Approximating Dynamic Time Warping Distance Between Run-Length Encoded Strings

Zoe Xi
Massachusetts Institute of Technology, Cambridge, MA, USA

William Kuszmaul
Massachusetts Institute of Technology, Cambridge, MA, USA

Abstract

Dynamic Time Warping (DTW) is a widely used similarity measure for comparing strings that encode time series data, with applications to areas including bioinformatics, signature verification, and speech recognition. The standard dynamic-programming algorithm for DTW takes $O(n^2)$ time, and there are conditional lower bounds showing that no algorithm can do substantially better.

In many applications, however, the strings x and y may contain long runs of repeated letters, meaning that they can be compressed using run-length encoding. A natural question is whether the DTW-distance between these compressed strings can be computed efficiently in terms of the lengths k and ℓ of the compressed strings. Recent work has shown how to achieve $O(k\ell^2 + \ell k^2)$ time, leaving open the question of whether a near-quadratic $\tilde{O}(k\ell)$-time algorithm might exist.

We show that, if a small approximation loss is permitted, then a near-quadratic time algorithm is indeed possible: our algorithm computes a $(1+\epsilon)$-approximation for $DTW(x, y)$ in $\tilde{O}(k\ell/\epsilon^3)$ time, where k and ℓ are the number of runs in x and y. Our algorithm allows for DTW to be computed over any metric space (Σ, δ) in which distances are $\mathrm{O}(\log n)$-bit integers. Surprisingly, the algorithm also works even if δ does not induce a metric space on Σ (e.g., δ need not satisfy the triangle inequality).

2012 ACM Subject Classification Theory of computation → Approximation algorithms analysis

Keywords and phrases Dynamic time warping distance, approximation algorithms, run-length encodings, computational geometry

Digital Object Identifier 10.4230/LIPIcs.ESA.2022.90

Related Version *Full Version*: `http://arxiv.org/abs/2207.00915`

Funding This research was funded by a Hertz Foundation Fellowship and an NSF GRFP Fellowhship. The research was also partially sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

Acknowledgements The authors would like to thank Charles E. Leiserson for his helpful feedback and suggestions.

1 Introduction

Dynamic Time Warping (DTW) distance is a well-known similarity measure for comparing strings that represent time-series data. DTW distance was first introduced by Vintsyuk in 1968 [40], who applied it to the problem of speech discrimination. In the decades since, DTW has become one of the most widely used similarity heuristics for comparing time series [28] in applications such as bioinformatics, signature verification, and speech recognition [20, 34, 33, 1, 14, 43].

Consider any two strings x and y, with characters taken from some metric space (Σ, δ). For example, in many applications, we have that $\Sigma = \mathbb{R}^c$ for some parameter c and that $\delta(a, b) = \|a - b\|_2$ computes ℓ_2 distance. Define a ***time warp*** of x (and similarly of y) to be any string x' that can be obtained by ***warping*** the letters in x, where warping a letter means replacing it with ≥ 1 consecutive copies of itself. The DTW-distance $DTW(x, y)$ is defined to be

$$\min_{|x'|=|y'|} \sum_{i=1}^{|x'|} \delta(x'_i, y'_i),$$

where x' and y' range over all time warps of x and y.

The most fundamental question concerning DTW is how to compute it efficiently. Vintsyuk showed that, given strings x and y of length n, it is possible to compute $DTW(x, y)$ in $O(n^2)$ time [40]. His algorithm, which was one of the earliest uses of dynamic programming, continues to be taught in textbooks and algorithms courses today.

For many decades, it was an open question whether any algorithm could achieve a running time of $O(n^{2-\Omega(1)})$. (Interestingly, it is known that one *can* shave small sub-polynomial factors off of the running time [22].) A major breakthrough occurred in 2015, when Abboud, Backurs, and Williams [2] and Bringmann and Künnemann [12] established conditional lower bounds prohibiting any strongly subquadratic-time algorithm for DTW, unless the Strong Exponential Time Hypothesis (SETH) fails.

This lower bound puts us in an interesting situation. On one hand, the classic $O(n^2)$-time algorithm is often too slow for practical applications. On the other hand, we have good reason to believe that it is nearly optimal. This has led researchers to focus on forms of beyond-worst-case analysis when studying the DTW problem.

An especially appealing question [39, 21] is what happens if x and y both contain long runs of repeated letters. In this case, the strings can be compressed using run-length encoding (RLE). For example, the string "`aaaaabbc`" has RLE encoding "(`a`, 5), (`b`, 2), (`c`, 1)". If a string x has k runs, then it is said to have an RLE representation of length k.

It is known that, if x and y each contain k runs, then $DTW(x, y)$ can be computed in $O(k^3)$ time [21].[1] It is still an open question whether it is possible to significantly reduce this cubic running time, and in particular, whether a near-quadratic time algorithm might be possible.

This paper: A Near-Quadratic Approximation Algorithm

We show that, if a small approximation loss is permitted, then a near-quadratic time algorithm is indeed possible. Consider any two run-length encoded strings $x \in \Sigma^n$ and $y \in \Sigma^n$, where x has k runs and y has ℓ runs. Let δ be an arbitrary distance function $\delta : \Sigma \times \Sigma \to [\text{poly}(n)]$ mapping pairs of characters to $O(\log n)$-bit nonnegative integers. (Perhaps surprisingly, our algorithms will not require δ to satisfy the triangle inequality, or even to be symmetric.)

Our main result is an algorithm that computes a $(1 + \epsilon)$-approximation for $DTW(x, y)$ in $\tilde{O}(k\ell/\epsilon^3)$ time[2]. In the special case where Σ is over Hamming space (i.e., $\delta(a, b)$ is either 0 or 1 for all $a, b \in \Sigma$), the running time of our algorithm further improves to $\tilde{O}(k\ell/\epsilon^2)$.

[1] More generally, if x contains k runs and y contains ℓ runs, then the time becomes $O(k\ell^2 + \ell k^2)$.
[2] Here we are using soft-O notation to mean that $\tilde{O}(k\ell/\epsilon^3)$ is equivalent to $O((k\ell/\epsilon^3)\,\text{polylog}(n))$.

Our algorithm takes a classical geometric interpretation of DTW in terms of paths through a grid, and shows how to decompose each path in such a way that its components can be efficiently approximated. This allows for us to reduce the problem of approximating DTW-distance between RLE strings to the problem of computing pairwise distance in a small directed acyclic graph.

Other related work

In addition to work on run-length-encoded strings [39, 21], there has been a recent push to study other theoretical facets of the DTW problem. This includes work on approximation algorithms [25, 3, 42], low-distance-regime algorithms [25], communication complexity [11], slightly-subquadratic algorithms [22], reductions to other similarity measures [25, 37, 36], binary DTW [26, 38], etc.

All of these results (along with the results in this paper) can be viewed as part of a larger effort to close the gap between what is known about DTW and what is known about its closely related cousin ***edit distance***, which measures the number of insertions, deletions, and substitutions of characters needed to turn one string x into another string y. Like DTW, edit distance can be computed in $\mathrm{O}(n^2)$ time using dynamic programming [40, 35] (and can be computed in *slightly* subquadratic time using lookup-table techniques [31]). Also like DTW, edit distance has conditional lower bounds [12, 2, 25] prohibiting strongly subquadratic time algorithms.

When it comes to beyond-worst-case analysis, however, edit distance has yielded much stronger results than DTW: it is known how to compute a constant-approximation for edit distance in strongly subquadratic time [5, 24, 10, 15, 4, 9, 6, 17]; it is known how to compute the edit distance between RLE strings in $\tilde{O}(k\ell)$ time [19, 8, 18, 29, 13, 7, 32, 23, 30]; and if two strings x and y have small edit distance k, it is known how to compute the edit distance in $\mathrm{O}(|x| + |y| + k^2)$ time [16, 27].

Whether or not any of these results can be replicated for DTW remains the central open question in modern theoretical work on DTW. There are several reasons to believe that DTW computation should be more challenging than edit distance. Whereas edit distance satisfies the triangle inequality, DTW does not (for example, if we take $\Sigma = \{0, 1\}$, then $DTW(111110, 100000) = 0$, $DTW(100000, 000000) = 1$, and $DTW(111110, 000000) = 5$). This erratic behavior of DTW seems to make it especially difficult to approximate. Additionally, whereas almost all work on edit distance focuses on insertion/deletion/substitution costs of 1, work on DTW must consider arbitrary cost functions δ for comparing characters. Finally, it is known that the problem of computing edit distance actually *reduces* to that of computing DTW [25], indicating that the latter problem is at least as hard (although, interestingly, this reduction *does not* apply in the run-length encoded setting).

Our paper represents the first evidence that an $\tilde{O}(k\ell)$-time algorithm for DTW may be within reach. Such an algorithm would finally unify edit distance and DTW in the run-length-encoded setting.

2 Technical Overview

This section gives a technical overview of how we approximate DTW-distance between run-length encoded strings. To simplify exposition, we focus here only on the big ideas in the algorithm design and we defer the detailed analysis to later sections.

