Dynamic String Alignment

Panagiotis Charalampopoulos
Department of Informatics, King's College London, UK
Institute of Informatics, University of Warsaw, Poland
panagiotis.charalampopoulos@kcl.ac.uk

Tomasz Kociumaka
Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
kociumaka@mimuw.edu.pl

Shay Mozes
Efi Arazi School of Computer Science, The Interdisciplinary Center Herzliya, Israel
smozes@idc.ac.il

Abstract

We consider the problem of dynamically maintaining an optimal alignment of two strings, each of length at most n, as they undergo insertions, deletions, and substitutions of letters. The string alignment problem generalizes the longest common subsequence (LCS) problem and the edit distance problem (also with non-unit costs, as long as insertions and deletions cost the same). The conditional lower bound of Backurs and Indyk [J. Comput. 2018] for computing the LCS in the static case implies that strongly sublinear update time for the dynamic string alignment problem would refute the Strong Exponential Time Hypothesis. We essentially match this lower bound when the alignment weights are constants, by showing how to process each update in $\tilde{\mathcal{O}}(n)$ time.[1] When the weights are integers bounded in absolute value by some $w = n^{\mathcal{O}(1)}$, we can maintain the alignment in $\tilde{\mathcal{O}}(n \cdot \min\{\sqrt{n}, w\})$ time per update. For the $\tilde{\mathcal{O}}(nw)$-time algorithm, we heavily rely on Tiskin's work on semi-local LCS, and in particular, in an implicit way, on his algorithm for computing the $(\min, +)$-product of two simple unit-Monge matrices [Algorithmica 2015]. As for the $\tilde{\mathcal{O}}(n\sqrt{n})$-time algorithm, we employ efficient data structures for computing distances in planar graphs.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases string alignment, edit distance, longest common subsequence, (unit-)Monge matrices, $(\min, +)$-product

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.9

Funding *Panagiotis Charalampopoulos*: Partially supported by ERC grant TOTAL (no. 677651) under the EU's Horizon 2020 Research and Innovation Programme.
Tomasz Kociumaka: Supported by ISF grants no. 1278/16 and 1926/19, a BSF grant no. 2018364, and an ERC grant MPM (no. 683064) under the EU's Horizon 2020 Research and Innovation Programme.
Shay Mozes: Partially supported by Israel Science Foundation grant ISF no. 592/17.

1 Introduction

The problems of computing an optimal string alignment, a longest common subsequence (LCS), or the edit distance of two strings have been studied for more than 50 years. In the string alignment problem, we are given weights w_{match} for aligning a pair of matching letters, w_{mis} for aligning a pair of mismatching letters, and w_{gap} for letters that are not aligned, and the goal to compute an alignment with maximum weight. The edit distance $d_E(S, T)$ of two strings S and T is the minimum cost of transforming string S to string T using insertions, deletions, and substitutions of letters, under specified costs c_{ins}, c_{del}, and

[1] The $\tilde{\mathcal{O}}(\cdot)$ notation suppresses $\log^{\mathcal{O}(1)} n$ factors.

© Panagiotis Charalampopoulos, Tomasz Kociumaka, and Shay Mozes;
licensed under Creative Commons License CC-BY

c_{sub}, respectively. When all costs are 1, this is also known as the Levenshtein distance of S and T [24]. Note that if $c_{ins} = c_{del}$, the edit distance problem is a special case of the string alignment problem, with $w_{match} = 0$, $w_{mis} = -c_{sub}$, and $w_{gap} = -c_{ins} = -c_{del}$. In turn, the LCS problem can be seen as a special case of the edit distance problem: Let the length of an LCS of S and T be denoted by $\texttt{LCS}(S,T)$. Then, for $c_{ins} = c_{del} = 1$ and $c_{sub} = 2$, we have $d_E(S,T) = |S| + |T| - 2 \cdot \texttt{LCS}(S,T)$. In this work, we consider the dynamic version of the string alignment problem, in which the strings S and T, each of length at most n, are maintained subject to insertions, deletions, and substitutions of letters, and we are to report an optimal alignment after each such update.

The textbook dynamic programming (DP) $\mathcal{O}(n^2)$-time algorithm for the (static) LCS and edit distance problems has been rediscovered several times, e.g. in [38, 28, 29, 30, 39]. When the desired output is just the edit distance or the length of an LCS, the space required by the DP algorithm is trivially $\mathcal{O}(n)$ as one needs to store just two rows or columns of the DP matrix. Hirschberg showed how to actually retrieve an LCS within $\mathcal{O}(n^2)$ time using only $\mathcal{O}(n)$ space [17]. A line of works has improved the complexity of the classic DP algorithm by factors polylogarithmic with respect to n (see [25, 40, 11, 5, 15]).

While we are the first to consider the dynamic string alignment problem in the general variant where edits are allowed in any position of either of the strings, already the DP algorithm is inherently "dynamic" in the sense that it supports appending a letter and deleting the last letter in either of the strings in linear time. A series of works examined variants of incremental and decremental LCS and edit distance problems [23, 21, 18]. The most general of these variants was considered by Tiskin in [32], where he presented a linear-time algorithm for maintaining an LCS in the case that both strings are subject to the following updates: prepending or appending a letter, and deleting the first or the last letter. Tiskin's solution not only maintains the LCS, but implicitly also the *semi-local LCS information*: the LCS lengths between all prefixes of S (resp. T) and all suffixes of T (resp. S), as well as the LCS between S (resp. T) and all fragments (substrings) of T (resp. S).

One of the main technical contributions of Tiskin in this area is an efficient algorithm for computing the $(\min, +)$ *product* (also known as *distance product*) of two simple unit-Monge matrices [37].[2] The algorithm itself and the ideas behind it have found numerous applications to variants of the LCS and string alignment problems. We refer the reader to Tiskin's monograph [32] as well as [33, 34, 35, 31, 36].

