation algorithms [4,22,42], the low distance regime [22], sparse inputs [20,31,32], and reductions to other similarity measures [22,36,37]. 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. His algorithm is one only known to be possible if we are willing to settle for an \((1+\varepsilon)\)-approximation [41]. 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 \( \text{DTW}(i, j) = \text{DTW}(S[1 \ldots i], T[1 \ldots j]) \), then \( \text{DTW}(0,0) = 0 \), \( \text{DTW}(i,0) = \text{DTW}(0,j) = \infty \) for every \( i > 0 \) and \( j > 0 \), and otherwise:

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

(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 \( V = [0 \ldots N] \times [0 \ldots M] \). 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 \( \text{dist}(i,j) \).

1 Clearly, \( \text{DTW}(i,j) = \text{dist}(i,j) \). Therefore, \( \text{DTW}(S,T) = \text{dist}(N,M) \) 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.

1 Abusing notation, we will later also use \( \text{dist}((x,y),(x',y')) \) to denote the distance from vertex \((x,y)\) to vertex \((x',y')\).
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 [13, 15, 22].

In order to go below \(O(Nm + Mn)\), in [15] 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 [41] 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 [35, 39]. 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, 10, 22], 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, 11–13, 19, 26, 28, 29]. Specifically, Clifford et. al. [13] 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 \(O(polylog(mn))\) time from the representation of the input. This implies an \(\tilde{O}(nm)\) time algorithm for edit distance.

In [41], Xi and Kusza explained 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. [13] 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 Kusza [41]: (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 [13] works only for Levenshtein distance (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 [22].

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).**

- **Lookup**\((i)\) - return \(A[i]\).
- **AddConst**\((i, j, c)\) - for every \(k \in \{i, \ldots, j\}\), set \(A[k] \leftarrow A[k] + c\).
- **AddGradient**\((i, j, g)\) - for every \(k \in \{i, \ldots, j\}\), set \(A[k] \leftarrow A[k] + k \cdot g\).
- **LeftLinearWave**\((i, j, \alpha)\) - for every \(k \in \{i, \ldots, j\}\), set \(A[k] \leftarrow \min_{t \in [i \ldots k]} (A[t] + (k - t)\alpha)\).
- **RightLinearWave**\((i, j, \alpha)\) - for every \(k \in \{i, \ldots, j\}\), set \(A[k] \leftarrow \min_{t \in [k \ldots j]} (A[t] + (t - k)\alpha)\).
In Section 3, we show our main technical contribution:

**Theorem 4.** Performing $s$ range operations of Definition 3 can be done in amortized $O(\text{polylog}(s))$ time per operation.

The proof of Theorem 4 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 shifting 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 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: $\text{AddConst}(i,j,c)$ - for every $k \in [i...j]$ set $A[k] \leftarrow A[k] + c$, and $\text{Min}(i,j,c)$ - for every $k \in [i...j]$ 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. We show a different and worst-case polylog time solution.2

**Implications for low regime DTW.** In Section 4, we show that our $O(n \cdot k)$ algorithm for computing $\text{DTW}(S,T)$ immediately implies an $\tilde{O}(n \cdot k)$ time algorithm where $k = \text{DTW}(S,T)$. 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 [22], the $\tilde{O}(n^2 \cdot k)$ algorithm of [15], and the $O(n \cdot k)$ time approximation algorithm of [41] (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 [13] runs in $O(n \cdot k)$ 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 [34] and Landau-Vishkin [24] for unweighted edit distance, and the very recent $O(N + k^3)$ time algorithm for weighted edit distance [14].