Throughout the section, we consider two strings x and y of length n whose characters are taken from a set Σ with a symmetric distance function $\delta : \Sigma \times \Sigma \to \mathbb{N} \cup \{0\}$. Our only assumption on δ is that $\delta(a, b) \in \{0, 1, 2, \ldots, \operatorname{poly}(n)\}$ for all $a, b \in \Sigma$. (We do not need the triangle inequality on δ.) Let k and ℓ be the number of runs in x and y, respectively. We will describe a $(1 + \mathrm{O}(\epsilon))$-approximate algorithm that takes $\tilde{O}(k\ell/\operatorname{poly}(\epsilon))$ time.

How to think about DTW

There are several mathematically equivalent ways (see, e.g., [25, 22, 21]) to define the dynamic time warping distance between x and y. In this paper, we work with the geometric interpretation: consider an $n \times n$ grid where cell (i, j) has ***cost*** $\delta(x_i, y_j)$; consider the ***paths*** through the grid that travel from $(1, 1)$ to (n, n) via steps of the form $\langle 1, 0\rangle$ (a horizontal step (h-step) to the right), $\langle 0, 1\rangle$ (a vertical step (v-step) up), and $\langle 1, 1\rangle$ (a diagonal step (d-step) to the upper right); the ***cost*** of such a path is the sum of the costs of the cells that it encounters, and $DTW(x, y)$ is defined to be the smallest cost of any such path. For an example, see Figure 1, which shows an optimal full path for computing $DTW(\texttt{aaabbbbddd}, \texttt{aabcdd}) = 1$, where the δ-function measures the distance between characters in the alphabet.

Note that the i-th column of the grid corresponds to x_i and the j-th row of the grid corresponds to y_j. Thus, each run $x_{i_0}, \ldots, x_{i_1}$ in x corresponds to a sequence of adjacent columns $i_0, \ldots, i_1$ in the grid, and each run $y_{j_0}, \ldots, y_{j_1}$ in y corresponds to a sequence of adjacent rows $j_0, \ldots, j_1$ in the grid.

If we want to design an algorithm that approximates $DTW(x, y)$ in $\tilde{O}(k\ell/\operatorname{poly}(\epsilon))$ time, then it is natural to think about the grid as follows. We break the grid into ***blocks*** by drawing a vertical line between every pair of runs in x and a horizontal line between every pair of runs in y; and label the blocks $\{\mathbf{B}_{i,j}\}_{i\in[k],j\in[\ell]}$, where block $\mathbf{B}_{i,j}$ corresponds horizontally to the i-th run in x and vertically to the j-th run in y. All of the cells within a given block B have the same cost, which we refer to as $\delta(B)$. We may also use $\delta_{i,j}$ for $\delta(\mathbf{B}_{i,j})$. We refer to the first/last row of each block as a lower/upper ***horizontal boundary*** and to the first/last column of each block as a left/right ***vertical boundary***.

Finally, it will be helpful to talk about sequences of blocks that are adjacent horizontally or vertically. An ***h-block segment*** consists of a sequence of consecutive blocks lined up horizontally. Formally, given $i_1 \le i_2$ in $[k]$ and j in $[\ell]$, we use $\mathbf{B}_{[i_1,i_2],j}$ for the h-block segment $\mathbf{B}_{i_1,j}, \mathbf{B}_{i_1+1,j}, \ldots, \mathbf{B}_{i_2,j}$. Similarly, a ***v-block segment*** consists of a sequence of consecutive blocks lined up vertically – we use $\mathbf{B}_{i,[j_1,j_2]}$ for the v-block segment $\mathbf{B}_{i,j_1}, \mathbf{B}_{i,j_1+1}, \ldots, \mathbf{B}_{i,j_2}$. In the same way that we can talk about the four boundaries of a block, we can talk about the four boundaries of a given h-block or v-block segment.

Intuitively, since there are $\mathrm{O}(k\ell)$ blocks, our goal is to design an algorithm that runs in time roughly proportional to the number of blocks.

How to think about the optimal path

Let P be a minimum-cost path through the grid. We can decompose the path into a sequence of disjoint ***components*** $P_1, P_2, P_3, \ldots$, where each component takes one of two forms:

1. A **horizontal-to-vertical** (h-to-v) component connects a cell on the lower boundary of some v-block segment to another cell on the right boundary of the same v-block segment.
2. A **vertical-to-horizontal** (v-to-h) component connects a cell on the left boundary of some h-block segment to another cell on the upper boundary of the same h-block segment.

The components $P_1, P_2, P_3, \ldots$ are defined such that the end cell of each P_r connects to the the start cell of each P_{r+1} via a single step (either horizontal, vertical, or diagonal).

We will now describe a series of simplifications that we can make to P while increasing its total cost by at most a $(1 + \mathrm{O}(\epsilon))$-factor. The simplifications are central to the design of our algorithm.

Simplification 1: Rounding each component to start and end on "snap points"

Let us call a grid cell (i, j) an ***intersection point*** if it lies in the intersection of a horizontal boundary and a vertical boundary. (Each block contains at most four intersection points.) We call a grid cell a ***snap point*** if either it is an intersection point, or it is of the form $(i+(1+\epsilon)^t, j)$ on the upper boundary of a block B, or it is of the form $(i+1+(1+\epsilon)^t, j)$ on the lower boundary of a block B, or it is of the form $(i, j+(1+\epsilon)^t)$ on the right boundary of a block B, or it is of the form $(i, j+1+(1+\epsilon)^t)$ on the left boundary of a block B, where (i, j) is an intersection point of the block B and t is nonnegative integer (since this is a technical overview, we ignore floor and ceiling issues). For each boundary cell p in the grid, define $\mathit{snap}(p)$ to be the nearest snap point to the right of p, if p is on a horizontal boundary, and to be the nearest snap point above p, if p is on a vertical boundary. If p is on both a horizontal boundary and a vertical boundary, then p is an intersection point, so $\mathit{snap}(p) = p$.

How much would the cost of P increase if we required each of its components to start and end on snap points? Suppose, in particular, that we replace each component P_r with a component P'_r whose start point p_r has been replaced with $\mathit{snap}(p_r)$ and whose end point q_r has been replaced with $\mathit{snap}(q_r)$. It may be that $\mathit{snap}(q_r)$ does not connect to $\mathit{snap}(p_{r+1})$, meaning that P'_r and P'_{r+1} do not connect properly. If this happens, however, then one can simply modify the starting-point of P'_{r+1} in order to connect it to P'_r (and it turns out this only makes P'_{r+1} cheaper).

Let P' be the concatenation of P'_1, P'_2, P'_3, …. To bound the cost of P', we can argue that the cost of each P'_r is at most $(1 + \epsilon)$ times that of P_r. To transform P_r into P'_r, the first step is to round the start point p_r of P_r to $\mathit{snap}(p_r)$ – one can readily see that this only decreases (or leaves unchanged) the cost of P_r. The second step is to round the end point q_r of P_r to $\mathit{snap}(q_r)$. For simplicity, assume that P_r is an h-to-v component that starts on the lower boundary of some block $\mathbf{B}_{i,j_1}$ and finishes on the right boundary of some block $\mathbf{B}_{i,j_2}$. Let (u, v) be the lower-right intersection point of $\mathbf{B}_{i,j_2}$ and suppose that P_r finishes in cell $(u, v + s)$. Then P_r incurs cost at least $(s + 1) \cdot \delta_{i,j_2}$ in block $\mathbf{B}_{i,j_2}$. Moreover, the snap point $\mathit{snap}(q_r) = \mathit{snap}(u, v + s)$ is guaranteed to be in the set $\{(u, v + s + t)\}_{t\in\{0,1,...,\epsilon\cdot s\}}$. Thus the cost of traveling from q_r to $\mathit{snap}(q_r)$ is at most $\epsilon \cdot s \cdot \delta_{i,j_2}$. So the cost of P'_r is at most $(1 + \epsilon)$ times that of P_r.

By analyzing each component in this way, we can argue that $\mathrm{cost}(P') \leq (1 + \epsilon)\,\mathrm{cost}(P)$. Throughout the rest of the section, we will assume that P has been replaced with P', meaning that each component starts and ends with a snap point.

Simplification 2: Understanding the structure of each component

Next we observe that each individual component can be assumed to have a relatively simple structure. For simplicity, let us focus on an h-to-v component P_r in a v-block segment $\mathbf{B}_{i,[j_1,j_2]}$. We may assume without loss of generality that all of P_r's h-steps occur together on the lower boundary of some block; and that all of P_r's v-steps occur at the end of P_r. In other words, P_r is of the form $D_1 \oplus H \oplus D_2 \oplus U$ where $\oplus$ is for path concatenation, D_1 consists of d-steps, H consists of h-steps (along a lower boundary), D_2 again consists of d-steps, and U consists of v-steps (along a right boundary).[3] (See Figure 2 where the path $p_1q_1q_2p_2p_3$ in solid lines is such an example.)

[3] Note that the components D_1, H, D_2, and U are each individually allowed to be length 0.

Combined, these assumptions make it so that P_r is fully determined by four quantities: (1) P_r's start point p_r, (2) the block $\mathbf{B}_{i,j}$ in which H occurs, (3) the length of H, and (4) the length of U.