On the lower-bound side, Backurs and Indyk showed that an $\mathcal{O}(n^{2-\epsilon})$-time algorithm for computing the edit distance of two strings of length at most n would refute the Strong Exponential Time Hypothesis (SETH) [4]. Bringmann and Künnemann generalized this conditional lower bound by showing that it holds even for binary strings under any non-trivial assignment of weights c_{ins}, c_{del}, and c_{sub} [7] – an assignment of weights is trivial if it allows one to infer the edit distance in constant time. Further consequences of subquadratic-time algorithms for the edit distance or LCS problems where shown by Abboud et al. [1]; interestingly, they proved that even shaving arbitrarily large polylogarithmic factors from n^2 would have major consequences. In light of the above results, an $\mathcal{O}(n^{1-\epsilon})$-time algorithm maintaining an optimal string alignment of two strings of length $\mathcal{O}(n)$ subject to edit operations seems highly unlikely, as it would directly imply an $\mathcal{O}(n^{2-\epsilon})$-time algorithm for the static version of the problem.

[2] A matrix M is a Monge matrix if $M[i,j] + M[i',j'] \leq M[i',j] + M[i,j']$ for all $i < i'$ and $j < j'$ [26]. An $n \times n$ Monge matrix is a simple unit-Monge matrix if its leftmost column and bottommost row consist of zeroes, while its rightmost column and topmost row consist of subsequent integers from 0 to $n-1$ [37].

Our results and approach. We heavily rely on Tiskin's work on efficient distance multiplication of simple unit-Monge matrices and its applications to the string alignment problem. Specifically for the LCS problem, Tiskin showed that the semi-local LCS information of two strings of length at most n can be retrieved from an $\tilde{\mathcal{O}}(n)$-size representation as a *permutation matrix* $P_{S,T}$. Based on his efficient algorithm for computing the $(\min,+)$-product of two simple unit-Monge matrices, he showed that given permutation matrices $P_{S,T}$ and $P_{S,T'}$, one can efficiently compute $P_{S,TT'}$. We formalize this in the preliminaries (Section 2).

In Section 3, we first describe our algorithm for maintaining an LCS of two strings S and T in $\tilde{\mathcal{O}}(n)$ time per edit operation, and then we extend it to maintaining a string alignment under integer weights. Our algorithm maintains a hierarchical partition of strings S and T to fragments of length roughly 2^s for each scale s, $0 \le s \le \log n$, and permutation matrices P_{S_i,T_j} for all pairs of fragments (S_i, T_j) at each scale. Then, upon an update to S or T, we need to update $\Theta(n/2^s)$ permutation matrices at each scale s. This is in contrast with the sequential approach of combining the permutation matrices in Tiskin's work.

In Section 4, we show that efficient data structures for computing distances in planar graphs outperform the approach outlined above when the alignment weights cannot be expressed as small integers.

2 Preliminaries

Let $T = T[0]T[1]\cdots T[n-1]$ be a *string* of length $|T| = n$ over an alphabet Σ. The elements of Σ are called *letters*. For two positions i and j in T, we denote by $T[i\,..\,j]$ the *fragment* of T that starts at position i and ends at position j (the fragment is empty if $i < j$). The fragment $T[i\,..\,j]$ is an *occurrence* of the underlying *substring* $T[i]\cdots T[j]$. A fragment of T is represented in $\mathcal{O}(1)$ space by specifying the indices i and j. We sometimes denote the fragment $T[i\,..\,j]$ as $T[i\,..\,j+1)$. A *prefix* of T is a fragment that starts at position 0 ($T[0\,..\,j]$) and a *suffix* is a fragment that ends at position $n-1$ ($T[i\,..\,n-1]$ or $T[i\,..\,n)$).

A *longest common subsequence* (LCS) of two strings S and T is a longest string that is a subsequence of both S and T. We denote the length of an LCS of S and T by $\texttt{LCS}(S,T)$.

▶ **Example 1.** An LCS of $S =$ `acbcddaaea` and $T =$ `abbbccdec` is `abcde`; $\texttt{LCS}(S,T) = 5$.

For strings S and T, of length m and n respectively, the alignment graph $G_{S,T}$ of S and T is a directed acyclic graph with vertex set $\{v_{i,j} : 0 \le i \le m, 0 \le j \le n\}$. For every $0 \le i \le m$ and $0 \le j \le n$, the graph $G_{S,T}$ has the following edges (defined only if both endpoints exist):

- $v_{i,j}v_{i+1,j}$ and $v_{i,j}v_{i,j+1}$ of length 0,
- $v_{i,j}v_{i+1,j+1}$ of length 1, present if and only if $S[i] = T[j]$.

Intuitively, $G_{S,T}$ is an $(m+1)\times(n+1)$ grid graph (with length-0 edges) augmented with length-1 diagonal edges corresponding to matching letters of S and T. We think of the vertex $v_{0,0}$ as the top left vertex of the grid and the vertex $v_{m,n}$ as the bottom right vertex of the grid. We shall refer to the rows and columns of $G_{S,T}$ in a natural way. It is easy to see that $\texttt{LCS}(S,T)$ equals the length of the highest scoring path between $v_{0,0}$ and $v_{m,n}$ in $G_{S,T}$.

We index matrices from 0. Let us define some matrices of interest.

▶ **Definition 2.** *The distribution matrix $\sigma(M)$ of an $m \times n$ matrix M is the $(m+1)\times(n+1)$ matrix satisfying $\sigma(M)[i,j] = \sum_{r\ge i, c<j} M[r,c]$.*

▶ **Definition 3.** *An $n \times n$ binary matrix is a permutation matrix if it has exactly one 1 entry in each row and each column. Such a matrix can be represented in $\mathcal{O}(n)$ space.*

By constructing a 2D orthogonal range counting data structure over the non-zero entries of a permutation matrix, one obtains the following lemma.