---

2 We note that the solution in [1] also supports range-sum queries and for such a conditional lower bound (from the Online Matrix-Vector Multiplication (OMV) problem) is known [16]. The lower bound implies that worst-case operations unlikely to be possible in $O(n^{1/2 - \epsilon})$ time. We are able to circumvent this lower bound because we only support lookups, but not range-sum queries.
Implications for pattern matching DTW. The pattern matching version of DTW asks to compute, for every index \( j \in [1 \ldots |T|] \) the value \( \min_{i \in [1 \ldots j]}(D(T,S[i \ldots j])) \). In \[18\], 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 \( \min_{i \in [1 \ldots j]}(D(T,S[i \ldots j])) \leq k \). The key ingredient of these algorithms (See \[18, Lemma 2\]) is a dynamic programming formula that is identical to Equation (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 4 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 \( S[i_1 \ldots i_2] \) and \( T[j_1 \ldots j_2] \) 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 \( a \in [i_1 \ldots i_2] \) and \( b \in [j_1 \ldots j_2] \). 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 full version).

Lemma 5. 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 \((x+1,y) \).
- 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 each step, the algorithm can process any block \( B \) as long as all its entering blocks have already been processed. When block \( B \) is processed, the algorithm computes \( \text{dist}(x,y) \) for every output vertex \((x,y) \) of \( B \). After processing block \( B \), we say that the output vertices of \( B \) are resolved. At each 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 \( d \in [-N \ldots M] \), 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 \( F_t[-N \ldots M] \) where \( F_t[d] = \text{dist}(x,y) \) such that vertex \((x,y) \) belongs to the current frontier and \( y - x = d \). In the full version of this paper (\[9\]), we prove the following lemma.

Lemma 6. \( F_{t+1} \) can be obtained by using \( O(1) \) range operations (Definition 3) on \( F_t \).

In the rest of this section, we prove that Lemma 6 implies our main result:
According to Equation (1), we need to set

\[
\text{The set } \{(x, y) | x \in [0, \ldots, N], y \in [M]\}.
\]

Here we use

\[
\text{Theorem 7. The Dynamic Time Warping distance of two run-length encoded strings } S \text{ and } T \text{ with } n \text{ and } m \text{ runs respectively can be computed in } O(nm) \text{ time.}
\]

**Proof.** We initialize the data structure of Theorem 4 as an array of length \(nM + 1\). We treat the indices of \(A\) as if they are in \([-N \ldots M]\)\(^3\). Initially, the frontier consists of the vertices \((x, 0)\) with \(x \in [0 \ldots N]\) and \((0, y)\) with \(y \in [M]\). We start by turning \(A\) into \(F_0\). According to Equation (1), we need to set \(A[0] = 0\) and \(A[i] = \infty\) for \(i \neq 0\). This can be done by applying \(\text{AddConst}(1, M, \infty)\) and \(\text{AddConst}(-N, -1, \infty)\).

The algorithm runs in \(nm\) iterations. At the beginning of iteration \(t\), we have \(A = F_t\). The algorithm picks any block \(B\) whose entering blocks have already been processed, and applies \(O(1)\) range operations (due to Lemma 6) on \(A\) in order to obtain \(A = F_{t+1}\). After the last iteration, it is guaranteed that the block \(B_{n,m}\) has been processed. Therefore, \(F_{nm}[M-N] = \text{DTW}(S,T)\). Every iteration requires \(O(1)\) range operations each in \(O(\text{polylog}(nm))\) time, so overall the algorithm performs \(O(nm)\) operations in total \(O(nm)\) time. \(\blacksquare\)

### 3 Implementing the Range Operations

In this section, we prove Theorem 4. We view the array \(A\) as a piecewise linear function. We associate with \(A\) a set \(\mathcal{P} = \{p_i = (x_1, y_1), p_2 = (x_2, y_2), \ldots\}\) 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 \(t_i(x) = \alpha_i x + \beta_i\) 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.\(^5\)

---

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

\(^4\) 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
Lemma 8 (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.