Define $\overline{P_r}$ to be the prefix of P_r that terminates as soon as U hits its first snap point. (See Figure 2 where the path $p_1q_1q_2p_2p_2'$ is such a prefix of the path $p_1q_1q_2p_2p_3$.) We will see later that $\overline{P_r}$ is, in some sense, the "important" part of P_r to our algorithm. Observe that $\overline{P_r}$ is fully determined by just three quantities: (1) P_r's start point p_r, (2) the block $\mathbf{B}_{i,j}$ in which H occurs, and (3) the length of H.

Simplification 3: Reducing the number of options for each component

We will now argue that, if we fix the start point p_r, and we are willing to tolerate a $(1 + \mathrm{O}(\epsilon))$-factor approximation loss, then we only need to consider $\mathrm{poly}(\epsilon^{-1} \log n)$ options for $\overline{P_r}$.

We begin by considering block $\mathbf{B}_{i,j}$ in which H occurs. Let us define the sequence of blocks $B_0, B_1, B_2, \ldots$ so that $B_s = \mathbf{B}_{i,j+s}$ and define the sequence of costs $\delta_0, \delta_1, \delta_2, \ldots$ so that $\delta_s = \delta_{i,j+s}$. We say that a block B_s is ***extremal*** if $(1 + \epsilon)\delta_s \leq \delta_t$ for all $t < s$. If we are willing to tolerate a $(1+\mathrm{O}(\epsilon))$-factor increase in $\overline{P_r}$'s cost, then we can assume without loss of generality that H occurs in an extremal block. On the other hand, there are only $\mathrm{O}(\log_{1+\epsilon}(n))$ extremal blocks, so this means that we only need to consider $\mathrm{O}(\log_{1+\epsilon}(n)) \leq \mathrm{poly}(\epsilon^{-1} \log n)$ options for the starting point of H.

Next we consider the length of the horizontal sub-component H. If we are willing to tolerate a $(1 + \mathrm{O}(\epsilon))$-factor increase in $\overline{P_r}$'s cost, then we can round $|H|$, the length of H, up to be a power of $(1 + \epsilon)$ (or to be whatever length brings us to the next vertical boundary). Thus we only need to consider $\mathrm{O}(\log_{1+\epsilon}(n)) \leq \mathrm{poly}(\epsilon^{-1} \log n)$ options for $|H|$.

Together, the block $\mathbf{B}_{i,j}$ in which H occurs and the length of H fully determine $\overline{P_r}$. Thus, we have reached the following conclusion: if the start point p_r of the component $\overline{P_r}$ is known, then there are only $\mathrm{poly}(\epsilon^{-1} \log n)$ options that we must consider for what $\overline{P_r}$ could look like. Moreover, although we have considered only h-to-v components here, one can make a similar argument for v-to-h components.

Approximating DTW in $\tilde{O}(k\ell/\mathrm{poly}(\epsilon))$ time

We will now construct a weighted directed acyclic graph $G = \langle V, E \rangle$ that has two special vertices $\mathbf{v}_0$ and $\mathbf{v}_*$ and that satisfies the following properties:

- G has a total of $\tilde{O}(k\ell/\mathrm{poly}(\epsilon))$ vertices/edges, and
- the distance from $\mathbf{v}_0$ to $\mathbf{v}_*$ in G is a $(1 + \mathrm{O}(\epsilon))$-approximation for $DTW(x, y)$.

This reduces the problem of approximating $DTW(x, y)$ to the problem of computing a distance in a weighted directed acyclic graph. The latter problem, of course, can be solved in linear time with dynamic programming; thus the graph G give us a $\tilde{O}(k\ell/\mathrm{poly}(\epsilon))$-time $(1 + \mathrm{O}(\epsilon))$-approximation algorithm for DTW.

We construct G to capture the different ways in which path components P_r can connect together (assuming that the path components take the simplified forms described above). As the vertices $v \in V$ correspond to the snap points p in the grid, we can use a vertex to refer to its corresponding snap point and vice versa. We define $\mathbf{v}_0$ to be the cell $(1, 1)$ in the grid and $\mathbf{v}_*$ to be the cell (n, n). We add edges E as follows:

- We connect each snap point p on a horizontal (resp. vertical) boundary to the next snap point q to its right (resp. above it).
- We connect each snap point p on a right (resp. upper) boundary to any snap points q on the adjacent left (resp. lower) boundary that can be reached from p in a single step.
- Each snap point $p \in V$ has $\operatorname{poly}(\epsilon^{-1} \log n)$ out-edges corresponding to the $\operatorname{poly}(\epsilon^{-1} \log n)$ options for what a (truncated) component $\overline{P_r}$ starting at p could look like.[4]

Note that, although we only add edges for *truncated* path components $\overline{P_r}$ (rather than full components P_r), these edges can be combined with edges of the first type in order to obtain the full component. This is why we said earlier that the truncated component is the "important" part of the component.

The paths from $\mathbf{v}_0$ to $\mathbf{v}_*$ in G correspond to the ways in which we can concatenate path components together to get a full path through the grid; if we assign the appropriate weights to the edges, then the cost of a path through G corresponds to the cost of the same path through the grid. The distance from $\mathbf{v}_0$ to $\mathbf{v}_*$ is therefore a $(1 + \mathrm{O}(\epsilon))$-approximation for $DTW(x, y)$.

Finally, we must bound the size of G. Each block contains at most four intersection points; so there are $\mathrm{O}(k\ell)$ total intersection points. Each intersection point creates at most $\mathrm{O}(\log_{1+\epsilon}(n))$ snap points; so there are $\mathrm{O}(k\ell\epsilon^{-1} \log n)$ snap points (which are the vertices in V). Each snap point has an out-degree of at most $\operatorname{poly}(\epsilon^{-1} \log n)$. Hence we have:

$$|E| \le \mathrm{O}(k\ell\epsilon^{-1} \log n) \operatorname{poly}(\epsilon^{-1} \log n) = \tilde{O}(k\ell/\operatorname{poly}(\epsilon)).$$

We can therefore compute the distance from $\mathbf{v}_0$ to $\mathbf{v}_*$ in $\tilde{O}(k\ell/\operatorname{poly}(\epsilon))$ time, as desired.

Paper outline

For the sake of simplicity, there are a number of details that we chose to ignore in this section (such as a time-efficient construction of G and a careful proof that the modifications to P incur only a $(1 + \mathrm{O}(\epsilon))$-factor change in its cost). In the remainder of the paper, we give a formal presentation and analysis of the algorithm outlined above.

3 Preliminaries

We use $[n_1, n_2]$ for the set $\{n_1, n_1 + 1, \ldots, n_2\}$ consisting of all the integers between n_1 and n_2, inclusive, and use $[n]$ as a shorthand for $[1, n]$. We use $T_{m,n}$ for a table consisting of m columns and n rows and $T_{m,n}[i, j]$ for the entry on the i-th column and j-th row, where $(i, j) \in [m] \times [n]$ is assumed. We may use T for $T_{m,n}$ if m and n can be readily inferred from the context. Please note that an entry $T_{m,n}[i, j]$ in a table should be distinguished from the value stored in the entry – when discussing the value, we shall refer to it as the content of the entry $T_{m,n}[i, j]$.

Letters

Let us assume an alphabet Σ, which is possibly infinite. We use δ for a distance function on letters such that $\delta(a, a) = 0$ for any $a \in \Sigma$. We do not require that δ be symmetric or the triangular inequality $\delta(a, c) \le \delta(a, b) + \delta(b, c)$ hold for δ.

[4] Note that G is not necessarily simple. If there are multiple ways that a component $\overline{P_r}$ could connect two vertices p_1 and p_2, then there will be multiple edges from p_1 to p_2.

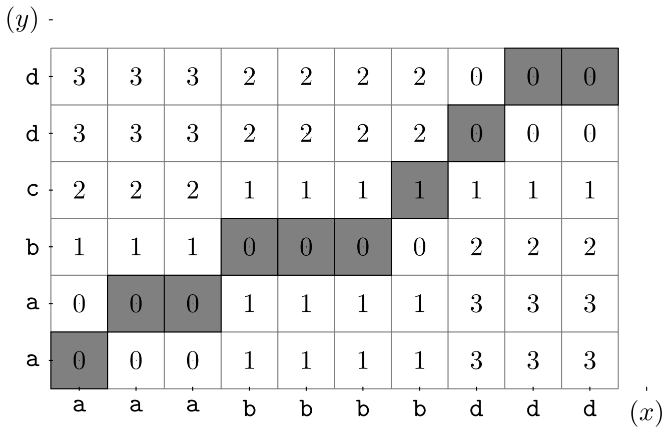

Figure 1 An optimal full path of the order $(10, 6)$ whose cost equals 1.

Strings

We use x and y for strings. We write $x = (a_1, \ldots, a_m)$ for a string consisting of m letters such that $x[i]$ (often written as x_i), the i-th letter in x, is a_i for each $i \in [m]$. We use $a\hat{}n$ for a string of n occurrences of a, which is also referred to as a *run* of a, and $\hat{x}$ for a run-length encoded (RLE) string, which consists of a sequence of runs. We use $|\hat{x}|$ and $\|\hat{x}\|$ for the length and r-length of $\hat{x}$, which are $m_1 + \cdots + m_k$ and k, respectively, in the case $\hat{x} = (a'_1\hat{}m_1, \ldots, a'_k\hat{}m_k)$.