▶ **Lemma 4** ([32, Theorem 2.15]; [9]). *An $n \times n$ permutation matrix P can be preprocessed $\mathcal{O}(n\sqrt{\log n})$ time so that any entry of $\sigma(P)$ can be retrieved in time $\mathcal{O}(\log n/\log\log n)$.*

Let $\diamond$ be a wildcard letter, i.e., a letter that matches all letters. Tiskin [32] defines an $(m+n+1)\times(m+n+1)$ distance matrix $H_{S,T}$ over $G_{S,\diamond^m T\diamond^m}$ so that $H_{S,T}[i,j]$ equals the hightest weight of a path from $v_{0,i}$ to $v_{m,m+j}$ in $G_{S,\diamond^m T\diamond^m}$. Note that if $j = i-m$, then $H_{S,T}[i,j]=0$. By convention, if $j < i-m$, then $H_{S,T}[i,j] = j-(i-m) < 0$. The matrix $H_{S,T}$ captures so-called *semi-local LCS* values as follows; see Figure 1 for an illustration.

$$H_{S,T}[i,j] = \begin{cases} \texttt{LCS}(S[m-i\,..\,m), T[0\,..\,j)) + m - i & \text{if } i \le m \text{ and } j \le n,\\ \texttt{LCS}(S[0\,..\,m+n-j), T[i-m\,..\,n)) + j - n & \text{if } i \ge m \text{ and } j \ge n,\\ \texttt{LCS}(S[m-i\,..\,m+n-j), T) + m - i + j - n & \text{if } i \le m \text{ and } n+i \ge j \ge n,\\ m & \text{if } n+i \le j,\\ \texttt{LCS}(S, T[i-m\,..\,j)) & \text{if } i \ge m \text{ and } i-m \le j \le n,\\ j+m-i & \text{if } j \le i-m. \end{cases}$$

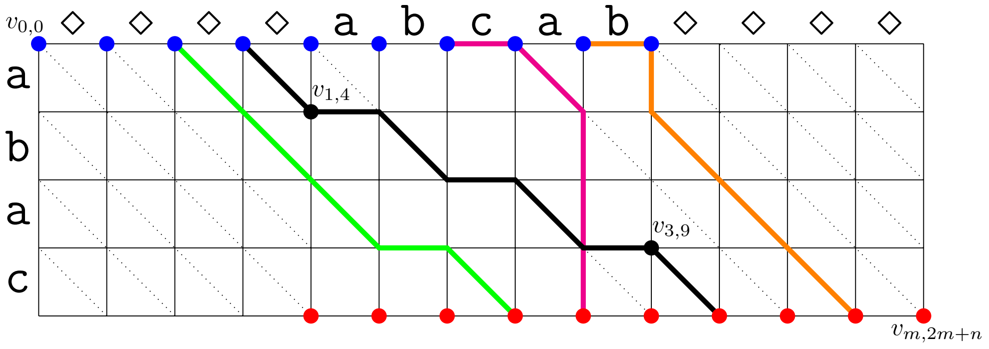

Figure 1 The figure illustrates how $H_{S,T}$ captures semi-local LCS information for $S=$ abac and $T=$ abcab. We have $m=4$ and $n=5$. The value $H_{S,T}(i,j)$ captures the length of the highest scoring path from the i-th blue node to the j-th red node in the above figure (in the left-to-right order). The underlying idea is that when there are wildcards $\diamond$ involved, one may always choose to use the diagonal edges corresponding to them and then fill in the rest of the path. Let us analyze one of the cases thoroughly, the analysis of the other cases is analogous.

- The highest weight of a path from $v_{0,3}$ to $v_{4,4+6}$ is 4, which corresponds to $H_{S,T}(3,6) = 4 = \texttt{LCS}(S[4-3,9-6),T)+4-3+6-5 = \texttt{LCS}(S[1,2],T)+2$ (case 3 of the equation above). The highest scoring path (in black), after trimming diagonal edges corresponding to wildcards, yields a highest scoring path from $v_{1,4}$ to $v_{3,4+5}$. Its weight indeed corresponds to $\texttt{LCS}(S[1\,..\,2],T)=2$.
- The $v_{0,2}$-to-$v_{4,4+3}$ highest scoring path (in green) illustrates case 1 of the equation above: $H_{S,T}(2,3) = 4 = \texttt{LCS}(S[4-2\,..\,4), T[0\,..\,3)) + 4 - 2 = \texttt{LCS}(S[2\,..\,3], T[0\,..\,2]) + 2$.
- The $v_{0,8}$-to-$v_{4,4+8}$ highest scoring path (in orange) illustrates case 2 of the equation above: $H_{S,T}(8,8) = 3 = \texttt{LCS}(S[0\,..\,9-8), T[8-4\,..\,5)) + 8 - 5 = \texttt{LCS}(S[0], T[4]) + 3$.
- The $v_{0,6}$-to-$v_{4,4+4}$ highest scoring path (in magenta) illustrates case 5 of the equation above: $H_{S,T}(6,4) = 1 = \texttt{LCS}(S, T[6-4\,..\,4)) = \texttt{LCS}(S, T[2\,..\,3])$.

.

▶ Remark 5. Let us try to provide some extra intuition by considering the *indel distance*, for which we get a more uniform formula. The indel distance of two strings, denoted $\delta(S,T)$, is the minimum number of insertions and deletions that are needed to transform S to T. In other words, $\delta(S,T) = |S| + |T| - 2\texttt{LCS}(S,T)$. Then $2m + j - i - 2H_{S,T}[i,j]$, which can be interpreted as the number of length-0 edges on the highest scoring path from $v_{0,i}$ to $v_{m,m+j}$ in $G_{S,\diamond^m T \diamond^m}$, admits a more uniform formula:

$$2m+j-i-2H_{S,T}[i,j] = \begin{cases} \delta(S[m-i\,..\,m), T[0\,..\,j)) & \text{if } i \le m \text{ and } j \le n,\\ \delta(S[0\,..\,m+n-j), T[i-m\,..\,n)) & \text{if } i \ge m \text{ and } j \ge n,\\ \delta(S[m-i\,..\,m+n-j), T) & \text{if } i \le m \text{ and } n+i \ge j \ge n,\\ j-i & \text{if } n+i \le j,\\ \delta(S, T[i-m\,..\,j)) & \text{if } i \ge m \text{ and } i-m \le j \le n,\\ i-j & \text{if } j \le i-m. \end{cases}$$

We now return to the LCS problem. Tiskin shows that the $(n+m) \times (n+m)$ matrix $P_{S,T}$ defined as

$$P_{S,T}[i,j] = H_{S,T}[i,j] + H_{S,T}[i+1,j+1] - H_{S,T}[i+1,j] - H_{S,T}[i,j+1], \qquad (1)$$

is a permutation matrix and satisfies $H_{S,T}[i,j] = j + m - i - \sigma(P_{S,T})[i,j]$. Note that for constant-length strings S and T, the matrix $P_{S,T}$ can be computed naively in constant time from $H_{S,T}$. Conversely, each entry of $H_{S,T}$ can be retrieved in time $\mathcal{O}(\log(n+m)/\log\log(n+m))$ after an $\mathcal{O}((n+m)\sqrt{\log(n+m)})$-time preprocessing of $P_{S,T}$ by Lemma 4. Crucially for our approach, Tiskin shows the following result.