- **Lookup**($x$) - return the second coordinate of the point with first coordinate $x$, if exists.
- **Insert**($x, y$) - insert the point $(x, y)$.
- **Remove**($x$) - remove the point with first coordinate $x$, if exists.
- **AddToRange**($i, j, c$) - for every point $(x, y)$ with $x \in [i \ldots j]$ set $y \leftarrow y + c$.
- **nextGT**($x, y$) - return the point $p' = (x', y')$ with smallest $x' > x$ among points with $y' > y$.
- **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 **AddConst**($i, j, c$) (see Figure 2) is to break the segment containing $i$ into at most three linear segments (a prefix ending at $i-1$, a segment $[i-1, i]$, and a suffix starting at $i$), and similarly for the segment containing $j$. Thus, to apply **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 **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 **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 **AddGradient**($i, j, g$) is similar. The only difference is that we invoke **AddToRange**($i, j, g$) on $D_\alpha$ instead of on $D_\beta$ because the slopes of the segments are shifted rather than their values.

The challenge is thus in supporting **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 $[i \ldots j]$ are $\ell_a, \ell_{a+1} \ldots \ell_b$.

If $\alpha_a \leq \alpha$ then the segment $\ell_a$ is not affected by the linear wave. This is because for every $k \in [x_a \ldots x_{a+1}]$, the linear wave assigns

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

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


\footnote{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. [21].}
Figure 2 An illustration of applying the 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 \([i ... j]\) are shifted by \(c\).

Let \(z \in [a ... b]\) be the minimum index such that \(\alpha_z > \alpha\). By the same reasoning, none of the segments \(\ell_a, \ell_{a+1}, ..., \ell_{z-1}\) are 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 \text{LeftLinearWave} by a ray shooting process from \(p_z\) (See Figure 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') = ([x^*], r_z([x^*]), and it replaces all the points \(p_w\) for \(w \in (z ... z')\). 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 \text{LeftLinearWave}(i, j, \alpha)\). The segments before \(p_a\) are not affected. The segments between \(p_a\) and \(p_{z'}\) are affected. Namely, a ray \(r_{x'}\) with slope \(\alpha\) (in dashed blue) is shot from \(p_a\) and intersects at point \(p^* = (x^*, y^*)\). The new endpoint of \(\ell_a\) becomes \(p'\) and all the segments between \(p_a\) 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 \text{LeftLinearWave}(i, j, \alpha)\) on the remaining part of \(A\), namely on \(A[x'...j]\) 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

\footnote{Note that in Figure 3 and in all subsequent figures we indicate the slope \(\alpha\) of the ray \(r_{x'}\) 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.}
α is shot from the next \( p_w \) with \( \alpha_w > \alpha \), and so on. In the full version of this paper ([9]) we formally prove that the above characterization indeed represents the new values of \( A[i...j] \).

We now describe a naive, non-efficient implementation of \texttt{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 \( x_z \in [i...j] \) and \( \alpha_z \geq \alpha \) by querying \( D_a\).nextGT(i, \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 \( r_z(x_w) \leq y_w \).

If so, \( p_w \) is removed by removing \( x_w \) from \( \mathcal{P} \), \( D_3 \) and \( D_\alpha \). Otherwise, we compute \( p'' = (x^*, y^*) \), the intersection point of \( r_z \) and \( \ell_{w-1} \), and from it the points \( p' = (x', y') = ([x^*, r_z([x^*])] \) and, if \( x^* \) is not an integer, also \( p'' = ([x^*, \ell_{w-1}([x^*])] \). Then, we insert the new points \( p' \) and \( p'' \) just before \( p_w \), as discussed above for \texttt{AddConst}. The scanning then continues with another \texttt{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 \((x_b, r_z(x_b))\).

**Time Complexity.** We now analyze the time complexity of this naive implementation. Each \texttt{AddConst} and \texttt{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 \texttt{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 \texttt{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 \texttt{LeftLinearWave} operation using a constant number of updates to a separate data structure that keeps track of the delayed rays.