A run in a string x is maximal if it is not contained in a longer run in x. There is a unique run-length encoding $\hat{x}$ of x that consists of only maximal runs in x, and this encoding $\hat{x}$ is referred to as the RLE representation of x. We also use $\|x\|$ for the number of maximal runs in x (and thus $\|x\| = \|\hat{x}\|$).

We use p for points, which are just integer pairs.

▶ **Definition 1.** *Given a point $p_1 = (i_1, j_1)$, another point $p_2 = (i_2, j_2)$ is a successor of p_1 if (1) $i_2 = i_1 + 1$ and $j_2 = j_1$, or (2) $i_2 = i_1$ and $j_2 = j_1 + 1$, or (3) $i_2 = i_1 + 1$ and $j_2 = j_1 + 1$.*

▶ **Definition 2.** *Let $P = \langle p_1, \ldots, p_R \rangle$ be a sequence of points such that $p_r \in [m] \times [n]$ holds for each $1 \leq r \leq R$. We call P a path of order (m, n) if p_{r+1} is a successor of p_r for each $1 \leq r < R$ (and this P is sometimes also called a "warping path" [21]). Also, we refer to a path of length 2 as a step that connects a point to one of its successors.*

We use $\mathcal{P}(m, n)$ for the set of paths of order (m, n). A path $P_1 \in \mathcal{P}(m, n)$ is a subpath of another path $P_2 \in \mathcal{P}(m, n)$ if P_1 is contained in P_2 (as a consecutive segment).

▶ **Definition 3.** *Let P_1 and P_2 be two non-empty paths. We use $P_1 \simeq P_2$ to mean that P_1 and P_2 begin at the same point and end at the same point.*

▶ **Definition 4.** *Let P_1 and P_2 be two paths such that the first point of P_2, if it exists, is the successor of the last point of P_1, if it exists. We write $P_1 + P_2$ to mean the concatenation of P_1 and P_2 (as sequences of points) that forms a path containing both P_1 and P_2 as its subpaths. In the case where both P_1 and P_2 are non-empty, there is a step in $P_1 + P_2$ connecting P_1 and P_2 that consists of the last point in P_1 and the first point in P_2.*

Also, we write $P_1 \oplus P_2$ to mean $P_1 + P'_2$ where the last point of P_1 is assumed to be the first point of P_2 and P'_2 is the tail of P_2, that is, P'_2 is obtained from removing the first point in P_2. In other words, $P_1 + P_2$ implies that P_1 and P_2 share no point while $P_1 \oplus P_2$ implies that P_1 and P_2 share one point, which is the last point of P_1 and the first point of P_2.

▶ **Definition 5.** *Let $x = (a_1, \ldots, a_m)$ and $y = (b_1, \ldots, b_n)$ be two strings. For each path $P \in \mathcal{P}(m, n)$, there is a value $cost_{x,y}(P) = \Sigma_{r=1}^{R} \delta(a_{i_r}, b_{j_r})$, where P equals $((i_1, j_1), \ldots, (i_R, j_R))$. This value is often referred to as the cost of P. We may write $cost(P)$ for $cost_{x,y}(P)$ if it is clear from the context what x and y should be.*

We call each $P \in \mathcal{P}(m, n)$ a full path if $(i_1, j_1) = (1, 1)$ and $(i_R, j_R) = (m, n)$. The DTW distance between x and y, denoted by $DTW(x, y)$, is formally defined as the minimum of $cost_{x,y}(P)$, where P ranges over the set of full paths of order (m, n). Also, a full path P is referred to as an optimal full path if $cost_{x,y}(P) = DTW(x, y)$. Given there are only finitely many paths of order (m, n), there must exist one full path that is optimal. As an example, the shaded squares in Figure 1 illustrate the following full path of the order $(10, 6)$:

$$\langle(1, 1), (2, 2), (3, 2), (4, 3), (5, 3), (6, 3), (7, 4), (8, 5), (9, 6), (10, 6)\rangle$$

where the number in each square is the assumed distance between the two corresponding letters (computed here as the difference between their positions in the alphabet).

▶ **Definition 6.** *Let $P = \langle(i_1, j_1), \ldots, (i_R, j_R)\rangle$.*

1. *P is a v-path if all the i_r are the same for $1 \le r \le R$.*
2. *P is a h-path if all the j_r are the same for $1 \le r \le R$.*
3. *P is a d-path if $i_{r+1} = i_r + 1$ and $j_{r+1} = j_r + 1$ for $1 \le r < R$.*

Please recall that a step is a path of length 2. If a step is a h-path/v-path/d-path, respectively, then it is a h-step/v-step/d-step, respectively.

▶ **Definition 7.** *Let $x = (a_1, \ldots, a_m)$ and $y = (b_1, \ldots, b_n)$ be two strings. We use $T_{DTW}(x, y)$ for the table $T_{m,n}$ such that the content of $T_{m,n}[i, j]$ is $\delta(a_i, b_j)$ for each $i \in [m]$ and $j \in [n]$.*

We may use T_{DTW} for $T_{DTW}(x, y)$ if x and y can be readily inferred from the context. If a table T_{DTW} can be readily inferred from the context, we often associate a point (i, j) with the entry $T_{DTW}[i, j]$ and think of a path $P = \langle(i_1, j_1), \ldots, (i_R, j_R)\rangle$ as the sequence of entries $T_{DTW}[i_r, j_r]$ for $1 \le r \le R$. As an example, a full path of the order $(10, 6)$ is given in Figure 1, where the path is indicated with the 10 shaded entries.

Suppose that the ith run (jth) in x (y) consists of the letters in x (y) from position i_1 (j_1) to position i_2 (j_2), inclusive. Then there is a corresponding block $\mathbf{B}_{i,j}$ consisting of all the entries $T_{DTW}[u, v]$ for $i_1 \le u \le i_2$ and $j_1 \le v \le j_2$. Finally, we introduce notation for discussing specific blocks:

▶ **Definition 8.** *If there exists a block B to the right of $\mathbf{B}_{i,j}$ such that $\delta(B) < \delta(\mathbf{B}_{i,j})$, we use $\beta_h(\mathbf{B}_{i,j})$ for such a B that is the closest to $\mathbf{B}_{i,j}$. In other words, $\beta_h(\mathbf{B}_{i,j})$ is $\mathbf{B}_{i',j}$ for the least i' satisfying $i < i'$ and $\delta(\mathbf{B}_{i',j}) < \delta(\mathbf{B}_{i,j})$. Similarly, if there exists a block B above $\mathbf{B}_{i,j}$ such that $\delta(B) < \delta(\mathbf{B}_{i,j})$, then $\beta_v(\mathbf{B}_{i,j})$ is $\mathbf{B}_{i,j'}$ for the least j' satisfying $j < j'$ and $\delta(\mathbf{B}_{i,j'}) < \delta(\mathbf{B}_{i,j})$.*

3.1 Computing DTW Distance with Graphs

It is well known [40] that one can turn the problem of computing $DTW(x, y)$ for two given strings x and y into a problem of finding the shortest distance between two given vertices in some graph, as follows.

Let $x = (a_1, \ldots, a_m)$ and $y = (b_1, \ldots, b_n)$. We can construct a directed graph $G_0 = \langle V_0, E_0\rangle$ such that

1. there is a vertex $\mathbf{v}_{i,j} \in V_0$ for each pair $(i, j) \in [m] \times [n]$, and
2. there is a directed edge $\mathbf{e}(\mathbf{v}_{i_1,j_1}, \mathbf{v}_{i_2,j_2})$ of length $\delta(a_{i_1}, b_{j_1})$ connecting $\mathbf{v}_{i_1,j_1}$ to $\mathbf{v}_{i_2,j_2}$ whenever (i_2, j_2) is a successor of (i_1, j_1).

We use $G_{DTW}(x, y)$ for this graph G_0 and use v and e to range over V_0 and E_0, respectively. We may also refer to each vertex $\mathbf{v}_{i,j} \in V_0$ simply as point (i, j) if there is no risk of confusion. Clearly, $|V_0|$, the size of V_0, is mn, and $|E_0|$, the size of E_0, is bounded by $3mn$ (since each point can have at most 3 successors).

As every warping path is naturally mapped to a path in the graph $G_{DTW}(x, y)$ and vice versa, we can use P to range over both warping paths in $T_{DTW}(x, y)$ and paths in $G_{DTW}(x, y)$ without risking confusion. Given a (non-empty) warping path P in $T_{DTW}(x, y)$, we use $len(P)$ for the length of the corresponding path of P in $G_{DTW}(x, y)$, which equals the cost of P minus the cost associated with the last point in P. Therefore, finding the value of $DTW(x, y)$ is equivalent to finding the shortest distance from $\mathbf{v}_{1,1}$ to $\mathbf{v}_{m,n}$, which can be done by running some version of Dijkstra's shortest distance algorithm. Alternatively, since $G_{DTW}(x, y)$ is acyclic, one can use dynamic programming to find the shortest distance, in which case the running time becomes $O(mn)$. This yields the classic dynamic-programming solution for computing $DTW(x, y)$ [40].