▶ **Theorem 6** ([32, Theorem 4.21]).

(a) *Given $P_{S,T}$ and $P_{S,T'}$ for three strings S,T,T', each of length at most n, one can compute $P_{S,TT'}$ in $\mathcal{O}(n\log n)$ time.*

(b) *Given $P_{T,S}$ and $P_{T',S}$ for three strings S,T,T', each of length at most n, one can compute $P_{TT',S}$ in $\mathcal{O}(n\log n)$ time.*

Actually, only part (a) of the above theorem is stated explicitly in [32]. Part (b) can be derived by symmetry as follows. One can check using the characterization of $H_{S,T}$ in terms of the semi-local LCS values that $H_{S,T}[i,j] = H_{T,S}[n+m-i, n+m-j] + m - i + j - n$; see [32, Lemma 4.14]. In particular, this means that $H_{S,T}$ can be obtained from $H_{T,S}$ by first performing a 180-degree rotation and then off-setting the values in every row i by $m-i$ and the values in every column j by $j-n$. This, in turn, means that $P_{T,S}$ can be obtained from $P_{S,T}$ just through a 180-degree rotation, as the offsets are cancelled out in the computation of $P_{S,T}[i,j]$ from $H_{T,S}$; see (1). Thus, we can rotate $P_{T,S}$ and $P_{T',S}$ to obtain $P_{S,T}$ and $P_{S,T'}$, compute $P_{S,TT'}$ using Theorem 6(a), and then rotate $P_{S,TT'}$ to obtain $P_{TT',S}$.

3 Main Algorithm

We show how to maintain the permutation matrix $P_{S,T}$ in $\mathcal{O}((m+n)\log(m+n))$ time per update when the strings S and T undergo substitutions, insertions, and deletions of single letters. Within the stated update time we can recompute the orthogonal range counting data structure that allows us to report, in $\mathcal{O}(\log(m+n)/\log\log(m+n))$ time, any element of the matrix $H_{S,T}$.

The high-level idea is to maintain the permutation matrices $P_{A,B}$ for fragments A of S and B of T, at exponentially growing scales. Local changes to S and T, such as substitutions, insertions, and deletions, only affect a single fragment at each scale. We can therefore use Theorem 6 to recompute the affected matrices efficiently in a bottom-up fashion.

We first describe the maintenance of a data structure that can only support substitutions in order to demonstrate the general approach. We will then describe how to also support insertions and deletions.

3.1 Supporting Only Substitutions

We can assume that both S and T are of length n and that n is a power of two; otherwise, we pad S with \$ characters and T with # characters such that $\$ \neq \#$ and neither \$ nor # is in the alphabet. We define $\log n + 1$ scales, where at scale s, each of S and T is partitioned into non-overlapping fragments of length 2^s. At every scale, and for every pair of fragments S_i and T_j of S and T, respectively, we store the permutation matrix P_{S_i,T_j} corresponding to H_{S_i,T_i}. At scale s, there are $(n/2^s)^2$ matrices, each stored in $\mathcal{O}(2^s)$ space. Thus, the overall space required by the data structure is $\mathcal{O}(n^2)$. Building the data structure in a bottom-up manner requires time $\sum_{s=0}^{\log n}(n/2^s)^2 \cdot 2^s \cdot s = \mathcal{O}(n^2)$ by Theorem 6.

Suppose, without loss of generality, that a letter of S is substituted (the other case is symmetric). We work in order of increasing scales $s = 0, 1, \ldots \log n$. Let S_i be the unique fragment of S in scale s that contains the substituted letter. We recompute the matrices P_{S_i,T_j} for each one of the $n/2^s$ fragments T_j of T at scale s. At scale $s = 0$, both S_i and T_j consist of single letters, and we recompute the constant-size permutation matrices P_{S_i,T_j} from scratch in total $\mathcal{O}(n)$ time. (In fact, there are only two types of matrices, one corresponding to the case that the letter S_i matches the letter T_j, and the other corresponding to a mismatch.) To recompute a matrix P_{S_i,T_j} at scale $s > 0$, let S'_i, S''_i be the two fragments of S at scale $s-1$ such that $S_i = S'_i S''_i$. Similarly, let T'_j, T''_j be the two fragments of T at scale $s-1$ such that $T_j = T'_j T''_j$. We repeatedly apply Theorem 6 to $P_{S'_i,T'_j}, P_{S'_i,T''_j}, P_{S''_i,T'_j}, P_{S''_i,T''_j}$ to obtain P_{S_i,T_j} in $\mathcal{O}(s \cdot 2^s)$ time. Thus, the total time to update all affected permutation matrices at all scales (and, in particular, to obtain the matrix $P_{S,T}$) is $\sum_{s=0}^{\log n} \frac{n}{2^s} \cdot s \cdot 2^s = \mathcal{O}(n \log^2 n)$.

3.2 Supporting Insertions and Deletions

To support insertions and deletions we use the same approach. However, as each update increases or decreases the length of the string it is applied to, we can no longer use fixed-length fragments at each scale. At each scale s, we maintain a partition of each string into consecutive fragments, each of length between $\frac{1}{4} \cdot 2^s$ and $2 \cdot 2^s$, such that the partition at scale s is a refinement of the partition at scale $s+1$. Let us denote by R_s (resp. C_s) the partition of S (resp. T) at level s. We only describe the process for S; the string T is handled analogously. The *refinement property* for R_s can be stated formally as follows. For any $s' > s$, for each fragment $S[a \mathinner{..} b] \in R_s$ there exists a fragment $S[a' \mathinner{..} b'] \in R_{s'}$ with $a' \leq a \leq b \leq b'$. We maintain each R_s as a linked list of the fragments, which are represented by their start and end indices, sorted by the start indices in increasing order. Upon an update in S, we update the partitions in a bottom-up manner.