**Symmetry of \texttt{RightLinearWave}.** The discussion so far was focused on the \texttt{LeftLinearWave} operation. We note that the analysis of \texttt{RightLinearWave} is symmetric. In particular, the execution of \texttt{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 \( z \in [a...b] \) such that \( \alpha_z < -\alpha \) (\( p_z \) is found using \( D_\alpha\).prevLT). Note that the condition for starting a ray shooting process for \texttt{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 \texttt{LeftLinearWave}, and will comment at the very end about the minor adjustments required to also handle the symmetric \texttt{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 9 (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 \( \mathcal{P}_{\text{active}} \).
Lemma 10. Ray shootings stemming from \textit{LeftLinearWave}(i, j, \alpha) occur either at point \( p_\alpha = (i, A[i]) \) or at active points. Ray shootings stemming from \textit{RightLinearWave}(i, j, \alpha) occur either at point \( p_\alpha = (j, A[j]) \) or at active points.

\textbf{Proof.} We focus on \textit{LeftLinearWave}. The proof for \textit{RightLinearWave} is symmetric. Assume to the contrary that a ray shooting process starts at a passive point \( p_\alpha \neq p_\alpha \). If \( p_z \) is the first point where a ray shooting starts, then \( z \) is the minimal index in \([a \ldots b]\) 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 \( p_z \neq p_\alpha \) so \( z - 1 \in [a \ldots b] \)).

Otherwise, let \( p_q \) be the first 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 \([q' \ldots b]\) 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 \( z - 1 \in [q' \ldots b] \), 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. \hfill \Box

Let \( P_{\text{active}} = (q_1, q_2, \ldots) \) be the restriction of the sequence \( P \) 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 \textit{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 11. Let \( q_z = p_w \) and \( q_{z+1} = p_{w'} \) be two consecutive active points. For every \( k \in [w \ldots w'] \), the passive point \( p_k \) is not below the mega-segment connecting \( q_z \) and \( q_{z+1} \). Furthermore, \( \alpha_w \geq \gamma_z \).

\textbf{Proof.} (See Figure 4) Clearly, \( p_w \) and \( p_{w'} \) are on the mega-segment, and in particular not below it. Assume by contradiction that there is a point below the mega-segment, and let \( k' \in (w \ldots w') \) be the smallest index of such a point. Since \( p_{k'-1} \) is not below the mega-segment and \( p_{w'} \) is below the mega-segment, we must have \( \alpha_{k'-1} < \gamma_z \). Moreover, since the points \( p_k \) with \( k \in [k' \ldots w'] \) 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 \( p_{w'} \) lying on the mega-segment. Furthermore, since \( p_{w+1} \) is not below the mega-segment, we have \( \alpha_w \geq \gamma_z \). \hfill \Box

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 \( \vert P_{\text{active}} \vert \) decreases.

Lemma 12. Consider a ray shooting process starting from point \( p_w = q_z \in P_{\text{active}} \) during the application of \textit{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 \( k \in [w+1 \ldots r] \) for some \( w < r < w' \). No other points between \( p_w \) and \( p_{w'} \) are deleted by \textit{LeftLinearWave}(i, j, \alpha).
3. If \( \gamma_z \geq \alpha \) then \( q_{z+1} \) is either deleted or becomes passive.
The impossible configuration in Lemma 11. The points $q_i$ and $q_{i+1}$ 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 $q_{i+1}$) must remain below the mega-segment.

**Proof.** Let $\ell$ be the ray starting from $p_w = q_i$. 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 $\gamma_i < \alpha$. Then $q_{i+1}$ is below the ray with slope $\alpha$ starting at $q_i$. Hence the ray shooting process terminates at a point after $p_{w+1}$ and before $q_{i+1}$. Since no active points are created, the next ray will be shot from $q_i + 1$ or later, so no other points between $q_i$ and $q_{i+1}$ are deleted by $\text{LeftLinearWave}(i, j, \alpha)$.

Now consider the case $\gamma_i > \alpha$. Then the mega-segment between $q_i$ and $q_{i+1}$ is above the ray with slope $\alpha$ shot from $q_i$. By Lemma 11, all the (passive) points between $q_i$ and $q_{i+1}$ are also above this ray. Hence $q_{i+1}$ is deleted by the ray shooting process.