The basic strategy that we use in this paper to design a DTW approximation algorithm can be outlined as follows. Let $G_0 = \langle V_0, E_0 \rangle$ be the graph $G_{DTW}(x, y)$ given above. We try to construct a graph $G = \langle V, E \rangle$ such that $V \subseteq V_0$ holds and the length of each edge e in E that connects a vertex v_1 to another vertex v_2 equals the shortest distance from v_1 to v_2 as is defined in G_0. Let $dist_0$ and $dist$ be the shortest distance functions on the graphs G_0 and G, respectively. We attempt to prove that

$$dist_0(\mathbf{v}_{1,1}, \mathbf{v}_{m,n}) \leq dist(\mathbf{v}_{1,1}, \mathbf{v}_{m,n}) \leq \alpha \cdot dist_0(\mathbf{v}_{1,1}, \mathbf{v}_{m,n})$$

for some approximation ratio $\alpha > 1$ (e.g., $\alpha = 1 + \epsilon$ for $\epsilon > 0$). By running a shortest-path algorithm on G, we are able to compute $dist(\mathbf{v}_{1,1}, \mathbf{v}_{m,n})$ and thus obtain an α-approximation algorithm for $DTW(x, y)$. As the time complexity of such an algorithm can be bounded by $O(|E|)$ plus the time needed for constructing G, the key to finding a fast algorithm is try to minimize $|E|$, the size of E (while ensuring that the construction of G can be done in $O(|E|)$ time).

4 A (1+ϵ)-Approximation Algorithm for DTW

In this section, we present and analyze a $(1 + \epsilon)$-approximation algorithm for approximating the DTW distance between two run-length encoded strings in near-quadratic time.

Let $x = (a_1, \ldots, a_m)$ and $y = (b_1, \ldots, b_n)$ be two non-empty strings. Following Section 3.1, our approach will be to construct a graph $G = \langle V, E \rangle$ based on $G_{DTW}(x, y)$, reducing the problem of computing a $(1 + \epsilon)$-approximation of $DTW(x, y)$ to finding the shortest distance between the vertex $\mathbf{v}_{1,1}$ and the vertex $\mathbf{v}_{m,n}$ in G.

We begin by describing the notions of h-to-v paths and v-to-h paths. The role that these will play in our algorithm is that we will show how to decompose any full path P into a concatenation $P_1 + P_2 + \cdots$ of h-to-v and v-to-h paths.

▶ **Definition 9.** *Let x and y be two non-empty strings. A horizontal-to-vertical (h-to-v) path in $T_{DTW}(x, y)$ is a path that connects a point on the lower boundary of a block B_{i,j_1} to another point on the right boundary of B_{i,j_2}. An h-to-v component in a full path is a maximal h-to-v path that is not contained in any longer h-to-v path in the same full path. A vertical-to-horizontal (v-to-h) path can be defined similarly.*

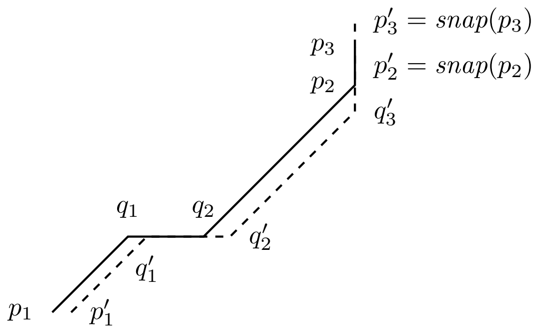

Figure 2 For illustrating h-to-v path approximation.

Note that an h-to-v path matches characters from a *single run* in x to characters from (possibly multiple) runs in y. We now make the (standard) observation that, when we are comparing a single run to characters to a multi-run string, DTW behaves in a very natural way:

▶ **Observation 10.** *Let $x = (a_1, \ldots, a_m)$ and $y = (b_1, \ldots, b_n)$ be two non-empty strings. Assume that x is a run of some letter a_0, that is, $a_0 = a_i$ for $1 \leq i \leq m$.*

1. *If $m \leq n$, then we have: $DTW(x, y) = \Sigma_{j=1}^{n} \delta(a_0, b_j)$. This case corresponds to a path of the form $D \oplus U$, where D consists of only d-steps and U only v-steps.*
2. *If $m \geq n$, then we have: $DTW(x, y) = \Sigma_{j=1}^{n} \delta(a_0, b_j) + (m - n) \cdot \delta(a_0, b_0)$, where b_0 is some b_j closest to a_0, that is, $\delta(a_0, b_0)$ equals the minimum of $\delta(a_0, b_j)$ for $1 \leq j \leq n$. This case corresponds to a path of the form $D_1 \oplus H \oplus D_2$, where D_1 and D_2 consist of only d-steps and H only h-steps. It should be further noted that, in this case, H can be assumed to travel along the lower boundary of some block, without loss of generality.*

We say that a path connecting p and q is optimal if its cost is the least among all the paths connecting p and q. By merging the two cases in Observation 10, we can assume that each optimal h-to-v path is of the form $D_1 \oplus H \oplus D_2 \oplus U$, where any of the four sub-components can vanish. From now on, we can use a 5-tuple $(p_1, q_1, q_2, p_2, p_3)$ (which may also be written as $p_1q_1q_2p_2p_3$) to refer to an h-to-v path, where p_1q_1 is D_1, q_1q_2 is H, q_2p_2 is D_2, and p_2p_3 is U. Similarly, each optimal v-to-h path is of the form $D_1 \oplus U \oplus D_2 \oplus H$, and we use a corresponding 5-tuple representation to refer to a v-to-h path as well.

Next we argue that any full path P_0 can be decomposed into h-to-v and v-to-h paths.

▶ **Lemma 11.** *Let x and y be two non-empty strings. Given a full path P_0 in $T_{DTW}(x, y)$, we have $P_0 = P_1 + \cdots + P_R$ where P_r is an h-to-v path for each odd $1 \leq r \leq R$ and P_r is a v-to-h path for each even $1 \leq r \leq R$.*

Proof. The proof follows directly from the defintions of h-to-v paths and v-to-h paths. For brevity, we defer the full proof to the extended version of the paper [41]. ◀

Given two non-empty strings $x = (a_1, \ldots, a_m)$ and $y = (b_1, \ldots, b_n)$, we outline as follows a strategy for approximating $DTW(x, y)$. Let P_0 be an optimal full path on $T_{DTW}(x, y)$ such that $cost(P_0) = DTW(x, y)$. By Lemma 11, we have $P_0 = P_1 + \cdots + P_R$, where P_1 is an h-to-v path and $P_1, \ldots, P_R$ are a sequence of alternating h-to-v paths and v-to-h paths. Let us choose an h-to-v path P_r for some $1 \leq r \leq R$. By Observation 10, we can assume that P_r is of the form of solid lines depicted in Figure 2.[5]

[5] The meaning of the dashed lines in the figure is to be explained later.

In more detail, the path P_r moves diagonally from a point p_1 on the lower boundary of a block B_1 until it meets the lower boundary of another block; it moves horizontally along that lower boundary for some distance; it then moves diagonally to reach a point p_2 on the right boundary of another block B_2 (which is either B_1 or sits above B_1); and finally it moves vertically to reach a point p_3 on the right boundary of another block B_3 (which is either B_2 or sits above B_2). Note that the horizontal moves contained in P_r must be inside a block where those moves cost the least.[6]

Let $G_0 = \langle V_0, E_0 \rangle$ be the graph $G_{DTW}(x, y)$ described in Section 3.1 for computing $DTW(x, y)$. We may use a point (that is, an integer pair) to refer to the corresponding vertex in V_0. We may also use a path P in $T_{DTW}(x, y)$ to denote its counterpart in G_0.

Given $\epsilon > 0$, we will construct a graph $G = \langle V, E \rangle$ such that:

- The point $(1, 1)$ is in V and $V \subseteq V_0$ holds.
- Every point in V is on a boundary. Each point in V that is on either a right or upper boundary is connected by an edge to the next snap point on the same boundary (if there is one).
- If there is a step (either an h-step, v-step, or d-step) connecting two boundary points p_1 and p_2 in E_0, then there is also a step connecting $snap(p_1)$ and $snap(p_2)$ in E, where $snap(p_1)$ (resp. $snap(p_2)$) is the nearest point in V above or to the right of p_1 (resp. p_2) on the same boundary as p_1.
- For each h-to-v (resp. v-to-h) path P from p_1 to p_2 (depicted by some solid lines in Figure 2) in G_0 and any point p'_1 in G to the right of p_1 (resp. above p_1) such that p_1 and p'_1 are on the same block boundary, there exists a path P' (depicted by some dashed lines in Figure 2) in G connecting p'_1 and the point $p'_2 = snap(p_2)$ in G such that $dist(P') \leq (1 + \epsilon) dist_0(P)$, where $dist$ and $dist_0$ are the shortest distance functions on the graphs G and G_0, respectively.

We remark that our construction of G will repeatedly make use of the following basic fact.