Let us first describe how to insert a letter in S after the letter at position k. We first scan R_s for all s and increment by 1 all the start indices that are greater than k and all the end indices that are at least k. This way, the newly inserted letter is assigned to a unique fragment in each partition. Then, we process the scales in increasing order, starting from scale 0. If the fragment $U_0 \in R_0$ that contains the newly inserted letter has just become of length greater than $2 \cdot 2^0 = 2$, then we split U_0 into two fragments of length at most 2. Note that this potential split does not violate the refinement property. Then, we proceed to the next scale. Generally, at scale s, if the length of the fragment $U_s \in R_s$ that contains the newly inserted letter does not exceed $2 \cdot 2^s$, we just proceed to the next scale. Otherwise, we need to make adjustments, ensuring that the refinement property is not violated. Note that, if $|U_s| = 2 \cdot 2^s + 1$, then, since fragments at scale $s-1$ have been already processed and respect the length constraint, the refinement property implies that U_s is the concatenation of at least three (and at most nine) fragments $V_1, V_2, \ldots, V_t$ at scale $s-1$. Let the middle letter of U_s belong to V_i. Then, either $\sum_{g<i} |V_g| \geq 2^s/4$ or $\sum_{g>i} |V_g| \geq 2^s/4$; let us assume without loss of generality that we are in the first case. We replace U at scale s by $V_1 \cdots V_{i-1}$ and $V_i \cdots V_t$. If such a replacement happens at the highest scale s (with $U_s = S$), then we create a new level $s+1$ with $R_{s+1} = \{U_s\} = \{S\}$. Note that the refinement property is maintained and the whole procedure requires $\mathcal{O}(m)$ time.

We now treat the complementary case of deleting $S[k]$. Again, we first scan R_s for all scales s and decrement by 1 all the start/end indices that are at least k – ensuring that none of them becomes negative. If some fragment becomes of length 0, then we remove it. We again process levels in increasing order. Suppose that the fragment $U_s \in R_s$ that contained the deleted letter has just become shorter than $\frac{1}{4} \cdot 2^s$. If R_s is the top level of the decomposition, then we simply remove this level. Otherwise, consider the fragment $U_{s+1} \in R_{s+1}$ that contains U_s. Note that, $1 \leq |U_s| = \frac{1}{4} \cdot 2^s - 1$ implies that the length $|U_s| + 1$ of the fragment corresponding to U_s prior to the deletion is smaller than $\frac{1}{4} \cdot 2^{s+1}$, and hence $|U_{s+1}| > |U_s|$. Thus, there exists a fragment V at scale s that is adjacent to U_s and is also a subfragment of U_{s+1}. Let us assume without loss of generality that V lies to the right of U_s – the other case is symmetric. If $|V| < \frac{7}{4} \cdot 2^s$, then we can just replace U_s and V in R_s by their concatenation, $U_s V$. Otherwise, let the first element of the decomposition of V at scale $s-1$ be X. In this case, we can replace U_s and V in R_s by $U_s X$ and $Y = V[|X| \mathinner{.\,.} |V|-1]$, since $\frac{1}{4} \cdot 2^s \leq |U_s X| < \frac{1}{4} \cdot 2^s + 2 \cdot 2^{s-1} < 2 \cdot 2^s$ and $\frac{1}{4} \cdot 2^s < \frac{7}{4} \cdot 2^s - 2 \cdot 2^{s-1} \leq |V| - |X| = |Y| < |V| \leq 2 \cdot 2^s$. The refinement property is maintained and the whole procedure requires $\mathcal{O}(m)$ time.

We maintain $P_{A,B}$ for each pair of fragments $(A, B) \in R_s \times C_s$ at scale s. In the case that R_s simply consists of S at scale s, while T is still fragmented, we consider R_j for any $j > s$ to simply consist of S. (Symmetrically for the opposite case.) The number of pairs of fragments that are affected at scale s is $\mathcal{O}((n+m)/2^s)$. We compute $P_{A,B}$, for each such pair (A, B), using a constant number of applications of Theorem 6 in $\mathcal{O}(s \cdot 2^s)$ time. Thus, the total time to handle scale s is $\mathcal{O}((n+m)s)$ and the total time to handle all scales is $\mathcal{O}((m+n)\log^2(m+n))$.

▶ Remark 7. Deletions of fragments of either of the strings can also be processed within the same time complexity with a straightforward generalisation of the above process.

Obtaining the longest common subsequence. We now describe how one can obtain the longest common subsequence, and not just its length, within $\tilde{\mathcal{O}}(n+m)$ time. Let us consider the following auxiliary problem: given some pair of fragments S_i, T_j at scale $s > 0$, compute the longest common subsequence of either some prefix of S_i (resp. T_j) and some suffix of T_j (resp. S_i), or some fragment of S_i (resp. T_j) and T_j (resp. S_i). Consider the refinement, at scale $s-1$, of S_i to $U_1, \ldots, U_k$ and of T_j to $V_1, \ldots, V_\ell$. Let $G_{S,T}(S[i_1 \mathinner{.\,.} i_2], T[j_1 \mathinner{.\,.} j_2])$ be the subgraph of $G_{S,T}$ induced by the set of vertices $\{v_{i',j'} : i_1 \leq i' \leq i_2 + 1, j_1 \leq j' \leq j_2 + 1\}$. Our aim is to decompose the highest scoring path in scope (say $v_{a,b}$-to-$v_{c,d}$) into subpaths, each lying entirely on some $G_{S,T}(U_r, V_t)$. We can then apply this procedure recursively.