Finally, consider the case $\gamma_i = \alpha$. Then the mega-segment between $q_i$ and $q_{i+1}$ coincides with the ray with slope $\alpha$ shot from $q_i$. By Lemma 11, all the (passive) points between $q_i$ and $q_{i+1}$ will be deleted by the ray shooting process. Let $w'$ be such that $q_{i+1} = p_{w'}$. If $\alpha_{w'} \geq \alpha$ then $q_{i+1}$ will be deleted by the process. Otherwise, $\alpha_{w'} < \alpha$, so the ray shooting process terminates at $q_{i+1}$. Since all the passive points between $q_i$ and $q_{i+1}$ were deleted, $q_i$ and $q_{i+1}$ become consecutive in $P$, and the slope of the corresponding segment is $\gamma_i = \alpha$. But the slope of the segment starting at $q_{i+1}$ is $\alpha_{w'} < \alpha$, so $q_{i+1}$ becomes passive. ▶

We call rays with $\gamma_i > \alpha$ long rays, and those with $\gamma_i \leq \alpha$ short rays. Since long rays decrease the size of $P_{\text{active}}$ 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 $|P_{\text{active}}|$. The short rays, which do not decrease $|P_{\text{active}}|$, will be handled lazily. Namely, instead of explicitly shooting a short ray in the mega-segment starting at an active point $q_i$, we only store the slope of the ray and postpone its execution until it is required (e.g., by a $\text{Lookup}$ 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

In this section, we provide a technical overview of the construction of the range operation data structure of Definition 3. The complete implementation details and proofs appear in the full version of this paper ([9]). 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.  However, we shall maintain that the set of active points with respect to $\mathcal{P}$ and $\mathcal{P}_{\text{active}}$ is the same:

► Invariant 1. $\mathcal{P}_{\text{active}} = \mathcal{P}_{\text{active}}$.

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 AddConst, 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.  

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

► Lemma 13 (Add-min Data Structure). There exists a data structure supporting the following operations in $O(\text{polylog} n)$ time on a set of points $S$.

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

---

7 It would have been more accurate to use $\bar{\alpha}, \bar{\beta}$, and $\bar{\gamma}$, but this would be too cumbersome, so we stick to using $\alpha, \beta, \gamma$.
8 See Invariant 3 and the note following it.
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 \( \tilde{P} \) 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 \( \tilde{P} \) and in \( \tilde{P}_{active} \), 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 \( P \), then \( A[x] = \tilde{A}[x] \).

![Figure 5](image-url) 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 \( \tilde{P} \) (in grey).

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 \( P_{active} \) is itself. Since \( P_{active} = \tilde{P}_{active} \) we have that \( p \in \tilde{P}_{active} \) is the predecessor of \( x \) in \( P_{active} \) as well. By definition \( r_z \) goes through \( p = q_z \), so \( r_z(x) = A[x] \), 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, \( \tilde{P} = P = \{(1, 0), (|A|, 0)\} \), and \( \rho_1 = \rho_2 = \infty \) (We interpret a line with a slope of \( \infty \) as \( y = \infty \)). 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.

**The flush Operation.** We first describe a service operation \( \text{flush}(q_z) \) which explicitly shoots the pending short ray in the mega-segment starting at the active point \( q_z \). It will be useful to invoke \( \text{flush} \) before serving \( \text{Lookup} \) 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 $q_z = p_w \in \mathcal{P}_{active}$ is performed as follows. Starting from $p_{w+1}$, we scan the points in $\mathcal{P}$. When scanning $p = (x, y)$, we compare $y$ and $r_z$. If $r_z(x) \leq y$, we remove $p$ from $\mathcal{P}$. Otherwise, the scan halts. Let $p_{end}$ 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 $p_{del}$ be the last point deleted by the scan. We compute the intersection $p' = ([x^\star], r_z([x^\star]))$ and $p'' = ([x^\star], \ell([x^\star]))$ to $\mathcal{P}$ (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 14. Applying flush to an active point $q_z \in \mathcal{P}_{active}$ preserves Invariants 1–3. Furthermore, it guarantees that the restriction of $\mathcal{P}$ and $\mathcal{F}$ to the (passive) points between $q_z$ and $q_{z+1}$ is identical, and that for every $x \in [x_z \ldots x_{z+1}]$, $A[x] = \hat{A}[x]$.