▶ **Observation 12.** *Let $\Delta(t) = \lfloor (1 + \epsilon)^t \rfloor$ for integers $t \geq 0$. For each integer $d \geq 1$, we have a $(1 + \epsilon)$-approximation of d that is of the form $\Delta(t)$. In other words, $d \leq \Delta(t) \leq (1 + \epsilon) \cdot d$ holds for some t.*

We now describe how to construct the graph G. We construct the set V of vertices as follows:

1. Each vertex in G_0 corresponding to a corner point in $T_{DTW}(x, y)$ should be added into V. There are at most $4k\ell$ such vertices, where $k = \|x\|$ and $\ell = \|y\|$.
2. Assume (i, j) is the lower-left corner of block B.
   - If a point $(i + 1 + \Delta(t), j)$ is on the lower boundary of B for some $t \geq 0$, then this point should be added into V. There are at most $\log_{1+\epsilon}(m)$ such points for the block B.
   - If a point $(i, j + 1 + \Delta(t))$ is on the left boundary of B for some $t \geq 0$, then this point should be added into V. There are at most $\log_{1+\epsilon}(n)$ such points for the block B.
3. Assume (i, j) is the upper-left corner of block B. If a point $(i + \Delta(t), j)$ is on the upper boundary of B for some $t \geq 0$, then this point should be added into V. There are at most $\log_{1+\epsilon}(m)$ such points for the block B.
4. Assume (i, j) is the lower-right corner of block B. If a point $(i, j + \Delta(t))$ is on the right boundary of B for some $t \geq 0$, then this point should be added into V. There are at most $\log_{1+\epsilon}(n)$ such points for the block B.

[6] If there are several blocks in which such horizontal moves can take place, we simply assume that the moves are inside the lowest of these blocks.

The points in V are referred to as ***snap points***. Given a snap point p' on the upper or right boundary of some block, if $p'q'$ is a d-step for some point q', then q' is also a snap point. This can be readily verified by inspecting the construction of V. Also, for each block B, there are at most $\mathrm{O}(\log(m+n)/\epsilon)$ points added to V. Therefore, $|V|$, the size of V, is $\mathrm{O}(k\ell \cdot \log(m+n)/\epsilon)$ or simply $\tilde{\mathrm{O}}(k\ell/\epsilon)$.

▶ **Definition 13.** *Given a point $p \in V_0$ on a horizontal boundary of a block B, we use $snap_h(p)$ for the point $p' \in V$ such that p' is p if $p \in V$ or p' is the closest point to the right of p that is on the same boundary of B. The existence of such a point is guaranteed as all of the corner points are included in V. Similarly, $snap_v(p)$ can be defined for each point p on a vertical boundary of a block.*

We can use $snap(p)$ for either $snap_h(p)$ or $snap_v(p)$ without confusion: If both $snap_h(p)$ and $snap_v(p)$ are defined for p, then p must be a corner of some block B, implying $p = snap_h(p) = snap_v(p)$ since $p \in V$ holds. We argue as follows that finding $snap(p)$ for each given p can be done in $\tilde{\mathrm{O}}(1)$ time.[7]

▶ **Definition 14.** *Let $x = (a_1, \ldots, a_m)$ be a string and $\hat{x} = (a'_1\hat{\ }m_1, \ldots, a'_k\hat{\ }m_k)$ be its RLE representation. Let M_r be $m_1 + \cdots + m_r$ for each $0 \le r < k$. For each $1 \le i \le m$, we use $\hat{i}_x$ for the pair (i_0, i_1) such that $i = M_{i_0} + i_1$ for $1 \le i_1 \le m_{i_0+1}$.*

We may use $\hat{i}$ for $\hat{i}_x$ if x can be readily inferred from the context.

It is worth taking a moment to verify that we can compute $\hat{i}_x$ efficiently. Assume that an array storing M_r for $0 \le r < k$ is already built (in $\mathrm{O}(k)$ time). Given $i \in [m]$, we can perform binary search on the array to find i_0 in $\mathrm{O}(\log(k))$ time such that $M_{i_0} < i \le M_{i_0+1}$; we can then compute $\hat{i}_x$ as $(i_0, i - M_{i_0})$.

It is also worth verifying that we can compute $snap(p)$ in $\tilde{\mathrm{O}}(1)$ time. Given a point $p = (i, j)$ on a boundary of some block B in $T_{DTW}(x, y)$, we can compute $\hat{i}_x = (i_0, i_1)$ in $\mathrm{O}(\log(k))$ time. Similarly, we can compute $\hat{j}_y = (j_0, j_1)$ in $\mathrm{O}(\log(\ell))$ time. We can locate the block B as $\mathbf{B}_{i_0+1, j_0+1}$, and then find $snap(p)$ in $\mathrm{O}(1)$ time (assuming $\log_{1+\epsilon}(i_1)$ and $\log_{1+\epsilon}(j_1)$ can be computed in $\mathrm{O}(1)$ time). Therefore, given p, we can compute $snap(p)$ in $\tilde{\mathrm{O}}(1)$ time.

Having established that we can compute $\hat{i}$ and $snap(p)$ efficiently, we are nearly ready to describe the construction of the edges E. Our final task before doing so is to establish a bit more notation for how to talk about blocks.

Please recall that $\beta_h(\mathbf{B}_{i,j})$ (resp. $\beta_v(\mathbf{B}_{i,j})$ refers to the closest block $\mathbf{B}_{i',j}$ (resp. $\mathbf{B}_{i,j'}$) such that $\delta(\mathbf{B}_{i',j}) < \delta(\mathbf{B}_{i,j})$ (resp. $\delta(\mathbf{B}_{i,j'}) < \delta(\mathbf{B}_{i,j})$) holds. If there is no such a block, $\beta_h(\mathbf{B}_{i,j})$ (resp. $\beta_h(\mathbf{B}_{i,j})$) is undefined.

▶ **Definition 15.** *We refer to $\mathbf{B}_{i_1,j}, \cdots, \mathbf{B}_{i_S,j}$ as a β_h-sequence if $\mathbf{B}_{i_{s+1},j} = \beta_h(\mathbf{B}_{i_s,j})$ for $1 \le s < S$. Let $\beta^*_h(x, y)$ be the length of a longest β_h-sequence. Clearly, we have $\beta^*_h(x, y) \le \|x\|$. Similarly, we refer to $\mathbf{B}_{i,j_1}, \cdots, \mathbf{B}_{i,j_S}$ as a β_v-sequence if $\mathbf{B}_{i,j_{s+1}} = \beta_v(\mathbf{B}_{i,j_s})$ for $1 \le s < S$. Let $\beta^*_v(x, y)$ be the length of a longest β_v-sequence. Clearly, we have $\beta^*_v(x, y) \le \|y\|$. Let $\beta^*(x, y) = \max(\beta^*_h(x, y), \beta^*_v(x, y))$.*

Observe that if the underlying distance function δ on letters is from Hamming space, then $\beta^*(x, y) \le 2$ for any x and y. Slightly more generally, if δ is bounded by a constant, then $\beta^*(x, y)$ is bounded by the same constant plus one. Later in the section, in the proof of Theorem 21, we will also see an important (and much more general) case where $\beta^*(x, y)$ is guaranteed to be $\tilde{\mathrm{O}}(1)$.

[7] We slightly abuse the $\tilde{\mathrm{O}}$ notation here as the parameters m and n for the implicit log-terms are not explicitly mentioned.

We are now ready to explain the construction of the set E of edges for connecting vertices in V. The basic idea is to construct E in such a way that, for any constructed path $p'_1q'_1q'_2q'_3p'_2$ (as is depicted in Figure 2), there should be a path in G going from p'_1 to p'_2 whose cost is at most the cost of the path in G_0 – this allows for the graph G to capture all such paths, and ultimately allows for G to be used in our approximation algorithm. Formally, the construction of E can be performed with the following steps:

1. Note that the corner points in V_0 are all in V. The edges connecting these corner points in E_0 should be added into E.
2. Given a point $p'_1 \in V$ on the upper boundary of a block B, if there is a d-step from p'_1 to p'_2 (on the lower boundary of the block above B), then p'_2 is in V and an edge from p'_1 to p'_2 should be added into E whose length equals $\delta(B)$.
3. Given a point $p'_1 \in V$ on the right boundary of a block B, if there is a d-step from p'_1 to p'_2 (on the left boundary of the block to the right of B), then p'_2 is in V and an edge from p'_1 to p'_2 should be added into E whose length equals $\delta(B)$.
4. If two points $p'_1 = (i_1, j_1)$ and $p'_2 = (i_2, j_2)$ in V are on the same horizontal or vertical boundary of a block B such that p'_2 is the closest point above or to the right of p'_1, then an edge from p'_1 to p'_2 should be added into E whose length equals $(i_2 - i_1) \cdot \delta(B)$ (horizontal) or $(j_2 - j_1) \cdot \delta(B)$ (vertical).
5. Let p'_1 be a point in V on the lower boundary of a block B_1. This step adds into E edges between p'_1 and certain chosen snap points q'_4 such that there are h-to-v paths connecting p'_1 and q'_4.
   Let us use $B_1^1, \ldots, B_1^S$ for the sequence where $B_1 = B_1^1$ and $B_1^{s+1} = \beta_v(B_1^s)$ for $1 \leq s < S$ and $\beta_v(B_1^S)$ is undefined. Clearly, S is bounded by $\beta_v^*(x, y)$ (according to the definition of $\beta_v^*(x, y)$). Let B'_1 range over $B_1^1, \ldots, B_1^S$.
   Let $q'_1 = (i'_1, j'_1)$ be the point on the lower boundary of B'_1 such that the path connecting p'_1 and q'_1 consists of only d-steps. Let $snaps_h(q'_1)$ be the set consisting of the point q'_1, the points on the lower boundary of B'_1 of the form $(i'_1 + \Delta(t), j'_1)$ for some $t \geq 0$, and the lower-right corner point of B'_1. For each q'_2 ranging over the set $snaps_h(q'_1)$, there exists at most one point q'_3 on the right boundary of some B_2 (which is either B'_1 or sits above B'_1) such that the path connecting q'_2 and q'_3 consists of only d-steps. As this q'_3 may not be in V, we choose q'_4 to be $snap_v(q'_3)$, which is in V by definition. Note that the path $p'_1q'_1q'_2q'_3q'_4$ is an h-to-v path in $T_{DTW}(x, y)$. We add into E an edge between p'_1 and q'_4 for each q'_4. The length of each added edge connecting p'_1 and q'_4 is the shortest distance between p'_1 and q'_4, which, by Lemma 17, can be computed in $\tilde{O}(1)$ time.
   There is one q'_1 for each B'_1, and there are at most $\log_{1+\epsilon}(m)$ many of q'_2 for each q'_1, and there is at most one q'_3 for each q'_2 and one q'_4 for each q'_3. Therefore, for each p'_1, there are at most $\beta_v^*(x, y) \cdot \log_{1+\epsilon}(m)$ edges added into E.
6. Let p'_1 be a point in V on the left boundary of a block B_1. This step adds into E edges between p'_1 and certain chosen snap points q'_4 such that there are v-to-h paths connecting p'_1 and q'_4. We omit the details that are parallel to those in the previous step. There are at most $\beta_h^*(x, y) \cdot \log_{1+\epsilon}(n)$ edges added into E for each p'_1.

Let us take a moment to discuss how to efficiently compute the lengths of the edges added to E during the construction of $G = \langle V, E \rangle$. That is, how to construct and determine the cost of each dotted path $p'_1q'_1q'_2q'_3q'_4$ depicted in Figure 2. (Note that $q'_4 = snap_v(q'_3)$ is not shown in the figure.)

▶ **Lemma 16.** *Let x and y be two non-empty strings. For $k = \|x\|$ and $\ell = \|y\|$,*

1. *we can compute $\beta_h(B)$ for all the blocks B in $T_{DTW}(x, y)$ in $O(k\ell \cdot \log(k))$ time, and*
2. *we can compute $\beta_v(B)$ for all the blocks B in $T_{DTW}(x, y)$ in $O(k\ell \cdot \log(\ell))$ time.*

Proof. We defer the full proof to the extended version of the paper [41]. ◀

▶ **Lemma 17.** *For each h-to-v path* $(p'_1, q'_1, q'_2, q'_3, q'_4)$*, its length can be computed in* $\tilde{O}(1)$ *time if the five snap points* p'_1*,* q'_1*,* q'_2*,* q'_3*, and* q'_4 *are given.*

Proof. We defer the full proof to the extended version of the paper [41]. ◀

For brevity, we omit the obvious lemma parallel to Lemma 17 that is instead on computing the lengths of v-to-h paths in Õ(1) time.

We are now in a position to state and prove the main theorems of the paper. As noted earlier, the basic idea behind our $(1+\epsilon)$-approximation algorithms is to compute a path-distance through the graph $G = (V, E)$, and show that this distance closely approximates $DTW(x, y)$.

We begin by stating a theorem that parameterizes its running time by $\beta^*(x, y)$ – we will then apply this result to obtain fast running times in the cases where the distance function δ outputs either $O(\log n)$-bit integer values (Theorem 21) or $\{0, 1\}$-values (Theorem 22).

▶ **Theorem 18.** *Let* $x = (a_1, \ldots, a_m)$ *and* $y = (b_1, \ldots, b_n)$ *be two non-empty strings, and let* $\hat{x}$ *and* $\hat{y}$ *denote the run-length encoded versions of the two strings. There exists a* $(1+\epsilon)$*-approximation algorithm (ApproxDTW) for each* $\epsilon > 0$ *that takes* $\hat{x}$ *and* $\hat{y}$ *as its input and returns a value* $\tilde{DTW}(x, y)$ *satisfying* $DTW(x, y) \leq \tilde{DTW}(x, y) \leq (1+\epsilon) \cdot DTW(x, y)$*. Moreover, the worst-case time complexity of this algorithm is* $\tilde{O}(k\ell \cdot \beta^*(x, y)/\epsilon^2)$ *for* $k = \|x\|$ *and* $\ell = \|y\|$*, where* $\beta^*(x, y)$ *is defined in Definition 15.*

Proof. The analysis of the approximation ratio follows as in Section 2. ◀

4.1 Time-Bound for Polynomially-Bounded Letter Distances

In this section, we present a variant of the algorithm ApproxDTW for approximating $DTW(x, y)$ under the general condition that the distances between letters are integer values bounded by some polynomial of the lengths of x and y. The time complexity of this variant, which takes $\hat{x}$ and $\hat{y}$ as its input to compute $DTW(x, y)$, is $\tilde{O}(k\ell/\epsilon^3)$ for $k = \|x\|$ and $\ell = \|y\|$.

▶ **Definition 19.** *Let* δ *be a distance function on letters such that* $\delta(a, b) \geq 1$ *if* $\delta(a, b) \neq 0$*. Given* $\epsilon_1 > 0$*, we use* δ_{ϵ_1} *for the distance function such that* $\delta_{\epsilon_1}(a, b) = 0$ *if* $\delta(a, b) = 0$*, or* $\delta_{\epsilon_1}(a, b) = cpow(1+\epsilon_1, \delta(a, b))$ *if* $\delta(a, b) \geq 1$*, where* $cpow(1+\epsilon_1, \alpha)$ *equals* $(1+\epsilon_1)^t$ *for the least integer* t *such that* $\alpha \leq (1+\epsilon_1)^t$ *holds.*

Please note that $\delta(a, b) \leq \delta_{\epsilon_1}(a, b) \leq (1+\epsilon_1) \cdot \delta(a, b)$ holds for any letters a and b.

▶ **Lemma 20.** *Let* $DTW(\delta)$ *be the DTW distance function where the underlying distance function for letters is* δ*. Given* $\epsilon_1 > 0$*, we have the following inequality for each pair of strings* x *and* y*:*

$$DTW(\delta_{\epsilon_1})(x, y) \leq (1+\epsilon_1) \cdot DTW(\delta)(x, y)$$

Proof. Let P be an optimal full path such that its length based on δ equals $DTW(\delta)(x, y)$. We know that the length of P based on δ_{ϵ_1} is bounded by $(1+\epsilon_1) \cdot DTW(\delta)(x, y)$ since $\delta_{\epsilon_1}(a, b) \leq (1+\epsilon_1) \cdot \delta(a, b)$ holds for any letters a and b. As $DTW(\delta_{\epsilon_1})(x, y)$ is bounded by the length of P based on δ_{ϵ_1}, we have the claimed inequality. ◀

Let $\epsilon_1 > 0$ and $\epsilon_2 > 0$. By Lemma 20, every $(1+\epsilon_2)$-approximation algorithm for DTW based on δ_{ϵ_1} is a $(1+\epsilon_1)\cdot(1+\epsilon_2)$-approximation algorithm for DTW based on δ. For each $\epsilon > 0$, if we choose, for example, $\epsilon_1 = \epsilon/2 - \epsilon^2/2$ and $\epsilon_2 = \epsilon/2$, then we have $(1+\epsilon_1)\cdot(1+\epsilon_2) < 1+\epsilon$, implying that every $(1+\epsilon_2)$-approximation algorithm for DTW based on δ_{ϵ_1} is a $(1+\epsilon)$-approximation algorithm for DTW based on δ.

▶ **Theorem 21.** *Let $x = (a_1, \dots, a_m)$ and $y = (b_1, \dots, b_n)$ be two non-empty strings.*

Assume that the underlying distance function δ satisfies (1) $\delta(a,b) \geq 1$ if $\delta(a,b) \neq 0$ and (2) $\delta(a,b)$ is $\mathrm{poly}(m+n)$ for any letters a in x and b in y. There exists a $(1+\epsilon)$-approximation algorithm for each $\epsilon > 0$ that takes $\hat{x}$ and $\hat{y}$ as its input and returns a value w satisfying $DTW(x,y) \leq w \leq (1+\epsilon)\cdot DTW(x,y)$. And the worst-case time complexity of this algorithm is $\tilde{O}(k\ell/\epsilon^3)$ for $k = \|x\|$ and $\ell = \|y\|$.

Proof. Let $\epsilon_1 = \epsilon/2 - \epsilon^2/2$ and $\epsilon_2 = \epsilon/2$. By Theorem 18, ApproxDTW (as is presented in the proof of Theorem 18) takes $\hat{x}$ and $\hat{y}$ as input and returns a $(1+\epsilon_2)$-approximation of $DTW(\delta_{\epsilon_1})(x,y)$. And the time complexity of the algorithm is $\tilde{\mathrm{O}}(k\ell \cdot \beta^*(x,y)/{\epsilon_2}^2)$.