P_{S_i,T_j} was obtained from the $k \times \ell$ matrices P_{U_t,V_r} through some order of applications of Theorem 6. We can store such intermediate matrices, preprocessed as in Lemma 4, without any extra asymptotic cost in the complexities. We refine the path by considering the reverse order. For clarity of presentation, let us assume that $k = \ell = 2$ and the intermediate matrices were $P_{U_1U_2,V_1}$ and $P_{U_1U_2,V_2}$. We can decompose the path to at most two subpaths, one lying entirely on $J_1 = G_{S,T}(U_1U_2, V_1)$ and one lying entirely on $J_2 = G_{S,T}(U_1U_2, V_2)$. The case that both $v_{a,b}$ and $v_{c,d}$ lie on one of J_1 or J_2 is trivial. In the other case, we wish to find a node that lies on both J_1 and J_2 and is on the path. To this end, we query $P_{U_1U_2,V_1}$ (resp. $P_{U_1U_2,V_2}$) for the length of the highest scoring $v_{a,b}$-to-u (resp. u-to-$v_{c,d}$) path for all nodes u that belong to both J_1 and J_2. Using Lemma 4, this can be done in $\mathcal{O}(2^s \cdot s/\log s)$ time (for $s > 0$). Any u for which the sum of these values equals the length of the highest scoring $v_{a,b}$-to-$v_{c,d}$ path is a valid vertex to decompose the path. We then recurse, further refining the path. Note that the $v_{a,b}$-to-$v_{c,d}$ path gets decomposed into $\mathcal{O}((n+m)/2^s)$ pieces at scale s, for all s. Hence, by summing over all scales, the total time required for applying this procedure is $\mathcal{O}((n+m)\log^2(n+m))$.

Fragment-to-fragment LCS queries. Our data structure also enables us to answer queries of the type $\texttt{LCS}(S[i_1\,..\,i_2], T[j_1\,..\,j_2])$ in time $\tilde{\mathcal{O}}(n+m)$ (in fact, in time $\tilde{\mathcal{O}}(1+i_2-i_1+j_2-j_1)$). Note that $G_{S,T}(S[i_1\,..\,i_2], T[j_1\,..\,j_2])$ can be decomposed in $\tilde{\mathcal{O}}(n+m)$ time to multiple pieces $G_{S,T}(U,V)$, overlapping at their boundaries, such that U and V are of the same scale and there are $\mathcal{O}((n+m)/2^s)$ pairs (U,V) of scale s. This can be done, intuitively, using a greedy approach, that each time uses a piece from the largest possible scale. One can also think of this as extending a rectangle using a constant number of layers consisting of pieces corresponding to pairs of strings at scale s, in order of decreasing s. Finally, a repeated application of Theorem 6 yields the claimed result.

3.3 Extension to String Alignment Under Integer Weights

Let us now consider the problem of computing an alignment of two strings S and T, under integer weights w_{match}, w_{mis} and w_{gap} – one may assume that $2w_{match} > 2w_{mis} \geq w_{gap}$ [32]. In this problem, the goal is to compute a highest scoring path from $v_{0,0}$ to $v_{m,n}$ in the following modification $\hat{G}_{S,T}$ of $G_{S,T}$. Edges of the form $v_{i,j}v_{i+1,j}$ and $v_{i,j}v_{i,j+1}$ have weight w_{gap}, while edges of the form $v_{i,j}v_{i+1,j+1}$ have weight w_{match} if $T[i] = S[i]$ and w_{mis} otherwise.

Tiskin shows in Section 6.1 of his monograph [32] that the alignment problem between strings S and T, can be reduced to the LCS problem between strings S' and T', obtained as follows. First, replace every letter a in S or in T by the string $\$^{\mu}a^{\nu-\mu}$, where $\$ \notin \Sigma$ and

$$\frac{\mu}{\nu} = \frac{w_{mis} - 2w_{gap}}{w_{match} - 2w_{gap}}.$$

Then, if one defines matrix $\hat{H}_{S,T}$ over $\hat{G}_{S,T}$ analogously to the definition of $H_{S,T}$ over $G_{S,T}$, we have that $\hat{H}_{S,T}(i,j) = \frac{1}{\nu} \cdot H_{S',T'}(\nu i, \nu j)$.

We maintain the same information as in the previous subsections, making sure that each fragment of each partition is a multiple of ν. At scale 0, we have only two options about how $P_{A,B}$ can look like, despite it being a $\nu \times \nu$ matrix; its structure only depends on whether $A = B$ or not. We precompute such possible $P_{A,B}$'s. This way, upon an update on S or T, updating scale 0 requires $\mathcal{O}(n\nu)$ time. At every other scale, the total length of the involved strings has just blown up by a ν multiplicative factor and hence the total update time is

$\mathcal{O}(n\nu \log^2(n\nu))$. The same reasoning shows that the preprocessing time is

$$\mathcal{O}\left(\sum_{s=0}^{\log n}\left(\frac{nw}{2^s w}\right)^2 \cdot 2^s w \cdot \log(2^s w)\right) = \mathcal{O}(n^2 w \log n \log w).$$

We summarize the results of this section in the following theorem.

▶ **Theorem 8.** *Given two strings S and T and integer weights w_{match}, w_{mis} and w_{gap}, bounded by w, the alignment score of S and T as they undergo insertions, deletions and substitutions of letters can be maintained in $\mathcal{O}(nw \log^2(nw))$ time per operation after an $\mathcal{O}(n^2 w \log n \log w)$-time preprocessing. The actual alignment can be retrieved in time $\mathcal{O}(nw \log^2(nw))$. In addition, the following queries are supported:*

- *the score of any semi-local string alignment can be computed in $\mathcal{O}(\log(nw)/\log\log(nw))$ time,*
- *the score of any fragment-to-fragment alignment can be computed in $\tilde{\mathcal{O}}(nw)$ time.*

4 Handling Large Weights

In this section, we describe an algorithm for string alignment that only relies on the planarity of $\hat{G}_{S,T}$. This algorithm outperforms the one from Theorem 8 when the alignment weights cannot be transformed to integers bounded by (roughly) $\sqrt{n}$.

Instead of computing a highest scoring path, we can reduce the problem to computing a shortest path in the alignment DAG. Given w_{match}, w_{mis} and w_{gap}, we define $w'_{match} = 0$, $w'_{mis} = w_{match} - w_{mis}$ and $w'_{gap} = \frac{1}{2}w_{match} - w_{gap}$. Then, a shortest path with respect to the new weights (of length W), corresponds to a highest scoring path with respect to the original weights (of score $\frac{1}{2}(m+n)w_{match} - W$).

4.1 Data Structures for Planar Graphs

Let us first introduce some data structures for shortest paths in planar graphs.