Proof. Invariant 2 is maintained because the flush operation sets $\rho_z$ to $\infty$. Since $\rho_z > \gamma_z$, it is guaranteed by Lemma 12 that the scan of flush ends at $q_{z+1}$ or before $q_{z+1}$. It follows that Invariant 1 is maintained because $\mathcal{P}_{active}$ does not change and flush only deletes passive points of $\mathcal{P}$. 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 $x \in [x_z \ldots x_{z+1}]$, so we need to show $A[x] = \hat{A}[x]$. Let $x \in [x_z \ldots x_{z+1}]$. If $x \leq x^\star$, then before flush was applied, we had $\hat{A}[x] \geq r_z(x)$, and therefore by Invariant 3 $A[x] = \min(\hat{A}[x], r_z(x)) = r_z(x)$. Since flush sets the value of $\hat{A}[x]$ to be $r_z(x)$ for $x < x^\star$, Invariant 3 still holds. If $x > x^\star$, the value of $\hat{A}[x]$ is not changed by flush. Since the line $\hat{\ell}$ between $p_{del}$ and $p_{end}$ starts not below the $r_z$ and ends below $r_z$, its slope is smaller than $\rho_z$. Since the points between $p_{end}$ and $q_z$ (excluding $q_z$) are passive, the slopes of the corresponding segments are also lower than $\rho_z$ and therefore $(x, \hat{A}[x])$ is below $r_z$ for every $x \in (x' \ldots x_{z+1})$. Due to Invariant 3 before the application of flush, we have $A[x] = \min(\hat{A}[x], r_z(x)) = \hat{A}[x]$. Therefore, assigning $\rho_z \leftarrow \infty$ and not changing $\hat{A}[x]$ satisfies Invariant 3.

4 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 $\text{DTW}_k(S, T) = \min(\text{DTW}(S, T), k+1)$ for a given integer $k$. In this section, we prove the following theorem:

Theorem 15. The Dynamic Time Warping distance of two run-length encoded strings $S$ and $T$ with $n$ and $m$ runs respectively can be computed in $O(nk)$ time if $\text{DTW}(S, T) \leq k$.

The key structural insight for Theorem 15 is that there is a set of $O(nk)$ blocks containing all the vertices $(x, y)$ with $\text{dist}(x, y) \leq k$. 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 16. 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 $\text{dist}(x, y) > k$. 

Proof. We assume without loss of generality that \( j - i > 2k \). Let \( p \) be a path from \((0,0)\) to \((x,y)\). Let \( P = B_{i_1,j_1}, B_{i_2,j_2} \ldots B_{i_{|P|},j_{|P|}} \) 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 \( a \in [1 \ldots |P| - 1] \) 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 \( |P| - i_P > 2k \), and \( i_1 - j_1 = 0 \), there must be at least \( 2k + 1 \) values of \( a \in [1 \ldots |P| - 1] \) 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 \( e_{B_{ia},j_a} \geq 1 \) or \( e_{B_{ia},j_a+1} \geq 1 \). Let \( B' \in \{B_{ia,j_a}, B_{ia,j_a+1}\} \) 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 16 that if \( \text{dist}(x,y) \leq k \) 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. 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 \text{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 \( \text{DTW}_k(S,T) = \min(\text{dist}(N,M), k + 1) \).

References

30:16 Öptimal Dynamic Time Warping on Run-Length Encoded Strings

16 Pawel Gawrychowski and Yanir Edri. private communication, 2016.