Note that there are only $\mathrm{O}(\log_{1+\epsilon_1}(m+n))$-many distinct values of $\delta_{\epsilon_1}(a,b)$ for a and b ranging over letters in x and y, respectively. Hence, for the underlying distance function δ_{ϵ_1} on letters, $\beta^*_h(x,y)$ is $\mathrm{O}(\log_{1+\epsilon_1}(m+n))$ and $\beta^*_v(x,y)$ is also $\mathrm{O}(\log_{1+\epsilon_1}(m+n))$, which implies that $\beta^*(x,y)$ is $\mathrm{O}(\log_{1+\epsilon_1}(m+n))$ or simply $\tilde{\mathrm{O}}(1/\epsilon_1)$. Therefore, we can use ApproxDTW to compute a $(1+\epsilon_2)$-approximation of $DTW(\delta_{\epsilon_1})(x,y)$ in $\tilde{\mathrm{O}}(k\ell/\epsilon_1{\epsilon_2}^2)$ time. Since any $(1+\epsilon_2)$-approximation of $DTW(\delta_{\epsilon_1})(x,y)$ is a $(1+\epsilon)$-approximation of $DTW(\delta)(x,y)$, we are done. ◀

4.2 Time-Bound for Constant-Bounded Letter Distances

Assume that there exists a constant N such that $\delta(a,b)$ is an integer less than N for each pair a and b in Σ. For instance, (Σ, δ) satisfies this condition if it is Hamming space (for which N can be set to 2).

▶ **Theorem 22.** *Assume that $\delta(a,b)$ are $O(1)$ for $a, b \in \Sigma$. Then ApproxDTW, the $(1+\epsilon)$-approximation algorithm for DTW given in the proof of Theorem 18, runs in $\tilde{O}(k\ell/\epsilon^2)$ time for $k = \|x\|$ and $\ell = \|y\|$, where $\hat{x}$ and $\hat{y}$ are the input of the algorithm.*

Proof. This theorem follows from Theorem 18 immediately since $\beta^*(x,y)$ is $\mathrm{O}(1)$. ◀

5 Conclusion

We have presented in this paper an algorithm for approximating the DTW distance between two RLE strings. Trading accuracy for efficiency, this algorithm is of (near) quadratic-time complexity and thus, as can be expected, asymptotically faster than the exact DTW algorithm of cubic-time complexity [21], which is currently considered the state-of-art of its kind.

It will be interesting to further investigate whether there exist asymptotically faster approximation algorithms for DTW than the one presented in this paper. In particular, it seems both interesting and challenging to answer the open question as to whether there exists a (near) quadratic-time algorithm for computing the (exact) DTW distance between two RLE strings.

References

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

2 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for lcs and other sequence similarity measures. In *56th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 59–78. IEEE, 2015.

3 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 International Symposium on Computational Geometry, SoCG 2016, June 14-18, 2016, Boston, MA, USA*, pages 6:1–6:16, 2016. doi:10.4230/LIPIcs.SoCG.2016.6.

4 Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Polylogarithmic approximation for edit distance and the asymmetric query complexity. In *Proceedings of the 51st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 377–386. IEEE, 2010.

5 Alexandr Andoni and Negev Shekel Nosatzki. Edit distance in near-linear time: It's a constant factor. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 990–1001. IEEE, 2020.

6 Alexandr Andoni and Krzysztof Onak. Approximating edit distance in near-linear time. *SIAM J. Comput.*, 41(6):1635–1648, 2012.

7 Alberto Apostolico, Gad M Landau, and Steven Skiena. *Matching for run-length encoded strings*. IEEE, 1997.

8 Ora Arbell, Gad M Landau, and Joseph SB Mitchell. Edit distance of run-length encoded strings. *Information Processing Letters*, 83(6):307–314, 2002.

9 Ziv Bar-Yossef, T.S. Jayram, Robert Krauthgamer, and Ravi Kumar. Approximating edit distance efficiently. In *Proceedings of 45th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 550–559. IEEE, 2004.

10 Joshua Brakensiek and Aviad Rubinstein. Constant-factor approximation of near-linear edit distance in near-linear time. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 685–698, 2020.

11 Vladimir Braverman, Moses Charikar, William Kuszmaul, David Woodruff, and Lin Yang. The one-way communication complexity of dynamic time warping distance. Manuscript submitted for publication, 2018.

12 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *56th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 79–97. IEEE, 2015.

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

14 EG Caiani, A Porta, G Baselli, M Turiel, S Muzzupappa, F Pieruzzi, C Crema, A Malliani, and S Cerutti. Warped-average template technique to track on a cycle-by-cycle basis the cardiac filling phases on left ventricular volume. In *Computers in Cardiology 1998*, pages 73–76. IEEE, 1998.

15 Diptarka Chakraborty, Debarati Das, Elazar Goldenberg, Michal Kouckỳ, and Michael Saks. Approximating edit distance within constant factor in truly sub-quadratic time. *Journal of the ACM (JACM)*, 67(6):1–22, 2020.

16 Diptarka Chakraborty, Elazar Goldenberg, and Michal Kouckỳ. Streaming algorithms for embedding and computing edit distance in the low distance regime. In *Proceedings of the 48th Annual Symposium on Theory of Computing (STOC)*, pages 712–725. ACM, 2016.

17 Moses Charikar, Ofir Geri, Michael P Kim, and William Kuszmaul. On estimating edit distance: Alignment, dimension reduction, and embeddings. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

18 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.

19 Raphaël Clifford, Pawel Gawrychowski, Tomasz Kociumaka, Daniel P Martin, and Przemyslaw Uznanski. Rle edit distance in near optimal time. In *44th International Symposium on Mathematical Foundations of Computer Science (MFCS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

20 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 *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 987–996. ACM, 2012.

21 Vincent Froese, Brijnesh J. Jain, Maciej Rymar, and Mathias Weller. Fast exact dynamic time warping on run-length encoded time series. *CoRR*, abs/1903.03003, 2019. arXiv:1903.03003.

22 Omer Gold and Micha Sharir. Dynamic time warping and geometric edit distance: Breaking the quadratic barrier. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 25:1–25:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ICALP.2017.25.

23 Guan Shieng Huang, Jia Jie Liu, and Yue Li Wang. Sequence alignment algorithms for run-length-encoded strings. In *International Computing and Combinatorics Conference*, pages 319–330. Springer, 2008.

24 Michal Koucký and Michael Saks. Constant factor approximations to edit distance on far input pairs in nearly linear time. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 699–712, 2020.

25 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 International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 80:1–80:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ICALP.2019.80.

26 William Kuszmaul. Binary dynamic time warping in linear time. *arXiv preprint*, 2021. arXiv:2101.01108.

27 Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582, 1998.

28 T. Warren Liao. Clustering of time series data – A survey. *Pattern Recognit.*, 38(11):1857–1874, 2005. doi:10.1016/j.patcog.2005.01.025.

29 Jia Jie Liu, Guan-Shieng Huang, Yue-Li Wang, and Richard CT Lee. Edit distance for a run-length-encoded string and an uncompressed string. *Information Processing Letters*, 105(1):12–16, 2007.

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

31 William J Masek and Michael S Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System sciences*, 20(1):18–31, 1980.

32 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.

33 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. arXiv:1003.4083.

34 Mario E Munich and Pietro Perona. Continuous dynamic time warping for translation-invariant curve alignment with applications to signature verification. In *Proceedings of 7th International Conference on Computer Vision*, volume 1, pages 108–115, 1999.

35 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.

36 Yoshifumi Sakai and Shunsuke Inenaga. A faster reduction of the dynamic time warping distance to the longest increasing subsequence length. *arXiv preprint*, 2020. `arXiv:2005.09169`.

37 Yoshifumi Sakai and Shunsuke Inenaga. A reduction of the dynamic time warping distance to the longest increasing subsequence length. In *31st International Symposium on Algorithms and Computation (ISAAC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

38 Nathan Schaar, Vincent Froese, and Rolf Niedermeier. Faster binary mean computation under dynamic time warping. *arXiv preprint*, 2020. `arXiv:2002.01178`.

39 Anooshiravan Sharabiani, Houshang Darabi, Samuel Harford, Elnaz Douzali, Fazle Karim, Hereford Johnson, and Shun Chen. Asymptotic dynamic time warping calculation with utilizing value repetition. *Knowl. Inf. Syst.*, 57(2):359–388, 2018. `doi:10.1007/s10115-018-1163-4`.

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

41 Zoe Xi and William Kuszmaul. Approximating dynamic time warping distance between run-length encoded strings. *CoRR*, abs/2207.00915, 2022. `arXiv:2207.00915`.

42 Rex Ying, Jiangwei Pan, Kyle Fox, and Pankaj K Agarwal. A simple efficient approximation algorithm for dynamic time warping. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 21. ACM, 2016.

43 Yunyue Zhu and Dennis Shasha. Warping indexes with envelope transforms for query by humming. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 181–192. ACM, 2003.