MSSP. The *multiple-source shortest paths* (MSSP) data structure of Klein [22] represents all shortest path trees rooted at the vertices of a single face f in a planar graph G of size n using a persistent dynamic tree. It can be constructed in $\mathcal{O}(n \log n)$ time, requires $\mathcal{O}(n \log n)$ space, and can report the distance between any vertex of f and any other vertex in G in $\mathcal{O}(\log n)$ time. The actual shortest path p can be retrieved in time $\mathcal{O}(\rho \log\log n)$, where ρ is the number of edges of p.

FR-Dijkstra. Let us consider a subgraph P of a planar graph G, and a face f of P. The *dense distance graph* of P with respect to f, denoted $DDG_{P,f}$ is a complete directed graph on the set of vertices F that lie on f. Each edge (u, v) has weight $d_P(u, v)$, equal to the length of the shortest u-to-v path in P. $DDG_{P,f}$ can be computed in time $\mathcal{O}((|F|^2 + |P|) \log |P|)$ using MSSP. In their seminal paper, Fakcharoenphol and Rao [12] designed an efficient implementation of Dijkstra's algorithm on any union of DDGs – it is nicknamed FR-Dijkstra. The algorithm exploits the fact that, due to planarity, certain submatrices of the adjacency matrix of $DDG_{P,f}$ satisfy the Monge property. We next give a – convenient for our purposes – interface for FR-Dijkstra, which was essentially proved in [12], with some additional components and details from [20, 27].

▶ **Theorem 9** ([12, 20, 27]). *Given a set of DDGs with $\mathcal{O}(M)$ vertices in total (with multiplicities), each having at most m vertices, we can (independently) preprocess each DDG with k vertices in time and extra space $\mathcal{O}(k \log k)$, so that, after this preprocessing, Dijkstra's algorithm can be run on the union of any subset of these DDGs with $\mathcal{O}(N)$ vertices in total (with multiplicities) in time $\mathcal{O}(N \log N \log m)$.*

▶ Remark 10. For an improvement in the logarithmic factors of Theorem 9 see [13].

4.2 Direct Application to String Alignment

Our approach is essentially the same as the one for dynamic distance oracles in planar graphs due to Klein [22], with extensions in [19, 20, 10]. We want to maintain a data structure that enables us to compute the length of the shortest $v_{0,0}$-to-$v_{m,n}$ path. However, instead of a single update to the graph, we have a batch of $\mathcal{O}(m+n)$ updates for each update to one of the strings. We rely on the fact that the updates to the graphs are clustered in a constant number of rows/columns of $\hat{G}_{S,T}$ in order to process them more efficiently compared to simply using dynamic distance oracles for planar graphs in a black-box manner.

Let us consider a partition of $\hat{G}_{S,T}$ into $\mathcal{O}((n/r)^2)$ pieces of size $\Theta(r) \times \Theta(r)$ each. We consider the vertices that lie on the infinite face of each piece as its boundary nodes. Then, as each piece has $\mathcal{O}(r)$ boundary vertices, the total number of boundary vertices is $\mathcal{O}(n^2/r)$. We compute the MSSP data structure and the DDG for each piece with respect to its outer face. Note that the shortest path from $v_{0,0}$ to $v_{m,n}$ can be decomposed to subpaths $p_1, \ldots, p_k$ such that each p_i lies entirely within some piece P_i and p_i's endpoints are boundary nodes of P_i. Thus, we can compute the length of the shortest $v_{0,0}$-to-$v_{m,n}$ path by running FR-Dijkstra from $v_{0,0}$ in the union of all DDGs in $\tilde{\mathcal{O}}(n^2/r)$ time. In order to retrieve the actual shortest path, we can refine the DDG edges of the shortest $v_{0,0}$-to-$v_{m,n}$ path to the actual underlying edges using the MSSP data structures for the respective pieces.

Each update to one of the strings affects a constant number of rows or of columns of the original matrix and these are covered by $\mathcal{O}(n/r)$ pieces. The MSSP data structures and DDGs for these pieces can be recomputed using MSSP and preprocessed for efficient shortest path computations in $\tilde{\mathcal{O}}(\frac{n}{r} \cdot r^2) = \tilde{\mathcal{O}}(nr)$ time. The balance is at $n^2/r = nr$, which yields $r = \sqrt{n}$, so the time per operation is $\tilde{\mathcal{O}}(n^{3/2})$. If a piece grows (resp. shrinks) too much, we break it into two pieces (resp. merge it with an adjacent piece and split in the middle) and recompute and preprocess the DDGs for the affected pieces. We obtain the following result.

▶ **Theorem 11.** *Given two strings S and T and alignment weights w_{match}, w_{mis}, and w_{gap}, the optimal alignment of S and T as they undergo insertions, deletions, and substitutions of letters can be maintained in $\tilde{\mathcal{O}}(n^{3/2})$ time per operation after an $\tilde{\mathcal{O}}(n^2)$-time preprocessing.*

5 Final Remarks

There has been a recent series of breakthrough papers on approximating the edit distance and length of the LCS, see e.g. [3, 2, 8, 16, 14, 6]. It is natural to ask about the maintenance of an approximation of the edit distance or LCS in the setting of dynamic strings.

References

1 Amir Abboud, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Ryan Williams. Simulating branching programs with edit distance and friends: or: a polylog shaved is a lower bound made. In *48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016*, pages 375–388. ACM, 2016. doi:10.1145/2897518.2897653.

2 Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Polylogarithmic approximation for edit distance and the asymmetric query complexity. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010*, pages 377–386. IEEE Computer Society, 2010. `doi:10.1109/FOCS.2010.43`.

3 Alexandr Andoni and Krzysztof Onak. Approximating edit distance in near-linear time. *SIAM Journal on Computing*, 41(6):1635–1648, 2012. `doi:10.1137/090767182`.

4 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM Journal on Computing*, 47(3):1087–1097, 2018. `doi:10.1137/15M1053128`.

5 Philip Bille and Martin Farach-Colton. Fast and compact regular expression matching. *Theoretical Computer Science*, 409(3):486–496, 2008. `doi:10.1016/j.tcs.2008.08.042`.

6 Mahdi Boroujeni, Masoud Seddighin, and Saeed Seddighin. Improved algorithms for edit distance and LCS: beyond worst case. In *31st ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1601–1620. SIAM, 2020. `doi:10.1137/1.9781611975994.99`.

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

8 Diptarka Chakraborty, Debarati Das, Elazar Goldenberg, Michal Koucký, and Michael E. Saks. Approximating edit distance within constant factor in truly sub-quadratic time. In *59th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2018*, pages 979–990. IEEE Computer Society, 2018. `doi:10.1109/FOCS.2018.00096`.

9 Timothy M. Chan and Mihai Pǎtraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *21st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010*, pages 161–173. SIAM, 2010. `doi:10.1137/1.9781611973075.15`.

10 Panagiotis Charalampopoulos, Shay Mozes, and Benjamin Tebeka. Exact distance oracles for planar graphs with failing vertices. In *30th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*, pages 2110–2123. SIAM, 2019. `doi:10.1137/1.9781611975482.127`.

11 Maxime Crochemore, Gad M. Landau, and Michal Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM Journal on Computing*, 32(6):1654–1673, 2003. `doi:10.1137/S0097539702402007`.

12 Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *Journal of Computer and System Sciences*, 72(5):868–889, 2006. `doi:10.1016/j.jcss.2005.05.007`.

13 Paweł Gawrychowski and Adam Karczmarz. Improved bounds for shortest paths in dense distance graphs. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018*, volume 107 of *LIPIcs*, pages 61:1–61:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018. `doi:10.4230/LIPIcs.ICALP.2018.61`.

14 Elazar Goldenberg, Robert Krauthgamer, and Barna Saha. Sublinear algorithms for gap edit distance. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019*, pages 1101–1120. IEEE Computer Society, 2019. `doi:10.1109/FOCS.2019.00070`.

15 Szymon Grabowski. New tabulation and sparse dynamic programming based techniques for sequence similarity problems. *Discrete Applied Mathematics*, 212:96–103, 2016. `doi:10.1016/j.dam.2015.10.040`.

16 MohammadTaghi Hajiaghayi, Masoud Seddighin, Saeed Seddighin, and Xiaorui Sun. Approximating LCS in linear time: Beating the $\sqrt{n}$ barrier. In *30th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*, pages 1181–1200. SIAM, 2019. `doi:10.1137/1.9781611975482.72`.

17 Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975. `doi:10.1145/360825.360861`.

18 Yusuke Ishida, Shunsuke Inenaga, Ayumi Shinohara, and Masayuki Takeda. Fully incremental LCS computation. In *15th International Symposium on Fundamentals of Computation Theory, FCT 2005*, volume 3623 of *LNCS*, pages 563–574. Springer, 2005. doi:10.1007/11537311_49.

19 Giuseppe F. Italiano, Yahav Nussbaum, Piotr Sankowski, and Christian Wulff-Nilsen. Improved algorithms for min cut and max flow in undirected planar graphs. In *43rd ACM Symposium on Theory of Computing, STOC 2011*, pages 313–322. ACM, 2011. doi:10.1145/1993636.1993679.

20 Haim Kaplan, Shay Mozes, Yahav Nussbaum, and Micha Sharir. Submatrix maximum queries in Monge matrices and partial Monge matrices, and their applications. *ACM Transactions on Algorithms*, 13(2):26:1–26:42, 2017. doi:10.1145/3039873.

21 Sung-Ryul Kim and Kunsoo Park. A dynamic edit distance table. *Journal of Discrete Algorithms*, 2(2):303–312, 2004. doi:10.1016/S1570-8667(03)00082-0.

22 Philip N. Klein. Multiple-source shortest paths in planar graphs. In *16th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005*, pages 146–155. SIAM, 2005. URL: http://dl.acm.org/citation.cfm?id=1070432.1070454.

23 Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582, 1998. doi:10.1137/S0097539794264810.

24 Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics. Doklady*, 10:707–710, 1966.

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

26 Gaspard Monge. *Mémoire sur la théorie des déblais et des remblais.* De l'Imprimerie Royale, 1781.

27 Shay Mozes and Christian Wulff-Nilsen. Shortest paths in planar graphs with real lengths in $O(n \log^2 n/\log\log n)$ time. In *18th Annual European Symposium on Algorithms, ESA 2010, Part II*, volume 6347 of *LNCS*, pages 206–217. Springer, 2010. doi:10.1007/978-3-642-15781-3_18.

28 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. doi:10.1016/0022-2836(70)90057-4.

29 David Sankoff. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences of the United States of America*, 69(1):4–6, 1972. doi:10.1073/pnas.69.1.4.

30 Peter H. Sellers. On the theory and computation of evolutionary distances. *SIAM Journal on Applied Mathematics*, 26(4):787–793, 1974. doi:10.1137/0126070.

31 Alexander Tiskin. Longest common subsequences in permutations and maximum cliques in circle graphs. In *17th Annual Symposium on Combinatorial Pattern Matching, CPM 2006*, volume 4009 of *LNCS*, pages 270–281. Springer, 2006. doi:10.1007/11780441_25.

32 Alexander Tiskin. Semi-local string comparison: algorithmic techniques and applications, 2007. arXiv:0707.3619.

33 Alexander Tiskin. Semi-local longest common subsequences in subquadratic time. *Journal of Discrete Algorithms*, 6(4):570–581, 2008. doi:10.1016/j.jda.2008.07.001.

34 Alexander Tiskin. Semi-local string comparison: Algorithmic techniques and applications. *Mathematics in Computer Science*, 1(4):571–603, 2008. doi:10.1007/s11786-007-0033-3.

35 Alexander Tiskin. Faster subsequence recognition in compressed strings. *Journal of Mathematical Sciences*, 158(5):759–769, 2009. doi:10.1007/s10958-009-9396-0.

36 Alexander Tiskin. Periodic string comparison. In *20th Annual Symposium on Combinatorial Pattern Matching, CPM 2009*, volume 5577 of *LNCS*, pages 193–206. Springer, 2009. doi:10.1007/978-3-642-02441-2_18.

37 Alexander Tiskin. Fast distance multiplication of unit-Monge matrices. *Algorithmica*, 71(4):859–888, 2015. doi:10.1007/s00453-013-9830-z.

38 Taras K. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics*, 4:52–57, 1968. doi:10.1007/BF01074755.

39 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974. doi:10.1145/321796.321811.

40 Sun Wu, Udi Manber, and Eugene W. Myers. A subquadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996. doi:10.1007/BF01942606.